Enhacing Out-Of-Distribution Accuracy in Code Completion via Data Augmentation

Melanie Becker[1] and Luka Jovanović[1]

December 2023

1 Abstract

Code completion models have been the subject of much research in recent years due to their potential for enhancing productivity in software development. However, these models commonly fall victim to the distribution-shift problem, in which train and test data come from the same distribution and thereby underprepare models to work with code they encounter in the real world. This investigation builds on the work of the `WILDS` project, which tackles this problem by aggregating data across thousands of distinct GitHub repositories. We aimed to enhance their approach by mutating their training data using data augmentation, with the hope that doing so would introduce additional variation and bolster out-of-distribution predictions. Our findings indicate that standard code augmentation techniques are insufficient for expanding the distributions of already robust training sets. We recommend that future research explore more sophisticated augmentations that better mimic real-world code to enhance the efficacy of this approach.

2 Introduction

2.1 Overview

Code completion in machine learning is a task in which a model predicts the next token in a line of written code based on context given by the sequence of tokens preceding it. Developing a model proficient in this task is a topic of interest in contemporary research due to its immense utility in enhancing programmer productivity. By alleviating the burden of typing out code and correcting basic syntactical errors, such models allow programmers to concentrate on more complex aspects of their work and can ultimately augment both the quality and quantity of code they produce.

[1]Both authors contributed equally to this work.

Training such a model, however, is no simple task. A central problem across domains in machine learning is that data used to train models often comes from the same distribution as the data used to test it. In the context of code completion, this could mean that the same programmers who wrote the code comprising the training data also wrote the code in the test set. As a result, when the model using this data is deployed in the real world, it may perform quite poorly for programmers whose coding styles deviate substantially from what the model saw during production.

2.2 The WILDS Approach

This distribution-shift problem is the focus of the `WILDS` project (Koh et al., 2021), whose work this investigation builds upon. `WILDS` explores solutions to the distribution-shift problem in ten different datasets aimed at tackling challenges ranging from identifying tumors in photos of cell tissue to using the text of Amazon customer reviews to predict their corresponding star ratings. Specifically, we extend the work WILDS has done with the `py150` dataset (Raychev et al., 2016), which trains models on a code completion task in the Python programming language.

In tackling the distribution-shift problem in code completion, Koh and colleagues aggregated program snippets from thousands of disparate GitHub repositories into a single dataset, considering each repository a distinct distribution. This dataset is an alternate version of `py150` they dub `py150-WILDS`. Models trained on this set are tested on snippets from repositories separate to those the model was trained on to evaluate out-of-distribution (OOD) performance. Henceforward, when we cite the "`WILDS` approach", we refer to this strategy of addressing the distribution-shift problem by purposefully collecting data from multiple unrelated distributions in the real world.

After training on nearly 80,000 code snippets across thousands of repositories, the best scores the `WILDS` approach yields are 69.6% accuracy on OOD validation data generally and 67.9% accuracy for method and class tokens specifically. Method and class token predictions are separately examined as their own subpopulation because they constitute the most common use of code completion functionality in the real world (Koh et al., 2021).

2.3 Our Approach

We endeavored to further improve these scores by combining the `WILDS` approach with another common solution to the distribution-shift problem: data augmentation (DA). This technique involves artificially introducing more variation to existing training data by subjecting it to systematic manipulations. Ideally, this makes models less prone to overfitting and more generalizable to unseen situations.

While DA has been transformative in computer vision (Shorten & Khoshgoftaar, 2019), its efficacy in natural language processing (NLP) (Pellicer et al., 2023) and, by extension, code generation, is less clear. Given that the study of

DA in code generation tasks is particularly nascent, we hope that this research will contribute to this budding conversation and shed light on the role of DA when working with program data.

To the authors' knowledge, no existing research has investigated the efficacy of DA in code completion tasks. Indeed, it was difficult to identify research using DA to bolster any code generation task at all. We managed to identify one study working with Java and Python snippets in bug detection and problem classification tasks which attributed data augmentation to a 6.28% increase in test accuracy and a 26.06% increase in robustness (Dong et al., 2023). The study defines robustness as the model's ability to generalize to "more diverse unseen data", suggesting the augmentations' potential in enhancing a model's OOD accuracy. We employed the same augmentation strategies as these researchers in an effort to validate their promising findings. If successful, such an approach could save researchers significant time in gathering real-world data by providing them an avenue for creating data of a similar caliber artificially. Ultimately, however, we demonstrate that applying straightforward augmentations like those employed by Dong and colleagues are likely not effective in enhancing the `WILDS` approach.

