On the Generalizability of Neural Program Models with respect to Semantic-Preserving Program Transformations

Md Rafiqul Islam Rabin\textsuperscript{a}, Nghi D. Q. Bui\textsuperscript{c}, Ke Wang\textsuperscript{b}, Yijun Yu\textsuperscript{d}, Lingxiao Jiang\textsuperscript{c}, Mohammad Amin Alipour\textsuperscript{a}

\textsuperscript{a}University of Houston, United States.
\textsuperscript{b}Visa Research, United States.
\textsuperscript{c}Singapore Management University, Singapore.
\textsuperscript{d}The Open University, UK.

Abstract

Context: With the prevalence of publicly available source code repositories to train deep neural network models, neural program models can do well in source code analysis tasks such as predicting method names in given programs that cannot be easily done by traditional program analysis techniques. Although such neural program models have been tested on various existing datasets, the extent to which they generalize to unforeseen source code is largely unknown. Objective: Since it is very challenging to test neural program models on all unforeseen programs, in this paper, we propose to evaluate the generalizability of neural program models with respect to semantic-preserving transformations: a generalizable neural program model should perform equally well on programs that are of the same semantics but of different lexical appearances and syntactical structures. Method: We compare the results of various neural program models for the method name prediction task on programs before and after automated semantic-preserving transformations. We use three Java datasets of different sizes and three state-of-the-art neural network models for code, namely code2vec, code2seq, and GGNN, to build nine such neural program models for evaluation. Results: Our results show that even with small semantically preserving changes to the programs, these neural program models often fail to generalize their performance. Our results also suggest that neural program models based on data and control dependencies in programs generalize better than neural program models based only on abstract syntax trees (ASTs). On the positive side, we observe that as the size of the training dataset grows and diversifies the generalizability of correct predictions produced by the neural program models can be improved too. Conclusion: Our results on the generalizability of neural program models provide insights to measure their limitations and
provide a stepping stone for their improvement.

Keywords: neural models, code representation, model evaluation, program transformation, generalizability

1. Introduction

Abundance of publicly available source code repositories has enabled a surge in data-driven approaches to programs analysis tasks. Those approaches aim to discover common programming patterns for various downstream applications that are not easily achievable via traditional program analysis techniques, e.g., prediction of data types in dynamically typed languages, detection of the variable naming issues, or repair of software defects. The advent of deep neural networks has accelerated the innovation in this area and has greatly enhanced the performance of these approaches. The performance of deep neural networks in cognitive tasks such as method name prediction or variable naming has reached or exceeded the performance of other data-driven approaches. The performance of neural networks has encouraged researchers to increasingly adopt neural networks in program analysis tasks, giving rise to increasing uses of neural program models.

While the performance of neural program models continues to improve, the extent to which they can generalize to new, unseen programs is still unknown, even if the programs are in the same programming language. This problem is of more importance if we want to use them in downstream safety-critical tasks, such as malware detection and automated defect repair. This problem is particularly hard, as the interpretation of neural models that constitute the core reasoning engine of neural program models remains challenging—especially for the complex neural networks (e.g., RNN) that are commonly used in the proposed neural program models.

A comprehensive understanding of the extent of generalizability of neural program models would help developers to know when to use data-driven approaches and when to resort to traditional deductive methods of program analysis. It would also help researchers to focus their efforts on devising new techniques to alleviate the shortcomings of existing neural program models. Lack of knowledge about the limits of neural program models may exaggerate their capability and cause careless applications of the neural program models on the domains that they are not suited for; or, spending time and efforts on developing neural program models while a traditional, more understandable technique can perform equally well or better.
Recently, we have seen a growing interest in the rigorous evaluation of neural program models. Wang and Christodorescu [50] compared the robustness of different program representations under compiler optimization transformations. They found that the program representations based on static code features are more sensitive to such changes than dynamic code features. Allamanis [1] evaluated the impact of code duplication in various neural program models and found that code duplication in the training and test datasets inflated the performance of almost all current neural program models. More recently, preliminary studies in this field started to emerge; e.g., Rabin et al. [40] proposed the idea of testing neural program models using semantic-preserving transformations; Bui et al. [13] measured the impact of a specific code fragment by deleting it from the original source code; Zhang et al. [58] proposed a sampling approach to generate adversarial examples for code classification models; and Compton et al. [15] showed that the obfuscation of variable names makes a model on source code more robust with less bias towards variable names. Further, Yefet et al. [56] followed and proposed adversarial example generation for neural program models using prediction attribution [45]; Ramakrishnan et al. [41] increased robustness of neural representations of code by adding semantically equivalent programs to the training data; and Bielik and Vechev [11] proposed an approach for increasing the robustness of neural program models for type prediction based on finding prediction attribution, adversarial training, and refining source code representations. Although these studies share the similar ultimate goal of evaluating and improving the performance of neural program models with respect to unseen programs, there is still a lack of systematic quantifiable metrics to measure the extent to which the neural program models can generalize to unseen programs, and it would not be fair either to evaluate a neural program model against all possible unseen programs that it was not designed for.

**Goal.** In this paper, we attempt to understand the limits of generalizability of neural program models by comparing their behavior before and after semantic-preserving program transformations. That is, how the results of a neural program model generalize to a semantically-equivalent program. By limiting unseen programs to semantically equivalent ones and controlling the semantic-preserving program transformations, we are able to provide a fair, systematic, quantifiable metric for evaluating the generalizability of a neural program model.

In this paper, we report the results of a study on the generalizability of three highly-cited neural program models: code2vec [7], code2seq [9], and GGNN [19]. To evaluate their generalizability, we transform programs in the original datasets for testing to generate semantically-equivalent counter-
parts. We employ six semantic-preserving transformations that impact the structure of programs (i.e. abstract syntax trees) with varying degrees, ranging from common refactoring, e.g., variable renaming, to more intrusive changes such as changing for-loops to while-loops.

Our results suggest that all neural program models evaluated in this study are sensitive to the semantic-preserving transformations; that is, the output of the neural program model would be different on transformed programs compared to its output on the original programs. This sensitivity remains an issue even in the cases of small changes to the programs, such as renaming variables or reordering independent statements in a basic block. Moreover, our results suggest that neural program models (e.g., GGNN) that encode data and control dependencies in programs generalize better than the neural program models that are solely based on abstract syntax trees, and in most cases the generalizability of a neural program model can be improved with the growth in the size of training datasets.

The results of this study reveal that the generalizability of neural program models is still far from ideal and require more attention from the research community to devise more generalizable models of source code, or designing pre-processing techniques, e.g. canonicalizing program representations, to increase immunity of neural program models to such program transformations.

