

Memorization and Generalization in Neural Code Intelligence Models

Md Rafiqul Islam Rabin
 mrabin@uh.edu
 University of Houston
 Houston, TX, USA

Aftab Hussain
 ahussain27@uh.edu
 University of Houston
 Houston, TX, USA

Vincent J. Hellendoorn
 vhellendoorn@cmu.edu
 Carnegie Mellon University
 Pittsburgh, PA, USA

Mohammad Amin Alipour
 maalipou@central.uh.edu
 University of Houston
 Houston, TX, USA

Abstract—Deep Neural Networks (DNN) are increasingly commonly used in software engineering and code intelligence tasks. These are powerful tools that are capable of learning highly generalizable patterns from large datasets through millions of parameters. At the same time, training DNNs means walking a knife’s edges, because their large capacity also renders them prone to *memorizing* data points. While traditionally thought of as an aspect of over-training, recent work suggests that the memorization risk manifests especially strongly when the training datasets are noisy and memorization is the only recourse. Unfortunately, most code intelligence tasks rely on rather noise-prone and repetitive data sources, such as GitHub, which, due to their sheer size, cannot be manually inspected and evaluated. We evaluate the memorization and generalization tendencies in neural code intelligence models through a case study across several benchmarks and model families by leveraging established approaches from other fields that use DNNs, such as introducing targeted noise into the training dataset. In addition to reinforcing prior general findings about the extent of memorization in DNNs, our results shed light on the impact of noisy dataset in training.

Index Terms—memorization, generalization, models of code

I. INTRODUCTION

Data-driven software engineering and program analysis approaches have been increasingly used in tasks such as type prediction [1], method name prediction [2], [3] and many more [4]–[6]. These approaches rely on large corpus of available code repositories and use machine learning techniques to extract useful patterns and insights about programs. In past few years, an increasing number of such approaches use neural networks at their core for extracting these patterns. Deep neural networks are powerful machine learning techniques that can express any hypothesis class through massive parameters and hyper-parameters; they are universal approximators [7]. Studies have shown that the neural networks can use these parameters and *memorize* arbitrary data points through their massive parameters.

Memorization can be a double-edge sword: while it can help to pick up sparse data points, it can also lead to memorizing noise. It is specially important for the code intelligence application where the data is usually extracted from noise-prone sources such as GitHub. The sheer size of such data sources makes inspection and cleaning of the dataset nearly impossible. Allamanis *et al.* [8] showed that almost all datasets used to train neural code intelligence models contain levels of code duplication of 20% or more, which spuriously inflates

the reported models performance. The key problem with such duplication is that it encourages the model to memorize, which recent evidence suggests is directly adverse to the ability of neural models to generalize [9].

In this work, we study memorization and generalization in training neural code intelligence models through a case study by following the study portrayed by Arpit *et al.* [9] and Zhang *et al.* [10]. To this end, we induce noise to the datasets by randomly altering the training labels and observing its impact on different characteristics of training, e.g. loss values. By studying the resulting trends, we can gain insight into the impact of memorization artifacts in various models uniquely proposed in our field, as well as memorization artifacts already present in currently popular datasets. We experiment with this approach by a case study on the method name prediction task [2], [3] – a popular task commonly used in the evaluation of code intelligence models [11]–[18]. We use two network architectures, CODE2VEC [19] and CODE2SEQ [20], and two datasets. To the best of our knowledge, it is the first case study to evaluate memorization and generalization phenomena in the neural code intelligence models.

We study multiple datasets including manually-crafted clean dataset and dataset collected from GitHub. We add noise in datasets by randomly changing a percentage of labels in the training data, and then train the model on each noisy dataset. By studying the resulting models in terms of both training and (uncorrupted) testing metrics, including prediction score, the spread of loss, F1-score, and critical sample ratio at various noise levels and across datasets and models, we derive a range of findings related to these models and datasets. For instance, in the manually-crafted clean dataset, there is a significantly less difference between scores among different noise levels than in the potentially noise-prone GitHub dataset, signalling that the latter is substantially more noise and ambiguous, to begin with. A similar observation is found for the distribution of loss, model’s F1-score, and critical adversarial sample ratio.