2.4 Limitations

Apart from the dearth of existing findings available to inform our experimental design, an undertaking of this nature comes with a number of limitations due to its inherent complexity.

First, the general task of applying DA to code poses a unique set of challenges. Primarily, code snippets must follow a very rigid set of syntactic rules to remain valid. Unlike in image-processing tasks, where no amount of distortion can change the fact that an image is still an image, it is easy to mutate a code snippet to the point that it no longer obeys constraints of its original programming language. Introducing augmentations that create syntax errors might ultimately train the model to make erroneous predictions, ultimately making it quite difficult to devise new augmentations to try.

Similarly, past research suggests that the naturalness of artificially generated code snippets is important in determining their effectiveness in training models (Yang et al., 2022; Yu et al., 2022). This makes sense; after all, training and validation data consist of code written by humans, so it follows that any augmentations performed on train data should still result in code that a human might write, restricting consideration of potential augmentations even further.

Lastly, our particular investigation came with a number of limitations due to the nature of the `py150-WILDS` dataset. The code snippets came pre-formatted in such a way that all indentations were removed, with no simple way to determine where indentations originally lay. As proper indentation is crucial to correct Python syntax, much work went into artificially inserting indentations into the snippets to make them parsable by library functions expecting a valid Python string as argument.

In addition, the dataset is massive. Running an experiment on the full

py150-WILDS set takes a competent machine about 10 hours, and a set containing versions of snippets both before and after DA take twice that time. As a result, because the timespan of the study was restricted to a single month, we operated only on smaller subsets of the data and had to prioritize amongst the myriad experimental approaches we considered pursuing.

3 Methods

3.1 Model and Parameters

Given that our aim was to improve the current top score in WILDS while only changing the data, we adopted the same model and parameters used in that project. We thus employed the pre-trained code completion model CodeGPT and trained it using a learning rate of $8 \cdot 10^{-5}$. No regularization was applied. We ran the model for three epochs for each experiment as the original authors did and validated this approach by confirming that OOD validation scores did not improve given more epochs, as we discuss below.

3.2 Preprocessing

Before training the model on our data, it was first tokenized using the built-in Python tokenizer and CodeGPT tokenizer. This step transformed the data into the form the model expected to receive and allowed it to learn contextual relationships between individual tokens within each code snippet.

3.3 Algorithm

We conduct our experiments using empirical risk minimization (ERM), which is currently the top-scoring algorithm for the py150 data in WILDS. This algorithm involves adjusting model parameters based on those that minimize training loss in the hopes that this will also minimize loss on unseen data (Chaudhuri et al., 2011).

3.4 Dataset

The py150-WILDS is a set of almost 150,000 Python code snippets divided across more than 5,400 repositories for train data and an additional 2,700 repositories for OOD test data. In the full set, about 80,000 of the snippets are devoted to training the model, about 5,000 of the snippets to OOD validation, 5,000 to in distribution (ID) validation, 20,000 to ID test data, and about 40,000 to OOD test data. As mentioned above, we primarily worked with smaller subsets of py150-WILDS due to time constraints. Each allocated the same proportions of snippets to train, ID/OOD test, and ID/OOD validation as the full set. One was a very small set used for testing many ideas relatively quickly, containing 1500 training snippets. We reasoned that working with roughly 2% of the overall training data was not so insignificant that our results would be meaningless,

but was small enough to proceed through experiments in a timely fashion. We ran results which seemed particularly promising or important on a larger, more representative set of 24,000 training snippets for validation. We refer to these sets as `data1500` and `data24k`, respectively.

Snippets in both `data1500` and `data24k` are randomly sampled from the original dataset. For consistency, the smaller sets were shared between the researchers via Google drive to ensure that all experiments were performed on the exact same set of original code snippets.

In all datasets, a snippet is classified as "in distribution" if it comes from the same GitHub repository as one of the snippets used in the training data. It is "out of distribution" if it does not.

3.5 Augmentations

Originally, we performed a set of 15 augmentations on the data. Each of these was derived from the work of Dong and colleagues, who reported promising results by using them in a code generation task. Details on these augmentations are given in **Table 1** below.