Compared to closely related work by Yefet et al. [56] and Ramakrishnan et al. [41] where their goals are adversarial code generation and increasing robustness of neural program models, this paper provides a complementary view to the evaluation of neural program models by focusing on the evaluation of generalizability of neural program models with a large number of transformations, and in-depth analysis of changes in their behavior on transformed programs. This paper also evaluates the impact of the size of datasets and programs on the generalizability of neural program models.

Contributions. This paper makes the following contributions.

• We introduce the notion of generalizability with respect to semantic-preserving transformations for neural program models.
• We perform a large-scale study to evaluate the generalizability of state-of-the-art neural program models. We also provide insights into the generalizability of existing neural program models and discuss their practical implications.
• We provide an in-depth analysis of changes in the prediction and evaluate the impact of the size of datasets and programs on the generalizability of neural program models.
2. Motivating Example & Definition

We use code2vec [7] for exposition in this section. The code2vec [7] is a recent, highly-cited (200+ citations as of Nov. 2020) neural program model that predicts the name of a Java method given the body of the method. Such a neural program model can assist developers in classification of methods, code similarity detection, and code search.

Figure 1 shows two semantically-identical methods that implement compareTo functionality. The only difference between them is in the name of one of the variables. The left snippet in Figure 1 uses other, while the code on the right uses var. However, the code2vec outputs, i.e., predictions, on these semantically equivalent programs are drastically different. code2vec predicts the snippet on the left to be compareTo function, and the function on the right to be getCount. It seems that the predictions of code2vec rely much on the identifier names (e.g., other). This reliance would make code2vec susceptible to a common refactoring such as variable renaming, and would make it not generalize to the code snippets that are semantically the same, but are different syntactically, even under common transformations.

Lack of generalizability would lead to distrust in the neural program models and hamper their wider adoption and application. If such neural program models were to be deployed in the problem settings wherein higher levels of generalizability are required, e.g., malware detection and bug repair, it would be much better for the neural program models to demonstrate a high level of generalizability with respect to certain metrics.

Generalizability. We define generalizability as the capability of a neural program model to return

---

1 var is not an uncommon identifier name in Java as it appears in the training vocabulary of the datasets. At the time of writing, a search on the GitHub returns more than 75K Java classes that use this identifier.
the same results under semantic-preserving transformations.

In this paper, we differentiate generalizability from the term robustness that is commonly used in the neural network literature for two main reasons. First, robustness is usually defined in the face of adversarial examples that have security implications, while we do not generate adversarial examples. Second, robustness implies imperceptible differences in the two focal inputs (e.g., minor pixel changes in two images) that are hard to attain in a sparse domain such as source code; the program transformations used in this paper often lead to perceptible changes of textual appearances and syntactic structures in program code. We also note that our definition of generalizability differs from what is used in that evaluates the usefulness of a neural program model in various downstream tasks.

Together with clearly defined semantic-preserving program transformations and their change impact on the prediction results of neural program models (cf. Section 4), we aim to provide a systematic quantifiable way to measure the generalizability of neural program models, and thus shed lights on their capabilities and limits for future improvements. With the extensibility of the program transformations and the measurements of their change impact, our evaluation approach may also be extended to measure the generalizability of neural program models more comprehensively in the near future.

3. Background

Most neural program models use neural network classifiers in their core components that take a code snippet or a whole program as an input, and make predictions about some of its characteristics; e.g., a bug prediction classifier that predicts the buggy-ness of statements in the input program.

Performance of a neural program model depends on three main factors: quality of data (i.e., source code for this study), the representation of data for the neural network, and the neural network characteristics and its training parameters.

Quality of the data is concerned with the representativeness of data, and proper cleaning and preprocessing of the data. Currently, most studies use open-source projects usually in mainstream programming languages, e.g., C#, Java, C, or JavaScript. The available datasets for these tasks are still very immature and not standardized, and their quality is somewhat unknown. For example, a recent study by Allamanis showed that virtually all available datasets suffer from code duplication that can greatly impact the performance of neural program models.
The second factor affecting the performance of neural program models is source code representations. Since neural networks need to take vectors of numbers as direct inputs, source code embeddings are used to produce a vector representation of source code. The representation determines which program features to include and how they should be represented in the vector embeddings. The representations can be broadly categorized into two categories: static and dynamic. Static program representations consider only the features that can be extracted from parsing texts of the programs, while dynamic representations include some features pertaining to the real executions of the programs.

The third factor impacting the performance of a neural program model is the characteristics—e.g., type, topology, and hyper-parameters—of the neural networks it uses. There are numerous choices of network architectures each with different properties. Currently, the class of recurrent neural networks (e.g., LSTM) and graph neural networks are among the most popular architectures in neural program models [7, 6, 19].

4. Evaluation Approach

Our approach for evaluating neural program models relies on a metamorphic relation that states: the outputs of a neural program model should not differ on semantically-equivalent programs. To this end, the evaluation approach is divided into two main steps: (1) generating new programs using semantic-preserving transformations, and (2) comparing the outputs of a neural program model before and after the transformations to compute generalizability metrics. We describe these steps in the rest of this section.

4.1. Target Downstream Task

We use the method name prediction task [2, 5] in this work to evaluate the generalizability of neural program models. The goal of the task is to predict the name of a method given the body of the method. This task has several applications such as code search [35], code summarization [5], and code analogies [7]. Figure 1 depicts an example of this task wherein neural program models are given a method body and return candidate names for the method body, i.e., compareTo and getCount. This task has been used as the downstream task to evaluate several state-of-the-art neural program models [4, 7, 6].
4.2. Transformations

In this work, we only evaluate neural program models that take a method body as their input, therefore, we use the following set of transformations that are applicable to method-level code to generate semantically-equivalent methods. This set includes transformations ranging from common refactoring like variable renaming to more intrusive ones like loop exchange. The goal is to evaluate the generalizability of neural program models under a wide range of semantic-preserving changes to the structure of a method.

• **Variable Renaming** (VN) is a refactoring that renames the name of a variable in a method. The new name of the variable will be in the form of `varN` for a value of N such that N has not been defined in the scope. VN is a widely-used refactoring for methods.

• **Permute Statement** (PS) swaps two independent statements (i.e., with no data or control dependence) in a basic block of a method.

• **Unused Statement** (UN) inserts an unused string declaration to a randomly selected basic block in a method. Unused variables in methods are a common malpractice by developers.

• **Loop Exchange** (LX) replaces `for` loops with `while` loops or vice versa.

• **Switch to If** (SF) replaces a `switch` statement in a method with an equivalent `if` statement.

• **Boolean Exchange** (BX) switches the value of a boolean variable in a method from `true` to `false` or vice versa, and propagates this change in the method to ensure a semantic equivalence of the transformed method with the original method.

Note that each transformation has different impact on the structure of methods as follows.