In our case study, all models manifested varying degrees of memorization; models with higher memorization create more complex hypothesis classes; heavily depends on the architecture of models, and the impact of noise in the models based on a manually-curated dataset was more pronounced than the models based on GitHub dataset. Our analysis and findings set the stage for using more advanced methods and metrics than the conventional performance-related ones to better understand

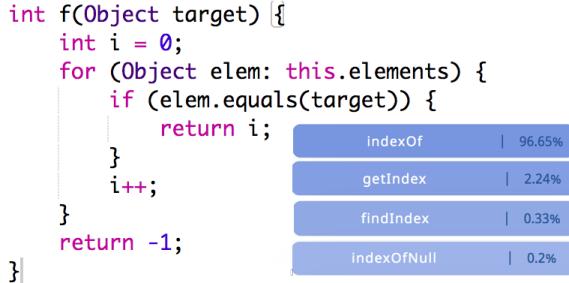


Fig. 1: An example of method name prediction task [19].

neural code intelligence models.

Contributions. This paper makes the following contributions.

- We conduct a case study on the evaluation of memorization and generalization in neural code intelligence models.
- We compare the memorization and generalization in models based on noise-prone and manually-crafted datasets.
- We discuss the potential ramifications of the findings.

II. CASE STUDY

We study memorization in the code intelligence models that use neural networks through a case study; in particular, we use the method name prediction [2], [3] task for this study.

A. Subject Task: METHODNAME

We use the method name prediction task [2], [3], a popular code intelligence task that has gained interest recently [21]. In the METHODNAME task, the model attempts to predict the name of a method, given its body. This task has several applications such as code search [22], code summarization [3], and reasoning about code analogies [19]. Figure 1 depicts an example of this task wherein a model is given a method body and returns candidate names for the method body with an associated probability score; e.g., `indexOf` (96.65%), `getIndex` (2.24%), `findIndex` (0.33%), and so on. This task has been used as the downstream task to evaluate several state-of-the-art neural code intelligence models [19], [20].

B. Models

We study two commonly used neural models for the above prediction task: CODE2VEC [19], and CODE2SEQ [20]. The models are similar, in that they rely on extracting “paths” from the method’s abstract syntax tree (AST) that connect one terminal or token to another. These paths, mapped to vector embeddings, are enumerated exhaustively and used by the models in different ways. Since these paths consolidate both lexical and syntactic information, these models give us more benefit than strictly token-only models.

In CODE2VEC [19] training, each path, along with its source and destination terminals, is mapped into a vector embedding, which is learned together with other network parameters. Then, the separate vectors obtained from each path-context are concatenated into a single context vector using a fully-connected layer. Additionally, the model learns

an attention vector that is used to aggregate the path-context representations into a single code vector that represents a method body. Finally, given a method body’s code vector, the model predicts the probability of each target method name using a softmax-normalization between the code vector and each of the embeddings of target method names.

In CODE2SEQ [20], an encoder-decoder architecture is used to encode paths node-by-node and generate labels as sequences at each step. Here, the encoder represents a method body as a set of paths in AST, among which individual paths are compressed to a fixed-length vector using a bi-directional LSTM. Thereby, the encoder encodes paths node-by-node while splitting tokens into sub-tokens. The decoder uses attention to select relevant paths while decoding, and predicts sub-tokens of a target sequence at each step while generating a method name.

C. Datasets

We perform experiments on the following two Java datasets based on the GitHub projects: SORTING-ALGORITHM and JAVA-SMALL.

- **SORTING-ALGORITHM (SA):** This dataset [23] contains 1000 sorting algorithms from 10 different labels written in Java. It has been manually crafted from GitHub and labeled into 10 classes, those are: bubble, bucket, heap, insertion, merge, quick, radix, selection, shell, and topological.
- **JAVA-SMALL (JS):** This dataset [20] contains nine Java projects for training, one Java project for validation, and one Java project for testing. In total, it contains about 700K methods.