After some experimentation with these augmentations, we devised four additional augmentations of our own and tested them. These augmentations are enumerated in **Table 2**.

All augmentations were applied as evenly as possible on the training data in our experiments. `enhance_if` and `apply_plus_zero_math` are exceptions to this, as they are far more situational and cannot apply to a given snippet unless that snippet has an if statement or an arithmetic operation, respectively. As a result, these augmentations frequently occupied a smaller percentage of augmented data compared to other refactoring methods.

We apply DA under two principal philosophies, which we refer to as "combined" and "mixed". In the "combined" approach, we include an original snippet as well as its augmented counterpart. This approach is more traditional of data augmentation and corresponds to the `MIXUP` technique reported by Dong and colleagues. In the "mixed" strategy, we explore the effects of including either the original version of a snippet *or* the augmented version, but not both, in the training data.

Augmentation	Description
rename_argument	Renames a randomly selected argument of the method to a synonym.
return_optimal	Modifies a method's return statement to handle null values by returning 0 if the return object is None, otherwise returning the original return object.
add_arguments	Adds additional arguments to the method's signature using synonyms of existing arguments.
rename_api	Renames an API call within the method to a synonym.
rename_local_variable	Renames a randomly selected local variable to a synonym.
add_local_variable	Adds a new local variable definition in the method. The new variable is a synonym of an existing variable.
rename_method_name	Renames the method name to a synonym.
enhance_if	Randomly modifies an if statement by changing comparison operators and operands, while maintaining the original logic's functionality.
add_print	Adds a print statement.
duplication	Duplicates a local variable definition.
apply_plus_zero_math	Applies a '+0' operation to numerical operations.
dead_branch_if_else	Introduces an if-else statement that effectively does nothing.
dead_branch_if	Introduces an if statement that effectively does nothing.
dead_branch_while	Introduces a while loop that effectively does nothing.
dead_branch_for	Introduces a for loop that effectively does nothing.

Table 1: Augmentations inspired by the work of Dong et al.

Augmentation	Description
insert_random_function	Inserts a random function with a coherent naming relationship among its name, arguments, and variables, accompanied by comments that explain its actions. This function executes random operations on the arguments and variables, then returns a random number of values.
insert_random_class	Inserts a random class characterized by a consistent naming relationship among its name, fields, and methods, complete with comments that elucidate its functionality. This class includes a constructor for initializing fields and various methods, some of which may be adorned with decorators.
insert_safe_random_space	Inserts n random spaces into safe locations in the given code snippet. Safe locations include spaces around operators, after commas, and around braces and brackets.
create_typo	Creates a typo in a variable within the method by swapping two adjacent characters at a random position.

Table 2: New augmentations we added

4 Experiments

In this section, we present the various experiments we performed in our investigation as well as their results, which we summarize in tables in subsection 4.6. For each experiment, we report four distinct measures: ID and OOD overall accuracy, and ID and OOD Method/Class accuracy. While the primary focus of our investigation is OOD performance, ID scores are also provided for robustness.

We followed the protocol of the `WILDS` team by ordering our results in terms of OOD Method/Class accuracy, which the `WILDS` authors identified as the most important sub-metric in code completion. Additionally, we followed their lead

by reporting the best score obtained for each category across all epochs.

4.1 Different Seeds

To contextualize our findings, we ran the exact same processed data on three different seeds: 0, 123, and 234. We did this both for `data1500` and `data24k`, and used these results as a metric to gauge whether augmentation strategies we tried displayed meaningful deviation from the control. Results are displayed in **Tables 3 and 4**. Note that for `data1500` we used `all-1 mixed` data, and for `data24k` we examined the control. We were primarily interested in investigating how much the model could deviate on the exact same set of preprocessed data when the seed is varied, so the specifics of the set of preprocessed data were deemed unimportant. `all-1` "mixed" and the control also yielded very similar results in the experiments we ran, further validating this approach.

For `data1500`, seeds differed in OOD Method/Class accuracy by about 0.5-1%, whereas overall OOD accuracy scores fell within 0.2% across epochs. In the case of `data24k`, OOD Method/Class deviations fell within 0.5%, and overall OOD scores fell within 0.1% of each other. These closer scores are likely due to the inclusion of more data points significantly decreasing the likelihood of anomalous variation.