• The **Variable Renaming** transformation only changes the terminal values and does not affect the structure of an AST.

• The **Permute Statement** transformation does not change the nodes, rather it only reorders two subtrees in an AST.

• The **Unused Statement** transformation adds a few nodes into an AST, which increases the number of paths in the AST.
• The *Loop Exchange* transformation extensively impacts an AST by removing and inserting nodes.

• The *Switch to If* transformation also impacts the AST of a method substantially by removing and inserting nodes.

• The *Boolean Exchange* transformation alters the value of `true` or `false` and modifies the structure of an AST by removing or inserting unary-not nodes.

### 4.3. Generalizability Metrics

In this study, we define a few metrics to measure different results of a neural program model for transformed programs and thus to quantify the generalizability of the neural program model.

Specifically, suppose $M$ denotes a set of methods, given a semantic-preserving program transformation $T$ that takes a method and creates a set $M' = \bigcup_{m \in M} T(m)$ of transformed methods, and a neural program model $NPM : M \rightarrow L$, where $L$ denotes a set of labels, maps methods to labels. We evaluate the generalizability of $NPM$ with respect to the transformation $T$, by comparing $NPM(m)$ and $NPM(m')$ for $m' \in T(m)$ for $m \in M$. Ideally, the neural program model should produce the same results on both $m$ and $m'$, that is $NPM(m) = NPM(m')$. We define the following metrics.

**Prediction Change Percentage.** We compute the prediction change percentage as follows:

$$PCP = \frac{|\{m' \in M'|NPM(m) \neq NPM(m')\}|}{|\{m' \in M'\}|} * 100.$$  

(1)

The lower values of PCP for $NPM$ would suggest higher a degree of its generalizability with respect to the transformation.

**Types of Changes.** Considering that the correctness of predicted labels of the $NPM$, five types of changes can happen:

(1) a correct prediction remains correct after the transformation,

(2) a correct prediction changes to a wrong prediction after the transformation,

(3) a wrong predicted label remains the same wrong label after the transformation,

(4) a wrong prediction changes to a correct prediction after the transformation,

(5) a wrong predicted label changes into a different, yet still wrong label after the transformation.
We use the following five metrics to denote the proportion of each of these cases in the experiments. CCP, CWP, WWSP, WCP, and WWDP respectively denote the percentage of correct predictions that stay correct, the percentage of correct predictions that become wrong, the percentage of wrong predictions that stay to the same wrong prediction after the transformation, the percentage of wrong predictions that become correct, and the percentage of wrong predictions that change to a different wrong prediction after the transformation.

**Precision, Recall, and F1-Score.** We also use the traditional sub-token metrics (precision, recall and F1-score) as commonly used in the literature for the method name prediction task [7, 6] in this generalizability study. Suppose, \( tp \) denotes the number of true positive sub-tokens, \( fp \) denotes the number of false positive sub-tokens, and \( fn \) denotes the number of false negative sub-tokens in the predicted method names.

- **Precision** indicates the percentage of predicted sub-tokens that are true positives. It is the ratio of the correctly predicted positive sub-tokens to the total number of predicted positive sub-tokens: 
  \[
  \text{Precision} = \frac{tp}{tp + fp}
  \]

- **Recall** indicates the percentage of true positive sub-tokens that are correctly predicted. It is the ratio of the correctly predicted positive sub-tokens to the total number of sub-tokens in actual method names: 
  \[
  \text{Recall} = \frac{tp}{tp + fn}
  \]

- **F1-Score** is the harmonic mean of precision (P) and recall (R):
  \[
  F_1-\text{Score} = \frac{2 \cdot P \cdot R}{P + R} = 2 \cdot \frac{P \cdot R}{P + R}
  \]

For example, a predicted name `result_compute` has two sub-tokens `result` and `compute`, and is considered as an exact match of the ground-truth name `computeResult` which also has the same two sub-tokens (ignoring the case and the ordering of the tokens). Similarly, a predicted name `compute` has 100% precision but only 50% recall with respect to the same ground truth, and `compute_model_result` has 100% recall but only 67% precision.

5. Experimental Setting

5.1. Subject Neural Program Models

The task of method name prediction [5] has attracted some attention recently. We use three neural program models that use different code representations and neural network characteristics for the task: `code2vec` [7], `code2seq` [6], and `GGNN` [19].
code2vec \cite{7} uses a bag of AST paths to model source code. Each path consists of a pair of terminal nodes and the corresponding path between them in the AST. Each path, along with source and destination terminals, is mapped into its vector embeddings which are learned jointly with other network parameters during training. The separate vectors of each path-context are then concatenated to a single context vector using a fully connected layer which is learned during training with the network. An attention vector is also learned with the network; it is used to score each path-context and aggregate multiple path-contexts to a single code vector representing a method body. After that, the model predicts the probability of each target method name given the code vector of the method body via a softmax-normalization between the code vector and each of the embeddings of all possible target method names.

While code2vec uses monolithic path embeddings and only generates a single label at a time, the code2seq \cite{6} model uses an encoder-decoder architecture to encode paths node-by-node and generate labels as sequences at each step. In code2seq, the encoder represents a method body as a set of AST paths where each path is compressed to a fixed-length vector using a bi-directional LSTM which encodes paths node-by-node. The decoder uses attentions to select relevant paths while decoding, and predicts sub-tokens of a target sequence at each step when generating the method name.

In GGNN \cite{19}, a variety of semantic edges are added into the AST of a method body to construct a graph, and the Gated Graph Neural Network (GGNN) is applied to encode such graphs \cite{4}. The initial embedding for a node of the graph is the concatenation between the node type embedding and node token embedding. Then a fixed number of message passing steps are applied for a node to aggregate the embeddings of its neighbors. The output of the GGNN encoder is then fed into a bi-directional LSTM decoder to generate the method name as a language model of sub-tokens \cite{19}.

5.2. Datasets

We have used the code2seq dataset for training neural program models for the study. There are three Java datasets based on the GitHub projects: JAVA-SMALL, JAVA-MED, and JAVA-LARGE.

- **JAVA-SMALL**: This dataset contains 9 Java projects for training, 1 for validation and 1 for testing. Overall, it contains about 700K methods. The compressed size is about 366MB and the extracted size is about 1.9GB.

- **JAVA-MED**: This dataset contains 800 Java projects for training, 100 for validation and 100 for testing. Overall, it contains about 4M methods. The compressed size is about 1.8GB and
Table 1: Performance of trained models for method name prediction in the testing dataset.