III. METHODOLOGY

This section describes the methodology to study memorization in code intelligence models.

We follow and adapt the methodology used in Arpit et al. [9] and Zhang et al. [10]. The underlying insight is simple: if the model still learns well on the noisy training set, it is memorizing data points; comparing learning in the original dataset and noisy dataset can help characterize memorization artifacts in the model for a given dataset. This methodology is general and can be applied to any learning model, however, memorization is more likely in neural networks.

Figure 2 depicts a high-level view of the workflow in the proposed methodology. Given the original training dataset, the approach creates several noisy training datasets by replacing labels in a portion of the training dataset with another randomly selected label. We create multiple training datasets with {0%, 25%, 50%, 75%, 100%}-noise, where 0%-noise denotes the original training dataset, and 100%-noise denotes a dataset where all labels in training set are fully replaced with different labels. We use each noisy training set to train a model while using the original validation set. Finally, we evaluate the training characteristics of each model on the original test dataset.

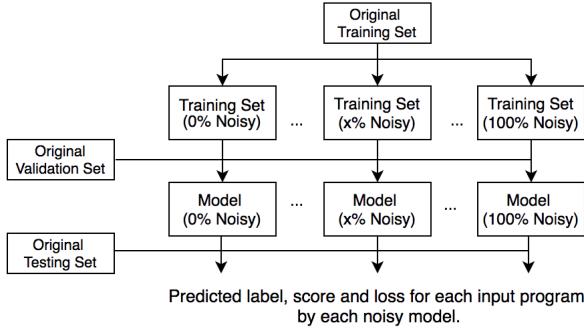


Fig. 2: Workflow of the approach.

Following Arpit et al. [9], we collect the following metrics to characterize training in the datasets. We describe those metrics in the rest of this section.

A. Predicted Score

Predicted score refers to the probability score assigned to the predicted output by the model. Depending on the model, we compute it differently. In CODE2VEC, the model computes the probability of the target name via a softmax-normalization between the code vector of a given method body and the embeddings of all possible method names; we, therefore, use the following formula to compute the predicted score.

$$P(name_i) = \frac{\exp(\text{code_vector}^T \cdot \text{name_embedding}_i)}{\sum_{name_j \in \text{all_name}} \exp(\text{code_vector}^T \cdot \text{name_embedding}_j)}$$

In CODE2SEQ, when predicting the method name, the model makes predictions for each sub-tokens of a target sequence at each step, hence, we compute an average score for a single program as follows.

$$P_{avg}(name_i) = \frac{\sum_{token_j \in name_i} P(token_j)}{|\{token_j \in name_i\}|}$$

B. Precision, Recall, and F1-Score

We use the traditional evaluation metrics, F1-score over sub-tokens, as commonly used in the literature for the method name prediction task [19], [20]. 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* denotes 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.

$$\text{Precision}(P) = \frac{tp}{tp + fp} * 100$$

- *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.

$$\text{Recall}(R) = \frac{tp}{tp + fn} * 100$$

- *F1-Score* is the harmonic mean of precision and recall.

$$F_1\text{-Score} = \frac{2}{P^{-1} + R^{-1}} = 2 \cdot \frac{P \cdot R}{P + R}$$

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

C. Spread of Loss

The loss refers to the error in the model's prediction, specifically in terms of the probability assigned to the ground-truth label. In CODE2VEC, the model computes cross-entropy loss between the softmax of raw logits (predicted distribution q) and the ground-truth targets (true distribution p). The true distribution p assigns a value of 1 to the actual name and 0 otherwise, therefore, the cross-entropy loss for a single input method is equivalent to the negative log-likelihood of the actual name.

$$L(name_i) = - \sum_{name_j \in \text{all_name}} p(name_j) \log q(name_j) \\ = -\log q(name_i)$$

In CODE2SEQ, the model makes predictions for each sub-token of a target sequence at each step; hence, we compute an average loss for a single input method as following.