Time constraints impeded us from running all of our experiments on multiple seeds for validation, but these trials give some insight into how to interpret which experimental results might be meaningful enough to warrant future study. Note that seed 0 was used for the remainder of the experiments reported in this section.

4.2 Combined vs. Mixed Augmentation Strategies

The "combined" augmentation approach did not perform nearly as well as the "mixed" strategy, and neither approach outperformed the control on either dataset. At first glance, **Table 5** appears to suggest that the `all-1` and `all-1-enh` augmentation approaches combined with the "mixed" strategy might have a slight advantage over the control. In the former, we applied a random one of Dong and colleagues' augmentations to each snippet. In the latter, we used these original refactoring methods in conjunction with the four new augmentations we devised. However, it should be noted that both approaches held a smaller-than-1% advantage over the control in OOD Method/Class accuracy, which is no larger than what can be observed by simply changing the seed on the same data. In addition, the control still performed better in overall OOD accuracy for `data1500`. These findings suggest that no strategy offered a meaningful improvement in model performance. In line with this, **Table 6** shows that the control outperforms all approaches in `data24k`; the advantages observed in `data1500` vanish in the more representative set.

Furthermore, both tables consistently show that the "combined" approach always performed worse than the other strategies, with performance gaps larger

than the deltas identified in the seed experiments. Whereas the "mixed" strategy appeared to perform almost identically to the control, the "combined" approach lagged definitively behind. This result is not consistent with the work of Dong and colleauges, who found that including both source and transformed code in the training data improved model accuracy.

The remaining subsections address the various strategies we employed in attempting to bridge this performance gap.

4.3 More Epochs

Based on the results of the previous section, we reasoned that with additional variation introduced into the data, the model might need more epochs in order to learn how to correctly contextualize tokens in code snippets. As such, we ran an experiment with `data1500` that allowed the model to run for 10 epochs. As displayed in **Table 7**, however, this does not appear to be the case. We ran `all-1` using the "mixed" strategy, since this appeared to be the superior augmentation approach, for 10 epochs. The table reports the score for each category's best epoch. Even taking scores from all 10 epochs into account, the control remained superior in every category. Furthermore, we note here that all of these top scores were found within the first three epoch, supporting our choice to train the model on three epochs for the remainder of the experiments.

4.4 Multiple Augmentations Per Snippet

In seeing that the augmentations were not enhancing model accuracy, even given many epochs to do so, we hypothesized that perhaps the variation introduced by adding only a single augmentation per snippet did not alter code snippets enough to make their augmented versions introduce any meaningful amount of new variation to the data. To test this conjecture, we augmented each snippet with *multiple* randomly chosen augmentations. In doing so, we aimed to investigate whether making every snippet very different from its original might vary snippets enough to broaden the distribution of the training data, thereby preparing the model to make more accurate OOD predictions.

However, **Tables 5 and 6** illustrate that `all-2-enh` and `all-3-enh`, in which we randomly applied two and three snippets per augmentation respectively, did worse than applying a single augmentation (`all-1` and `all-1-enh`) in both datasets. In fact, the performance gaps between these approaches and the control are larger than the gaps obtained by running the the same set of processed data on a different seeds. This suggests that the inferior performance of this approach might not be due to random chance, and that applying these augmentations may have even actively hindered the models performance rather than simply failing to help it.

4.5 Additional Augmentations

Finally, when applying multiple augmentations per snippet failed, we speculated that the augmentations currently being applied might not have been complex enough to broaden the training distribution at all. As such, we developed a few more complicated augmentations of our own. Namely, we created augmentations `insert_random_class` and `insert_random_function`, which generated entirely new python classes and functions and inserted them into code snippets. Generated functions are given random numbers of local variables and operations performed on them. Names are derived from random word generation functions in existing Python libraries, and artificially inserted classes contain a random number of generated member functions and fields.

We also added two additional simplistic augmentations: `create_typo`, which swaps adjacent character pairs in random variables to mimic the common typo, and `insert_random_safe_space`, which adds an additional space character to the program without violating pythonic syntax constraints. In line with the idea that simple augmentations do not introduce additional meaningful difference to the data, adding these augmentations did not change results obtained from simply working with the original refactoring methods.

The enhanced augmentation set is deployed in the experiments with the `-enh` suffix in **Tables 5 and 6**. As with `all-1`, `all-1-enh` performed marginally better than the control in Method/Class OOD accuracy in `data1500`, but this advantage disappears in `data24k`. It does not beat the control in overall OOD accuracy in either dataset, nor does it beat the standard `all-1` approach on any metric.