<table>
<thead>
<tr>
<th>Model</th>
<th>Dataset</th>
<th># Original methods in the testing dataset</th>
<th>Precision</th>
<th>Recall</th>
<th>$F_1$-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>code2vec</td>
<td>Java-Small</td>
<td>44426</td>
<td>28.36</td>
<td>22.37</td>
<td>25.01</td>
</tr>
<tr>
<td>code2vec</td>
<td>Java-Med</td>
<td>351628</td>
<td>42.55</td>
<td>30.85</td>
<td>35.76</td>
</tr>
<tr>
<td>code2vec</td>
<td>Java-Large</td>
<td>370930</td>
<td>45.17</td>
<td>32.28</td>
<td>37.65</td>
</tr>
<tr>
<td>code2seq</td>
<td>Java-Small</td>
<td>44426</td>
<td>46.30</td>
<td>38.81</td>
<td>42.23</td>
</tr>
<tr>
<td>code2seq</td>
<td>Java-Med</td>
<td>351628</td>
<td>59.94</td>
<td>48.03</td>
<td>53.33</td>
</tr>
<tr>
<td>code2seq</td>
<td>Java-Large</td>
<td>370930</td>
<td>64.03</td>
<td>55.02</td>
<td>59.19</td>
</tr>
<tr>
<td>GGNN</td>
<td>Java-Small</td>
<td>44426</td>
<td>49.12</td>
<td>47.18</td>
<td>48.59</td>
</tr>
<tr>
<td>GGNN</td>
<td>Java-Med</td>
<td>351628</td>
<td>58.30</td>
<td>47.49</td>
<td>52.34</td>
</tr>
<tr>
<td>GGNN</td>
<td>Java-Large</td>
<td>370930</td>
<td>60.76</td>
<td>50.32</td>
<td>55.23</td>
</tr>
</tbody>
</table>

the extracted size is about 9.3GB.

- **Java-Large**: This dataset contains 9000 Java projects for training, 200 for validation and 300 for testing. Overall, it contains about 16M methods. The compressed size is about 7.2GB and the extracted size is about 37GB.

5.3. Training Models per Datasets

The authors of code2vec and code2seq have made the source code public for training and evaluating their models. For GGNN, the implementation of the network is available but the code graph generation is not; so we re-implement the step to generate graphs. We use the parser SrcSlice, an extension of SrcML, to produce data dependency edges among AST nodes for training GGNN.

We train each model for the method name prediction task with the configurations described in their original papers on each of the three aforementioned datasets, and thus construct three code2vec, three code2seq, and three GGNN neural program models. Similar to the state-of-the-art approaches, i.e. [7, 6], we train models on the training set, tune on the validation set for maximizing $F_1$-score, and finally report results on the unseen testing set. Table 1 summarizes the performance of trained models for method name prediction on the testing set. While the performance of our trained models for code2seq is on par with to the ones reported in the corresponding paper [6], the performance of code2vec did not reach the performance reported in [7], due to the differences in the dataset. However, the performance of our trained code2vec models is similar to the one reported in [6]. For GGNN, the performance is reasonably different from what were reported in [19].

---

1. [https://github.com/srcML/srcSlice](https://github.com/srcML/srcSlice)
2. [https://www.srcml.org/](https://www.srcml.org/) (+400 node types for supporting multiple programming languages)
mainly three reasons: (1) the ASTs produced by our parser are different, (2) the extraction of some types of semantic edges proposed in [19] requires expensive analysis of the methods; therefore, we implemented and included only a subset (seven out of ten) of semantic edges into the ASTs when constructing the graphs, and (3) the datasets are different.

5.4. Population of Transformed Programs

We have used our own tool based on the JavaParser library to transform Java methods. Henceforth we use terms program and method interchangeably. Two authors were involved in the implementation, testing and code review. We have performed manual inspection of sample transformed programs to ensure correctness of the transformations.

We have applied the applicable transformations to the methods available in the testing data of the three datasets mentioned in Section 5.2. The number of original methods in our study is 1,415,116 overall. Overall, the number of original methods with incorrect predictions is, on average, 2.8 times higher than the number of methods with correct predictions.

We create a set of single-place transformed programs (Table 2) by applying transformations to each eligible location in methods separately resulting in 2,822,810 transformed methods; e.g., if a method has three eligible locations for a transformation, we would generate three distinct methods by transforming each individual location separately. The types and number of applicable transformations vary from a method to another. Therefore, in our approach, different methods, based on the language features that they use, produce a different number of transformed programs. In total, the number of transformed programs generated from the programs with incorrect initial predictions is much higher (4.2x and higher) than the number of transformed programs generated from the programs with correct initial predictions, which may suggest that programs with correct predictions may be smaller and simpler.

Artifacts. The source code of the transformation tool and transformed datasets used in this paper are publicly available at https://github.com/mdrafiqulrabin/tnpa-generalizability/.

5.5. Research Questions

In this paper, we seek to answer the following research questions.
Table 2: Prediction Change Percentage (PCP) across all models, datasets, and transformations.