$$L_{avg}(name_i) = \frac{\sum_{token_j \in name_i} L(token_j)}{|\{token_j \in name_i\}|}$$

Gini-coefficient. We measure the spread of loss by computing the Gini-coefficient [24] over training loss after each epoch as training progress. A Gini-coefficient of 0 means perfect equality where all values are the same. On the other end of the spectrum, a Gini-coefficient of 1 means maximal inequality among values. The Gini-coefficient is computed as the relative mean absolute difference of all pairs of items in the population. If L_i is the loss of test input program t_i , and there are n test input programs, then the Gini-coefficient (G) is computed as follows:

$$G(\text{loss}) = \frac{\sum_{i=1}^n \sum_{j=1}^n |L_i - L_j|}{2n \sum_{i=1}^n L_i}$$

D. Critical Sample Ratio

We also estimate the complexity of decision boundaries by computing the critical sample ratio (CSR) [9]. An input program is called a critical sample if there exists at least one

adversarial sample program¹ in close proximity (δ) of the input program. For a given test set D_t , the critical sample ratio (CSR) is measured as following:

$$CSR_{\delta} = \frac{\# \text{critical_samples}_{\delta}}{|D_t|}$$

A higher value of CSR (closer to 1) indicates a complex decision boundary, where many samples are just a small transformation away from being labeled differently, whereas a lower value of CSR (closer to 0) indicates a simpler, more robust decision boundary.

We explore programs within single-transformation distance ($\delta = 1$) of a given input program for adversarial programs following [14], [25]. We check for an adversarial program within the single transformation distance of a given input program t_i to identify whether t_i is a critical sample. Specifically, we apply the single-place variable renaming transformation [14] on the input program t_i to generate candidate programs. The transformation changes the name of a single variable in the input program to a new name following the predefined format $var[0 - 9]+$ (e.g. “var3”). The transformation is performed one-by-one on each variable in the input program, creating a set of candidate programs within the single transformation distance. Suppose, T_{C_i} is a set of candidate programs generated within the single transformation distance of t_i . Then, t_i is a critical sample if there exist at least one candidate program $t_j \in T_{C_i}$ such that $M(t_j) \neq M(t_i)$, where $M(t)$ indicates the predicted name of the program t by model M .

IV. EXPERIMENTATION SETTING

In this section, we briefly describe the experimentation setting.

A. Sample Types

A model prediction on test input data can be correct or incorrect. To have a more focused study on the impact of memorization in the prediction, we considered two samples in the test set: *all samples* and *correct-only samples*. In *all samples*, the entire set of samples in the dataset regardless of the model’s prediction, while in *correct-only samples*, the set of samples for which model correctly predicts at least one sub-token. Each sample in our datasets constitute a method body as input and a method name (or label in case of SORTING-ALGORITHM dataset) as output. If not mentioned explicitly, we use *all samples* from the test set in figures and results.

B. Label Matching

Each sample in our dataset constitutes a method body as input and a method name (in case of JAVA-SMALL dataset) or label (in case of SORTING-ALGORITHM dataset) as output. Method names may comprise several tokens as developers usually use

¹In machine learning, an adversarial example of an input in a model is a sample with a slight, ideally imperceptible and/or irrelevant difference to the original input, that misleads the model into providing a different prediction. In the method name prediction task, this can be a semantically-equivalent and largely syntactically identical sample program that leads to a different predicted method name.

camelCase (or even snake_case) to express those names. In the METHODNAME models, we consider two types of matches of the predicted output: *sub-token match* and *exact-match*. In exact matching, the prediction is considered correct only if the prediction exactly matches the label, while in the sub-token match, we consider a prediction correct if at least one of its sub-tokens matches with the sub-tokens in the actual name. Note that sub-token matches are commonly used for experiments with JAVA-SMALL dataset. In the SORTING-ALGORITHM samples the labels are only a single token, therefore being matched exactly.

C. Hardware

We used a server with an Intel(R) Xeon(R) 2.30GHz CPU and a single NVIDIA Tesla P100 GPU with 12GB of memory to run the experiments in this study.