4.6 Results

In this subsection, we present the tables containing the results of the experiments described in subsections 4.1-4.5. To facilitate the interpretation of these results, we provide below a description of what each column in the tables represents.

- **Experiment:** This label identifies each experiment, using the format *a*-*b*-*c*. The exception is the 'control' experiment, which uses the original (non-augmented) dataset. In the *a*-*b*-*c* format, *a* denotes the augmentations applied to the dataset, *b* indicates the number of augmentations per data point, and *c* is an optional component specifying the augmentation set. The absence of *c* implies the use of the initial set of 15 augmentations (as described in Table 1). If *c* is labeled as 'enh', it signifies the use of the enhanced set, combining elements from Tables 1 and 2. For example, 'all-1' means all augmentations from Table 1 were applied, with each code snippet receiving a single augmentation.

- **Strategy:** Describes the approach used, categorized as either 'mixed' or 'combined', as detailed in Section 4.2.

- **Epoch:** This represents the epoch number in the training process of the

model. An epoch refers to one complete cycle through the entire training dataset.

- **No. Epochs:** This represents the total number of epochs used in the training process of the model.
- **ID M/C:** The test score representing in-distribution Method/Class accuracy.
- **ID All:** The test score representing in-distribution overall accuracy.
- **OOD M/C:** The test score representing out-of-distribution Method/Class accuracy.
- **OOD All:** The test score representing out-of-distribution overall accuracy.

Experiment	Strategy	Epoch	ID M/C	ID All	OOD M/C	OOD All
all-1	mixed	0	Δ0.2%	Δ3.7%	Δ1%	Δ0.1%
all-1	mixed	1	Δ0.4%	Δ2.9%	Δ0.5%	Δ0.2%
all-1	mixed	2	Δ0.3%	Δ1.5%	Δ0.6%	Δ0.2%

Table 3: Largest difference between seeds 0, 123, and 234 for **data1500**

Experiment	Epoch	ID M/C	ID All	OOD M/C	OOD All
control	0	Δ0.8%	Δ0.9%	Δ0.4%	Δ0.0%
control	1	Δ1.1%	Δ0.8%	Δ0.5%	Δ0.1%
control	2	Δ1.3%	Δ0.7%	Δ0.2%	Δ0.1%

Table 4: Largest difference between seeds 0, 123, and 234 for **data24k**

Experiment	Strategy	ID M/C	ID All	OOD M/C ↓	OOD All
all-1	mixed	63.0%	69.8%	63.4%	63.9%
all-1-enh	mixed	68.3%	69.8%	62.6%	63.6%
control	N/A	69.2%	73.0%	62.5%	64.1%
all-2-enh	mixed	61.7%	70.6%	62.5%	63.4%
all-3-enh	mixed	61.3%	70.7%	62.3%	63.4%
all-1-enh	combined	68.6%	68.2%	61.2%	63.4%
all-1	combined	68.3%	71.3%	61.1%	63.6%
all-2	combined	68.3%	71.3%	61.1%	63.6%

Table 5: Comparison for **data1500**

Experiment	Strategy	ID M/C	ID All	OOD M/C ↓	OOD All
control	N/A	72.4%	72.5%	66.4%	68.0%
all-1	mixed	71.7%	72.3%	66.2%	68.0%
all-1-enh	mixed	71.5%	72.7%	66.0%	67.7%
all-2-enh	mixed	71.2%	72.9%	65.8%	67.6%
all-1	combined	71.2%	72.2%	65.0%	67.1%
all-1-enh	combined	76.3%	72.9%	64.8%	67.2%

Table 6: Comparison for **data24k**

Experiment	Strategy	No. Epochs	ID M/C	ID All	OOD M/C ↓	OOD All
control	N/A	3	69.2%	73.0%	62.5%	64.1%
all-1	mixed	10	62.0%	69.6%	62.1%	63.1%

Table 7: Comparison between control and all-1 with 10 epochs for **data1500**

5 Discussion

Taken as a whole, these results suggest that the augmentation strategies we applied did not significantly widen the distribution of the training data. In this section, we first examine why our efforts might have failed. After doing so, we propose a number of recommended directions for future research on this topic.