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Dataset</th>
<th># Original methods</th>
<th># Transformed methods</th>
<th>Prediction change (%) (PCP)</th>
<th>code2vec</th>
<th>code2seq</th>
<th>GGNN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>code2vec</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Variable Renaming</td>
<td>JAVA-SMALL</td>
<td>31113</td>
<td>124123</td>
<td>54.92</td>
<td>57.16</td>
<td>28.17</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JAVA-MED</td>
<td>239961</td>
<td>771208</td>
<td>46.55</td>
<td>48.75</td>
<td>35.96</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JAVA-LARGE</td>
<td>252725</td>
<td>916565</td>
<td>42.06</td>
<td>47.04</td>
<td>31.92</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Weighted Average =</td>
<td>44.85</td>
<td>48.46</td>
<td>33.39</td>
</tr>
<tr>
<td>Boolean Exchange</td>
<td>JAVA-SMALL</td>
<td>1158</td>
<td>1519</td>
<td>53.85</td>
<td>54.31</td>
<td>29.37</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JAVA-MED</td>
<td>6407</td>
<td>8840</td>
<td>50.35</td>
<td>44.71</td>
<td>33.74</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JAVA-LARGE</td>
<td>8868</td>
<td>12107</td>
<td>47.80</td>
<td>51.43</td>
<td>31.98</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Weighted Average =</td>
<td>49.21</td>
<td>48.98</td>
<td>32.50</td>
</tr>
<tr>
<td>Loop Exchange</td>
<td>JAVA-SMALL</td>
<td>3699</td>
<td>5169</td>
<td>59.38</td>
<td>52.54</td>
<td>31.66</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JAVA-MED</td>
<td>17107</td>
<td>23533</td>
<td>62.77</td>
<td>45.29</td>
<td>36.67</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JAVA-LARGE</td>
<td>35565</td>
<td>49665</td>
<td>46.52</td>
<td>42.51</td>
<td>31.75</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Weighted Average =</td>
<td>52.25</td>
<td>44.01</td>
<td>33.22</td>
</tr>
<tr>
<td>Switch to If</td>
<td>JAVA-SMALL</td>
<td>246</td>
<td>259</td>
<td>68.73</td>
<td>61.78</td>
<td>31.45</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JAVA-MED</td>
<td>3312</td>
<td>3839</td>
<td>59.91</td>
<td>41.60</td>
<td>43.75</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JAVA-LARGE</td>
<td>10478</td>
<td>11165</td>
<td>30.33</td>
<td>29.08</td>
<td>45.30</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Weighted Average =</td>
<td>38.42</td>
<td>32.78</td>
<td>44.82</td>
</tr>
<tr>
<td>Permute Statement</td>
<td>JAVA-SMALL</td>
<td>3997</td>
<td>9169</td>
<td>72.80</td>
<td>57.32</td>
<td>26.36</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JAVA-MED</td>
<td>16150</td>
<td>43711</td>
<td>65.44</td>
<td>42.64</td>
<td>34.09</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JAVA-LARGE</td>
<td>21956</td>
<td>74973</td>
<td>64.38</td>
<td>41.93</td>
<td>26.32</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Weighted Average =</td>
<td>65.35</td>
<td>43.27</td>
<td>29.02</td>
</tr>
<tr>
<td>Unused Statement</td>
<td>JAVA-SMALL</td>
<td>44420</td>
<td>44420</td>
<td>39.97</td>
<td>45.60</td>
<td>28.34</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JAVA-MED</td>
<td>351621</td>
<td>351621</td>
<td>35.80</td>
<td>40.25</td>
<td>42.79</td>
<td></td>
</tr>
<tr>
<td></td>
<td>JAVA-LARGE</td>
<td>370927</td>
<td>370927</td>
<td>31.21</td>
<td>37.44</td>
<td>35.07</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Weighted Average =</td>
<td>33.82</td>
<td>39.20</td>
<td>38.51</td>
</tr>
</tbody>
</table>

RQ1 How do the transformations impact the predictions of neural program models in the single-place transformed dataset?

RQ2 When do the transformations affect neural program models the most?

RQ3 How does the method length impact the generalizability of neural program models?

RQ4 What are the trends in types of changes?

RQ5 How do the transformations affect the precision, recall and $F_1$-score of the neural program models?

6. Results

6.1. RQ1: Impact of Transformations on the Predictions of Neural Program Models

Table 2 shows the prediction change percentage (PCP) of the neural program models for each transformation and dataset. In this table, “# Original methods” denotes the number of methods eligible for the corresponding transformation, “# Transformed methods” denotes the number of
methods generated as the result of applying the corresponding transformations on the original methods, and “Prediction change (%)” denotes PCP as defined in Section 4. “Weighted Average” provides the weighted average of PCP for each transformation and neural models. The bold values in the Table 2 highlight the minimum value of PCP for the transformations. Since a transformation can be applied in more than one place separately in a method body, the number of transformed methods can be larger than the number of original methods.

As Table 2 depicts, all neural program models are likely to susceptible to semantic-equivalent transformations; however, the impact of transformations on PCP differs among different neural networks and datasets. Overall, GGNN seems less prone to prediction changes; in 14 out of 18 cases, PCP in GGNN is significantly less than code2vec and code2seq. Moreover, in four out of six transformations, the weighted average of PCP for GGNN is lower than the rest.

**Observation 1:** In most cases, GGNN seems less susceptible to prediction changes under semantic-preserving transformations, compared to code2vec and code2seq.

Within code2vec, code2seq, and GGNN, the PCP trend varies for different transformations and datasets. code2vec is comparatively most sensitive to Permute Statement on all datasets. On the other hand, code2seq is most vulnerable to Switch to If in JAVA-SMALL, Variable Renaming in JAVA-MED, and Boolean Exchange in JAVA-LARGE. In GGNN, Switch to If is the most powerful transformation on all datasets. In most cases, for code2vec and code2seq, the PCP for Unused Statement is comparatively less than the other transformations, except for code2seq in JAVA-LARGE where Switch to If is less sensitive. In GGNN, Permute Statement is a comparatively less powerful transformation than others on all datasets. Overall, based on the weighted average, it is likely that, code2vec is most sensitive to Permute Statement and least sensitive to Unused Statement, code2seq is most sensitive to Boolean Exchange and least sensitive to Switch to If, and GGNN is most sensitive to Switch to If and least sensitive to Permute Statement.

Based on the weighted average, GGNN performs worst for Switch to If and Unused Statement transformations. These two transformations add some additional nodes and paths in the AST. For code2vec and code2seq, if models give less attention to those new paths, then the change can less effective. However, GGNN works by using a message passing mechanism among the nodes with a limited number of passing steps. In Unused Statement, because there is some irrelevant information added into the code, the passing steps in GGNN can capture this information and ignore other useful information, thus having a strong impact on the prediction results. In Switch to If, because
the structure of the AST is modified by adding and removing nodes, and GGNN is a node-based
method, i.e., combining node information with message passing, thus the GGNN can sensitive to
node modification in the AST for Switch to If.

Table 2 also supports that, in most cases, Permute Statement is more powerful than Variable
Renaming in code2vec model whereas Variable Renaming is more powerful than Permute Statement
in code2seq model. This is probably caused by the real-value embeddings of AST paths are different
for code2vec and code2seq. In code2vec, an embedding matrix is initialized randomly for paths and
learned during training, that contains rows that are mapped to each of the AST paths. On the
other hand, in code2seq, each node of a path comes from a learned embedding matrix, and then a
bi-directional LSTM is used to encode each of the AST paths separately. The bi-directional LSTM
reads the path once from beginning to the end (as original order) and once from end to beginning
(in reverse order). Therefore, the order changes by Permute Statement may become less sensitive
to code2seq than code2vec.

Another observation is that, in most cases of code2vec and code2seq, the PCP of the transfor-
mations in Java-Small is high, and it is significantly lower on larger datasets, i.e., Java-Med,
and Java-Large. In GGNN, the PCP of the transformations shows a different trend: lowest in
Java-Small, in most cases, and highest in Java-Med.

**Observation 2:** In most cases, the effect of prediction change for code2vec and code2seq is
reduced as the dataset size increases, compared to GGNN.

6.2. RQ2: When Transformations Affect Neural Program Models the Most?
6.2.1. Single-place transformation vs. All-place transformation

In our analysis, thus far, if a program has multiple candidates for a transformation, say \( n \)
candidates, for transformation, we only apply them one at the time and end up with \( n \) distinct
transformed programs. We call this single-place transformation. Alternatively, we can apply the
transformations to all candidate locations in the program simultaneously to create only one trans-
formed program. We call this all-place transformation. We evaluate the generalizability of neural
program models under all-place transformation for the following transformations: Variable Renam-
ing, Boolean Exchange, Loop Exchange, and Switch to If. Note that the all-place transformation
is not applicable to Permute Statement and Unused Statement transformations, as we apply the
Permulate Statement on a pair of statements and the Unused Statement on a random block.
Figure 2: Prediction Change Percentage (PCP) in single-place vs. all-place transformations.