V. RESULTS

In this section, we present the results of our experiments where we seek to answer the following research questions:

- RQ1 What are the effects of dataset noise on the model’s performance?
- RQ2 What are the effects of dataset noise on the learned decision surface?
- RQ3 What are the effects of dataset noise on the predicted score?
- RQ4 What are the effects of dataset noise on the spread of loss?

A. RQ1: Effects of dataset noise on model’s performance

In order to assess performance of the METHODNAME models, we focused on the models’ F1-scores, which are commonly used at the sub-token level as a performance indicator of the models [19], [20]. Specifically, we tracked this score while training, and evaluated it on the test data at the end of each epoch (a pass through the full training data) on the corresponding test datasets (SORTING-ALGORITHM and JAVA-SMALL). Figure 3 shows the resulting changes in F1-score on the training set (solid line) and test set (dashed line) up to 50 epochs at different noise levels. The results show that the training accuracy with both clean and noisy data converges to the same point for CODE2VEC, after a varying number of epochs (slightly more for noisier data). CODE2SEQ, meanwhile, struggles more with memorization and is unable to fit the noisy training data perfectly within 50 epochs.

As expected, the models trained on the original data demonstrate much higher test accuracy than those trained on noisy data: the latter *generalize* less well. This effect is consistent between CODE2VEC and CODE2SEQ, though the underlying curves are quite distinct. We also note that the F1-Score on the test sets are generally lower than what is achieved by their counterpart training sets at the same noise level – a trend that holds with both models and datasets, except for 25-75% noise in Figure 3c and 75% noise in Figure 3d. This gap is particularly prominent in the CODE2VEC model (Fig. 3a and 3b), clearly indicating the increased role of over-fitting