First, it strikes us as quite interesting that the "mixed" augmentation strategy outperforms the "combined" strategy. This is particularly noteworthy to us because DA traditionally involves keeping both a data point and its augmented version in a dataset, and often serves as a means to artificially increase the amount of training data at one's disposal when gathering more would prove particularly difficult. In addition, this directly contrasts with Dong and colleagues' finding that mixing in augmented data with original data yields better accuracy scores than training the model on the control or augmented data alone (Dong et al., 2023).

We contend that this finding speaks to the insufficient quality of the augmentations we explored. In order to improve upon the `WILDS` approach, it is likely that any applied augmentations would have to make the code snippets even more different from each other than they already are by virtue of being written by different programmers. Otherwise, the DA would fail in its goal of injecting additional variation into the data. In the "combined" approach, by including both versions of a given code snippet, we double the amount of data in our training set that comes from a given repository. That the model underperforms on this data suggests that it may be overfitting to these repositories. In other words, the augmentations do not distinguish snippets from each other enough to make them come from significantly different distributions. That the "combined" data does not perform worse on ID accuracy, only OOD, further supports this.

While the "mixed" strategy does not perform worse than the control, it also does not do better. This is in line with the above conclusion in two princi-

pal respects. First, the performance gap disappears once we stop doubling the amount of code coming from the same repository. Second, the "mixed" data failing to outperform the control suggests that, indeed, the applied augmentations are not creating any more variation in the data than what already existed by virtue of the `WILDS` approach. That applying multiple augmentations to a single snippet still fails to produce meaningful results further suggests that the augmentations themselves are not good enough to serve their intended purpose.

With this in mind, it is worth exploring why the augmentations may have fallen short of their goal, especially given the promising results reported by Dong and his colleagues. One possible explanation is that the researchers only test their results on ID data. They use `Java250` and `Python800`, both robust datasets having 63000 and 201600 different code snippets respectively (Dong et al., 2023). While it is highly likely such large datasets span a variety of distributions, the authors make no note of tracking this information when they divide the snippets into test and train categories. Thus, it is possible that the train and test distributions are quite similar, and that observed improvements are not generalizable to OOD data.

The authors appear to account for this by separately offering a "robustness" measure. Dong and colleagues report a 26.06% gain in the model's ability to generalize to unseen data, which suggests that the augmentations would indeed be well-suited to enhancing OOD performance. In investigating this, we more closely examined how this metric was calculated. To obtain the robustness score, the researchers diversified the test data by applying the very same augmentations they applied to the train data. In other words, the robustness improvement of 26.06% that they report is only a measure of how well a model trained on data mutated by their augmentations will be able to predict other data mutated by these same augmentations. This casts doubt on whether the model is truly generalizable to unseen code one might find in the real world, or if it is only generalizable to new code stemming from the refactoring methods they themselves conceived. If the augmentations are not actually reflective of real-world differences between programmers, the metric may not actually indicate the model's generalizability at all. Our findings indicate that this may indeed be the case.

We hypothesize that the lack of naturalness in many of the applied augmentations may be a key factor underlying their minimal impact. For example, the `rename_api` augmentation does not rename imported APIs to other, existing APIs; instead, it simply renames a given API to a randomly generated synonym of itself and does not alter names of any functions that come from it. While it is true that Python programmers often use different APIs to achieve a given goal, these various APIs have specific names and come with their own distinct functions. In other words, this augmentation mutates code snippets into code that one would never actually find in a real Python program.

The same can be said for many of the other augmentations, including our own. For example, it is not very common to find real-world code containing dead branches. In addition, `add_arguments` increases the number of parameters in a function declaration string but then does not actually use any of the additional

arguments in the function body. This, too, is not a very natural representation of how functions are actually written. And while `insert_random_class` and `insert_random_function` attempted to apply more sophisticated refactoring, generated classes and functions still contained much more randomness in choice of names, operations, and how function bodies *relate* to their names than they tend to in the real world. As per the conclusions of Yang et al. (2022) and Yu et al. (2022), this lack of naturalness may be responsible for the augmentations' failure to help the code generalize.