Figure 2 compares the impact of single-place transformation and all-place transformation on the prediction changes in all neural program models. For the code2vec model, the percentage of prediction change for the all-place transformation is higher than the single-place transformation by a good margin for all the cases. Similarly, for the code2seq model, the percentage of prediction change for the all-place transformation is higher than the single-place transformation by a good margin except for the case (Switch to If, Java-Small). After a closer examination of Java-Small dataset and Switch to If transformation, we observe that the number of transformed methods for all-place is only 13, which is too low to provide comparative insight. For the GGNN model, the difference between all-place transformation and single-place transformation is relatively very small compared to the code2vec and code2seq models. Even for (Boolean Exchange, Java-Small + Java-Med), (Loop Exchange, Java-Med + Java-Large), and (Switch to If, Java-Large), the percentage of prediction changes for the single-place transformation is higher than the all-place transformation. The results may suggest that the performance of GGNN under single-place transformations and all-place transformations is almost consistent.

Observation 3: While all-place transformations are more likely to induce prediction changes in code2vec and code2seq than single-place transformations, the performance of GGNN remains relatively similar under both types of transformations.

6.2.2. Correctly predicted methods vs. Incorrectly predicted methods

We also evaluate the generalizability of neural program models under correctly and incorrectly predicted methods. Figure 3 compares the impact of correctly predicted methods and incorrectly predicted methods on the prediction changes in all neural program models. In the code2vec model, the percentage of changes in predictions after transformation in the correctly predicted methods
ranges from 10.45% to 42.86%, while, in the incorrectly predicted methods, a larger portion of transformations, 38.18% to 76.00%, change the prediction of code2vec. Similarly, in the code2seq model, the percentage of changes in predictions after transformation in the correctly predicted methods and in the incorrectly predicted methods ranges from 9.19% to 36.36% and 46.66% to 62.90%, respectively. However, in the GGNN model, while the percentage of changes in predictions after transformation on the correctly predicted methods ranges from 1.90% to 8.58%, the percentages range from 31.05% to 62.01% in the incorrectly predicted methods.

**Observation 4:** It is likely that GGNN is more stable than code2vec and code2seq in the originally correct methods, and the changes in prediction happen more frequently in the originally incorrect methods for all models.

### 6.2.3. The Effect of X%-Transformation

In this section, we evaluate the generalizability of neural program models under X%-transformation for the following transformations: **Variable Renaming**, **Boolean Exchange**, **Loop Exchange**, and **Switch to If**. Note that we apply the **Permute Statement** on a pair of independent statements and the **Unused Statement** on one randomly selected block in a method, and thus X%-transformation is not suitable for them. If a transformation \( t \) is applicable to \( n \) locations in a method body, X%-transformation randomly picks \( \left\lfloor \frac{n \times X}{100} \right\rfloor \) of those locations and applies \( t \) to create a new transformed program. The number of all X%-transformed programs grows exponentially with the number of locations; therefore, to manage the complexity, in X%-transformation we randomly pick the locations in a method body, instead of considering all possible combinations, to create transformed programs. We study the X%-transformation with \( X = \{25, 50, 75\} \). For each transformation \( t \), we first create a dataset \( d_t \) that contains methods.
Table 3: The PCP for $X\%$-transformations across different datasets and models.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Transformation</th>
<th># Transformed methods</th>
<th>25% Transformation</th>
<th>50% Transformation</th>
<th>75% Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>code2vec</td>
<td>code2seq</td>
<td>GGNN</td>
</tr>
<tr>
<td>JAVA-SMALL</td>
<td>Variable Renaming</td>
<td>15637</td>
<td>63.29</td>
<td>54.96</td>
<td>29.56</td>
</tr>
<tr>
<td></td>
<td>Boolean Exchange</td>
<td>75</td>
<td>80.00</td>
<td>63.00</td>
<td>37.70</td>
</tr>
<tr>
<td></td>
<td>Loop Exchange</td>
<td>302</td>
<td>81.95</td>
<td>65.90</td>
<td>32.44</td>
</tr>
<tr>
<td></td>
<td>Switch to If</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>JAVA-MED</td>
<td>Variable Renaming</td>
<td>101003</td>
<td>54.07</td>
<td>46.51</td>
<td>37.18</td>
</tr>
<tr>
<td></td>
<td>Boolean Exchange</td>
<td>428</td>
<td>69.66</td>
<td>48.20</td>
<td>31.37</td>
</tr>
<tr>
<td></td>
<td>Loop Exchange</td>
<td>1292</td>
<td>86.03</td>
<td>55.11</td>
<td>28.72</td>
</tr>
<tr>
<td></td>
<td>Switch to If</td>
<td>98</td>
<td>82.91</td>
<td>43.62</td>
<td>55.42</td>
</tr>
<tr>
<td>JAVA-LARGE</td>
<td>Variable Renaming</td>
<td>114748</td>
<td>45.62</td>
<td>41.23</td>
<td>33.37</td>
</tr>
<tr>
<td></td>
<td>Boolean Exchange</td>
<td>642</td>
<td>71.81</td>
<td>59.03</td>
<td>28.18</td>
</tr>
<tr>
<td></td>
<td>Loop Exchange</td>
<td>2899</td>
<td>79.77</td>
<td>56.00</td>
<td>25.96</td>
</tr>
<tr>
<td></td>
<td>Switch to If</td>
<td>125</td>
<td>69.00</td>
<td>56.00</td>
<td>34.78</td>
</tr>
</tbody>
</table>

Observation 5: The performance of GGNN in terms of PCP remains similar in all cases under $X\%$-transformation, but the PCP of code2vec and code2seq for Variable Renaming increases as $X$ grows.

6.3. RQ3: Impact of Method Length on Generalizability

An important metric of interest might be the generalizability in terms of the number of statements in the methods. Figure [4] depicts the relation between the length of methods and the prediction changes percentage (i.e., PCP) in the single-place transformed data. In the figure, the...
Figure 4: Prediction Change Percentage (PCP) across the number of statements in methods.

“Number of statements in method” denotes the number of executable lines in the body of methods before the transformation.

As shown in Figure 4(a-f), in most cases, the code2vec and code2seq models exhibit notable increases in PCP for all the transformations and datasets as the number of lines in methods increases. However, looking at Figure 4(g-i), it seems that GGNN is less sensitive to the number of lines in methods compared to code2vec and code2seq with respect to the transformations.