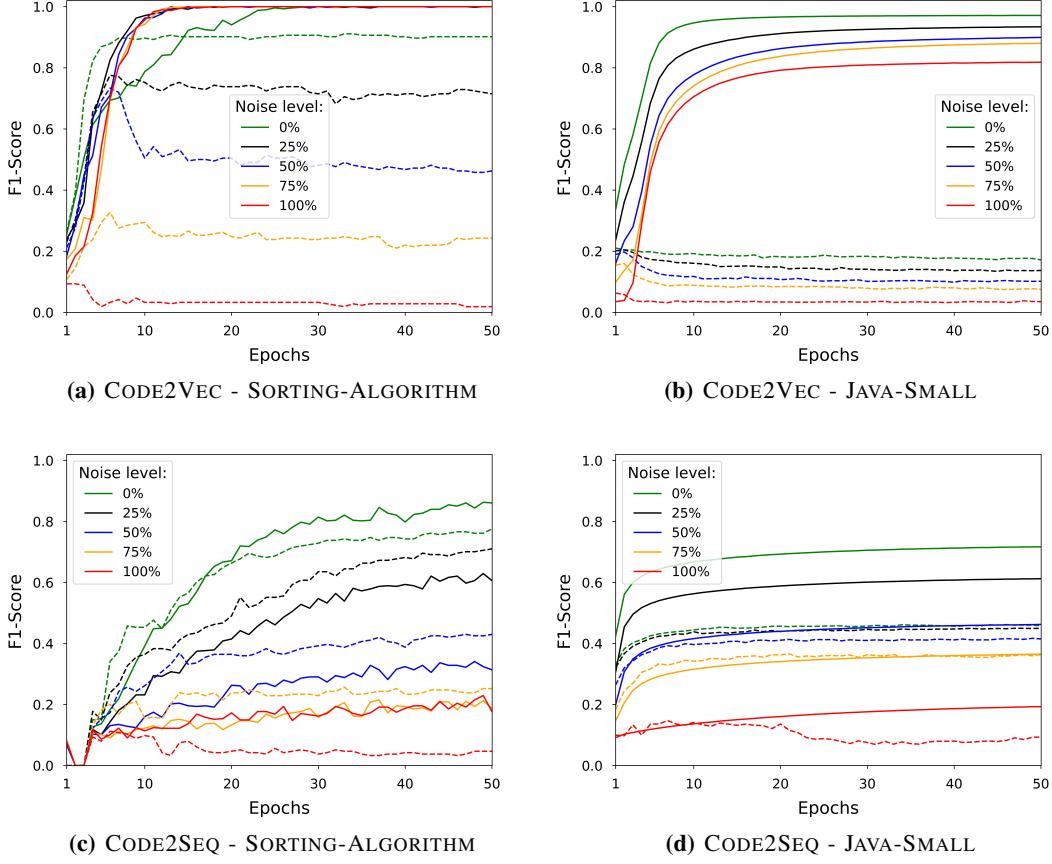


Fig. 3: F1-Score of training set and test set at different noise levels (solid is training, dashed is test).

(memorization) during the training process as more noise is present.

Comparing across datasets is informative as well: high noise levels lead to test accuracy quickly saturating on **SORTING-ALGORITHM**, and even dropping somewhat (e.g. the 50% noise level of **CODE2VEC**), whereas the 0% and 25% models continue learning meaningful patterns all the way through, even converging to roughly the same test accuracy under **CODE2SEQ**. This discrepancy in learning duration is absent on **JAVA-SMALL**, although there too, the 0% and 25% test curves are roughly even. That this trend is less present in **JAVA-SMALL** may be explained by its larger dataset size, and consequentially longer epochs; yet, even so, the gaps between the generalization quality of the various noise levels under **CODE2SEQ** is unexpected: we would expect these to be roughly spaced evenly, as on **SORTING-ALGORITHM**, but with **JAVA-SMALL** even a 75% noise level still yields surprisingly good test performance. Only at 100% noise does the generalization drop steeply, to a random base rate. That learning with 75% noise compromises the model by only a small margin may well echo a significant degree of randomness inherent in the latter dataset – a theme echoed in later results.

Observation 1: The training accuracy with both clean and noisy data converges towards the same point in **CODE2VEC**, however, **CODE2SEQ** struggles more with memorization and is unable to fit the noisy training data perfectly.

B. RQ2: Effects of dataset noise on the learned decision surface

In order to understand how noise affects the complexity of the hypotheses learned by these CI models, we examined the number of critical samples in the test set after each epoch throughout training. Figures 4 and 5 show the critical sample ratio (CSR) obtained for all samples (both correct and incorrect prediction) and for correct samples only, respectively, up to 50 epochs at different noise levels. According to Arpit et al. [9], obtaining a higher number of critical samples for models trained on noisy data than that obtained for models trained on the original data indicates a more complex learned decision surface for noisy data. In general, Arpit et al. [9] got higher CSR with more noisy datasets with their models. In our experiments, this trend was most prominently reflected with **CODE2VEC** for both **SORTING-ALGORITHM** and **JAVA-SMALL** datasets. This shows that noise tends to increase the complexity of the learned decision surface of **CODE2VEC**. In **CODE2SEQ**, this trend was more prominent with **SORTING-ALGORITHM** dataset