Based on these observations, we offer two recommendations for future research in this field. The first is a call for standardized measurements of model robustness across code generation research. Proposals of such a standard have not begun until quite recently, and the discrepancy in findings between our work and that of Dong and colleagues underscores their importance. Existing work has proposed testing robustness by using DA to manipulate train data ane examining how well the model performs on the altered set (Dong et al., 2023; Wang et al., 2023). The present investigation indicates that this approach may fall short when applied augmentations are not natural or sophisticated enough. The approach put forth by `WILDS` - that is, evaluating how well the model predicts on snippets written by programmers whose code it was not trained on - may be an effective alternative. While it is much more costly to collect such data than artificially altering data already in one's possession, this approach skirts the naturalness issue faced by Dong et al., since the `WILDS` approach utilizes code written by real programmers.

Along these lines, we also suggest a formal and detailed investigation into the ways different programmers' styles differ so that more advanced augmentations can be devised moving forward. Our results suggest that simple data augmentation strategies are not effective in helping models generalize to OOD data, particularly when the distribution of training data is already quite wide. If we are to create augmentations that truly broaden a training distribution, we posit that such augmentations must transform a snippet into a form that is significantly different from its original yet also still resembles code that a human would actually write. After all, if we are to improve models' ability to predict on code "in the wild", that is precisely the type of code that data augmentation should generate.

6 Conclusion

We combined two standard approaches to the distribution-shift problem to investigate whether existing data augmentation approaches in code generation could be used to augment the scores obtained by the `WILDS` project. Based on the results of our various experiments, we conclude that presently available code augmentations cannot sufficiently enhance already robust training sets. Employing data augmentation to supplement the inclusion of additional code sources is a worthwhile enterprise, as accomplishing this would greatly diminish time costs of actively seeking natural data from new sources. However, an

augmentation effective at introducing variation where so much data diversity already exists would likely need to be highly advanced. Further study is required to determine precisely what such an augmentation might look like.

While there has been much application of data augmentation in vision and natural language tasks, the applications of data augmentation in code generation are only beginning to be explored. We hope that this work will contribute to this nascent conversation and help guide endeavors to train top-quality code completion models as the field progresses.

7 About the authors

Melanie Becker is a second-year M.S. student in computer science at Northeastern University's Khoury College of Computer Science. She is interested in using the practical applications of machine learning to improve the lives of others. Contact her at becker.mel@northeastern.edu

Luka Jovanović is a second-year M.S. student in computer science at the HPC laboratory in Northeastern University in Boston, USA. He is interested in machine learning and machine learning theory.
Contact him at luca.jovanovich@gmail.com
ORCID iD

References

[1] Chaudhuri K., Monteleoni C., Sarwate A. (2011). Differentially private empirical risk minimization. Journal of Machine Learning Research, 12, 1069–109.

[2] Dong Z., Hu Q., Guo Y., Cordy M., Papadakis M., Zhang Z., et al. (2023). "Mixcode: Enhancing code classification by mixup-based data augmentation". 30th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).

[3] Pellicer L. F. A. O., Ferreira T. M., Costa A. H. R. (2023). Data augmentation techniques in natural language processing. Applied Soft Computing, 132, 109803. https://doi.org/10.1016/j.asoc.2022.109803.

[4] Koh P. W., Sagawa S., Marklund H., Xie S. M., Zhang M., Balsubramani A., Hu W., Yasunaga M., Phillips R. L., Gao I., et al. (2021). "Wilds: A benchmark of in-the-wild distribution shifts". International Conference on Machine Learning (ICML), 5637–5664.

[5] Shorten C., Khoshgoftaar T. M. (2019). A survey on image data augmentation for deep learning. Journal of Big Data, 6(1), 60.

[6] Raychev V., Bielik P., Vechev M. (2016). Probabilistic model for code with decision trees. ACM SIGPLAN Notices.

[7] Wang S., Li Z., Qian H., Yang C., Wang Z., Shang M., Kumar V., Tan S., Ray B., Bhatia P., Nallapati R., Ramanathan M. K., Roth D., Xiang B. (2023). ReCode: Robustness evaluation of code generation models. Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 13818–13843. https://aclanthology.org/2023.acl-long.773, doi:10.18653/v1/2023.acl-long.773.

[8] Yang Z., Shi J., He J., Lo D. (2022). Natural Attack for PreTrained Models of Code. Proceedings of the 44th International Conference on Software Engineering (ICSE), 1482–1493.

[9] Yu S., Wang T., Wang J. (2022). Data Augmentation by Program Transformation. Journal of Systems and Software, 190, 111304. https://doi.org/10.1016/j.jss.2022.111304.