**Observation 6:** The code2vec and code2seq show notable increases in PCP as the length of methods grows, but PCP in GGNN seems to be less sensitive to the length of methods.
Table 4: The detailed PCP across all models, datasets, and transformations.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Transformation</th>
<th>CCP</th>
<th>CWP</th>
<th>WWSP</th>
<th>WCP</th>
<th>WWDP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>code2vec</td>
<td>code2seq</td>
<td>GGNN</td>
<td>code2vec</td>
<td>code2seq</td>
</tr>
<tr>
<td></td>
<td>Java-Small</td>
<td>2.32</td>
<td>3.75</td>
<td>15.76</td>
<td>1.18</td>
<td>1.65</td>
</tr>
<tr>
<td></td>
<td>Boolean Exchange</td>
<td>3.88</td>
<td>4.54</td>
<td>22.25</td>
<td>0.72</td>
<td>2.57</td>
</tr>
<tr>
<td></td>
<td>Loop Exchange</td>
<td>1.86</td>
<td>4.24</td>
<td>16.23</td>
<td>0.74</td>
<td>0.74</td>
</tr>
<tr>
<td></td>
<td>Switch to If</td>
<td>1.54</td>
<td>2.70</td>
<td>26.61</td>
<td>1.16</td>
<td>1.54</td>
</tr>
<tr>
<td></td>
<td>Permute Statement</td>
<td>4.64</td>
<td>3.96</td>
<td>16.86</td>
<td>1.35</td>
<td>1.28</td>
</tr>
<tr>
<td></td>
<td>Unused Statement</td>
<td>8.08</td>
<td>10.40</td>
<td>20.97</td>
<td>1.73</td>
<td>3.20</td>
</tr>
<tr>
<td></td>
<td>Java-Med</td>
<td>7.56</td>
<td>9.41</td>
<td>20.39</td>
<td>2.83</td>
<td>3.90</td>
</tr>
<tr>
<td></td>
<td>Boolean Exchange</td>
<td>12.76</td>
<td>13.90</td>
<td>27.83</td>
<td>1.79</td>
<td>1.91</td>
</tr>
<tr>
<td></td>
<td>Loop Exchange</td>
<td>6.61</td>
<td>7.62</td>
<td>20.93</td>
<td>2.21</td>
<td>1.71</td>
</tr>
<tr>
<td></td>
<td>Switch to If</td>
<td>11.15</td>
<td>17.90</td>
<td>31.28</td>
<td>5.99</td>
<td>2.94</td>
</tr>
<tr>
<td></td>
<td>Permute Statement</td>
<td>11.53</td>
<td>14.55</td>
<td>25.47</td>
<td>4.31</td>
<td>1.57</td>
</tr>
<tr>
<td></td>
<td>Unused Statement</td>
<td>16.90</td>
<td>21.58</td>
<td>25.07</td>
<td>3.24</td>
<td>5.22</td>
</tr>
<tr>
<td></td>
<td>Java-Large</td>
<td>15.40</td>
<td>14.54</td>
<td>14.57</td>
<td>3.25</td>
<td>4.69</td>
</tr>
<tr>
<td></td>
<td>Boolean Exchange</td>
<td>11.33</td>
<td>10.14</td>
<td>13.97</td>
<td>2.09</td>
<td>2.92</td>
</tr>
<tr>
<td></td>
<td>Loop Exchange</td>
<td>19.81</td>
<td>18.87</td>
<td>13.29</td>
<td>3.00</td>
<td>2.49</td>
</tr>
<tr>
<td></td>
<td>Switch to If</td>
<td>48.30</td>
<td>52.23</td>
<td>9.30</td>
<td>5.63</td>
<td>5.28</td>
</tr>
<tr>
<td></td>
<td>Permute Statement</td>
<td>11.13</td>
<td>13.35</td>
<td>21.89</td>
<td>6.66</td>
<td>2.17</td>
</tr>
<tr>
<td></td>
<td>Unused Statement</td>
<td>24.31</td>
<td>26.57</td>
<td>22.28</td>
<td>3.75</td>
<td>5.95</td>
</tr>
</tbody>
</table>
6.4. RQ4: Trends in the Types of Changes

Table 4 shows the full breakdown of the proportion of different types of changes after the transformation of methods. In this experiment, we use the same single-place transformed data that have been used for the PCP in Table 2. In `code2vec` and `code2seq`, the value of CCP increases with increase in the size of datasets. It may suggest that with a larger dataset the neural program model can generalize the correct predictions better.

In addition, we calculate \( \frac{CWP}{CCP+CWP} \) to approximate the ratio of cases that the neural program model’s prediction switches from correct to wrong after transformations with respect to all the cases whose initial predictions are correct. The ratio helps us to simplify the comparison of (in)generalizability across different models. On average, 23% and 20% of cases, the neural program model switches from a correct prediction to a wrong one in `code2vec` and `code2seq`, respectively. In GGNN, on the other hand, this switch happens in less than 5% of transformations.

Similarly, \( \frac{WCP}{WWP+WWDP+WCP} \) approximates the ratio of cases switching from a wrong prediction to a correct prediction after transformations with respect to all the cases whose initial prediction are wrong. In `code2vec` and `code2seq`, a transformation switches from a wrong prediction to correct prediction in less than 3% of cases, however, this switch happens in around 1% of transformations for GGNN. Higher \( \frac{CWP}{CCP+CWP} \) than \( \frac{WCP}{WWP+WWDP+WCP} \) implies that transformations are likely to reduce the overall performance of the neural program models.

**Observation 7:** Transformations are likely to decrease the overall performance of neural program models, and they are more likely to change the correct prediction in `code2vec` and `code2seq` than GGNN, while the generalizability of `code2vec` and `code2seq` can be compensated by larger datasets more than GGNN.

6.5. RQ5: Impact of the Transformations on Precision, Recall, and \( F_1 \)-Score

The performance of neural program models in the literature are often measured in classic metrics, such as precision, recall, and \( F_1 \)-score. In particular for the method name prediction task trained for `code2vec`, `code2seq` and GGNN, subtoken-level comparison is used to calculate the metrics; i.e., the method names in both predicted results and ground-truth names are split into individual tokens for the measurements (cf. the definitions in Section 4.3).

We also study the impact of the program transformations on the performance of neural program models in terms of these classic metrics. Table 5 shows the changed precision, recall, and \( F_1 \)-scores for the programs transformed by different transformations. In this...
In this paper, we study the current state of generalizability in neural program models built on code2vec, code2seq, and GGNN. Although limited, it provides interesting insights. In this section,
we discuss why neural networks have become a popular, or perhaps the de-facto, tool for processing programs, and what are the implications of using neural networks in processing source code.