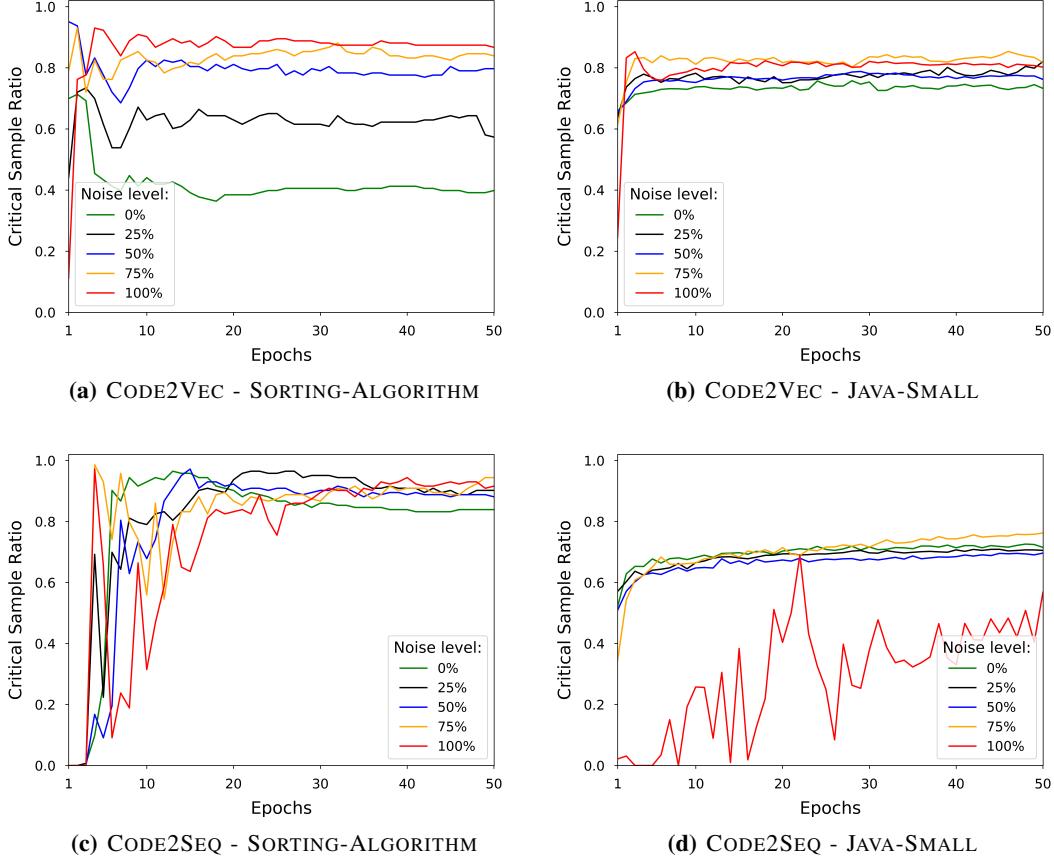


Fig. 4: Critical Sample Ratio (CSR) for all examples.

after 30 epochs, but with JAVA-SMALL we did not observe any such relation between the amount of noise and CSR. For example, in Figure 4d and 5d, we see the model trained with the unchanged dataset (0%-noise) yields higher CSR than it did when trained with 25%- , 50%- , and even 100%-noise datasets.

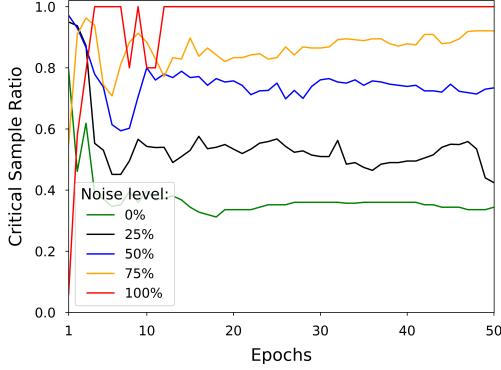
We also observe CSR values reaching almost stability within 50 epochs for both the models, for almost all versions of the datasets, the exceptions being the trends with the 100%-noise JAVA-SMALL dataset on CODE2SEQ model. In addition, we see the CSR values initially change for each dataset with time in a very erratic manner (e.g. see Fig. 4c, 5c), before settling into a more stable, expected pattern towards the end of training. This starkly contrasts with Arpit et al. [9]'s results on their datasets, where the CSR increases gradually with increasing number of epochs and then reaches stability. Elaborating on their observation, they suggested that a gradual increase in CSR during model training shows a gradual increase in learning more complex hypotheses [9]. Compared to that observation, the strong fluctuations of CSR values early in training and relatively minor differences upon convergence are quite unexpected. Additionally, between the two models, CODE2VEC exhibits much more pronounced, reliable CSR differences than CODE2SEQ in varying noise levels, which consistently suffers from a very high CSR, likely making it

more susceptible to adversarial perturbations [17]. Our work is the first to use this metric to quantify training robustness; the unusual and strongly divergent patterns found across models and datasets suggest that further analyses with such methods is necessary in our field.