Neural networks constitute a powerful class of machine learning models with a large hypothesis class. For instance, a multi-layer feed-forward network is called a universal approximator, meaning that it can essentially represent any function [25]. Unlike traditional learning techniques that require extensive feature engineering and tuning, deep neural networks facilitate representation learning. That is, they are capable of performing feature extraction out of raw data on their own [32]. Given a sufficiently large dataset, neural networks with adequate capabilities can substantially reduce the burden of feature engineering. Availability of a large number of code repositories makes data-driven program analysis a good application of neural networks. However, it is still unknown if neural networks are the best way to process programs [24] vs. [29].

Although the large hypothesis class of neural networks and feature learning make them very appealing to use, the complex models built by neural networks are still too difficult to understand and interpret. Therefore, as we apply neural networks in program analysis, we should develop specialized tools and techniques to enhance their interpretability, generalizability and robustness.

7.1. Generalizability vs. Interpretability vs. Robustness and Others

Interpretability studied in the literature may help to build more understandable neural networks, revealing the limits and strengths of the networks, and thus to some extent, it helps to evaluate and understand the generalizability of the networks. However, our study of generalizability with respect to program transformation provides a different perspective complement to interpretability; the approach may have the potential in the future to help identify interpretable code elements by measuring the impact of certain types of code transformations.

As mentioned in Section [2] there is a substantial line of work on evaluating the robustness of neural networks especially in the domain of vision and pattern recognition [48]. The key insight in such domains is that small, imperceptible changes in input should not impact the result of output. While this observation can be true for domains such as vision, it might not be directly applicable to the discrete domain of neural program models, since some minor changes to a program can drastically change the semantic and behavior of the program. Quantifying the imperceptibility and many other aspects of source code is our future research goal.
7.2. Are we there yet?

Are neural program models ready for widespread use in program analysis? The neural program models in our experiments are brittle to even very small changes in the methods. The semantic-preserving transformations can change the outputs of the neural program models in 26% to 73% of cases. Although our findings are limited to only one task, they suggest caution. The literature lacks techniques for rigorous evaluation of neural program models. The recent line of work by Nghi et al. 13 in interpretability of neural program models, Rabin et al. 40 in testing them, and Yefet et al. 56 are much needed steps in a right direction.

7.3. Code Representation

The performance of models used in neural program models, such as ones used in this study, is relatively low compared to the performance of neural models in domains such as natural language understanding 42, text classification 31. To improve their performance, we would need novel code representations that better capture interesting characteristics of programs.

8. Related Work

Robustness of Neural Networks. There is a substantial line of work on the robustness of artificial intelligence (AI) systems in general and deep neural networks in particular. Szegedy et al. 16 is the first to discover that deep neural networks are vulnerable to small perturbations that are imperceptible to human eyes. They developed the L-BFGS method for the systematic generation of such adversarial examples. Goodfellow et al. 21 proposes a more efficient method, called the Fast Gradient Sign Method that exploits the linearity of deep neural networks. Many following up works 30, 57, 14, 18, 57, 61, 56 further demonstrated the severity of the robustness issues with a variety of attacking methods and defenses. While aforementioned approaches only apply to models for image classification, new attacks have been proposed that target models in other domains, such as natural language processing 33, 27, 62 and graphs 16, 63.

The automated verification research community has proposed techniques to offer guarantees for the robustness of neural networks by adapting bounded model checking 44, abstract interpretation 20, and Satisfiability Modulo Theory 26. Amershi et al. 8 study the challenges in developing AI solutions and Zhang et al. 59 survey testing of machine-learning systems.

Models of Code. Early works directly adopted NLP models to discover textual patterns existed in the source code 22, 39. Those methods, unfortunately, do not account for the structural
information programs exhibit. Following approaches address this issue by generalizing from the abstract syntax trees \cite{36, 38, 7, 6}. As Graph Neural Networks (GNN) \cite{13} have been gaining increasing popularity due to its remarkable representation capacity, many works have leveraged GNN to tackle challenging tasks like program repair and bug finding, and obtained quite promising results \cite{4, 53, 17, 34, 12, 17, 47}. Besides, the attention mechanism \cite{9} has been applied into GNNs to improve the performance further \cite{10, 54, 60}. It is very interesting to see how the attention can help to explain the output of the neural models \cite{48, 55, 13}. In parallel, Wang et al. developed a number of models \cite{49, 51, 52} that feed off the run time information for enhancing the precision of semantic representation for model inputs.

9. Threats to Validity

There are various threats to the validity of our approach.

**Limited Data and Evaluation Scope.** We only evaluated the generalizability of neural program models built on code2vec, code2seq, and GGNN, for one task in Java programs. Therefore, our results may not generalize to other neural program models or other tasks or other programming languages. We leave the evaluation of the general applicability of our approach as future work.

**Transformations.** The proposed transformations in this paper impact program ASTs in varying degrees. Some of the transformations, e.g. variable renaming, are common refactoring techniques. However, these transformations may not represent many possible transformations in other domains. We will instantiate and extend our approach with other transformations from other domains.

**Internal Validity.** Some bugs may exist in the toolchain and neural program models implemented in this paper. To reduce the probability of bugs, two authors reviewed the code and manually inspected a sample of transformed programs to ensure the reliability of transformations.

10. Conclusion & Future Work

In this paper, we perform a large-scale, systematic evaluation of the generalizability of state-of-the-art neural program models built on code2vec, code2seq, and GGNN. In particular, we apply six semantic-preserving program transformations to produce new programs on which we expect the neural program models to keep their original predictions. We find that such program transformations frequently sway the predictions of these neural program models, indicating serious generalization
issues that could negatively impact the wider applications of deep neural networks in program analysis tasks. Although neural program models that encode more program dependency information and are trained with larger datasets may exhibit more generalizable behavior, their generalizability is still limited. We believe this work provides a systematic approach and metrics for evaluating neural program models, and can motivate future research on training not only accurate but also generalizable deep models of code. Future work that includes more semantic-preserving and even some semi-semantic-preserving transformations in our approach and adapts more fine-grained prediction change metrics may further extend the applicability of our approach to various neural program models designed for different tasks. We also plan to explore using transformed programs to improve the generalizability of the neural program models.

Acknowledgments

This research is supported by the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 Grant No. 19-C220-SMU-002 and the Research Lab for Intelligent Software Engineering (RISE) Operational Fund from the School of Computing and Information Systems (SCIS) at Singapore Management University (SMU). We also thank the anonymous reviewers for their insightful comments and suggestions, and thank the authors of previous related works for sharing data and models.

References


doi: 10.1145/3212695


doi: 10.1145/3290353


doi: 10.1109/ICSE-SEIP.2019.00042