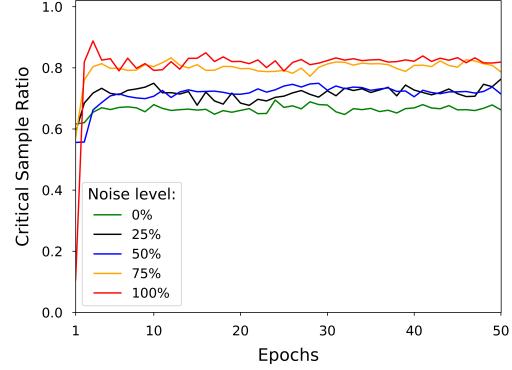
Observation 2: CODE2VEC mostly exhibits significant CSR differences than CODE2SEQ in varying noise levels, which consistently suffers from a very high CSR, likely making it more susceptible to adversarial perturbations.

C. RQ3: Effects of dataset noise on the predicted score

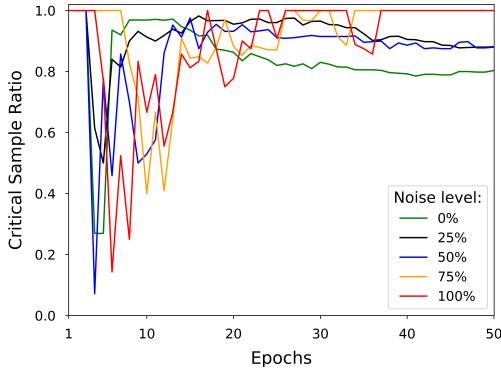
Some instances fit highly predictive patterns better than others. As a result, easily predicted examples will have (much) higher associated probability scores than harder examples. This difference in scores should be quite pronounced when comparing original data (some instances fit patterns) with noisy data (fits independently). In order to observe these behaviors in CI models, we obtained the probability assigned to the predicted method name for all examples in the test set after each epoch. Figure 6 shows the sorted probability of predicted name for all examples after the best epoch of training, while Figure 7 shows the sorted average probability of the predicted name for all examples considering all epochs of training.



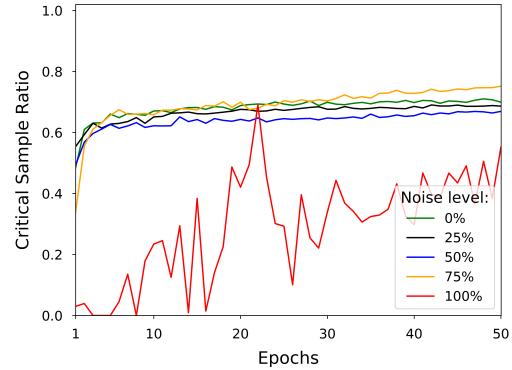
(a) CODE2VEC - SORTING-ALGORITHM



(b) CODE2VEC - JAVA-SMALL



(c) CODE2SEQ - SORTING-ALGORITHM



(d) CODE2SEQ - JAVA-SMALL

Fig. 5: Critical Sample Ratio (CSR) for correct examples.

From these figures, we can see that prediction scores of CODE2VEC and CODE2SEQ steadily get worse with the increase of noise in dataset (except for the 100% noisy JAVA-SMALL with CODE2VEC). This trend of probability among examples at different noise levels is especially prevalent in the SORTING-ALGORITHM dataset, whereas it is not as consistently observed in the JAVA-SMALL dataset as in the SORTING-ALGORITHM dataset. For example, the CODE2VEC model on the JAVA-SMALL dataset shows higher prediction scores with 100% noise than with 25%--, 50%- and 75%-noise in Figure 6b. When taking all epochs into consideration, this effect of 100% noise on prediction scores appears on par with 75% noise in Figure 7c. We conjecture that this due to the inferior quality of JAVA-SMALL dataset, where labels often correspond poorly, or ambiguously, to the code samples even in the original dataset, as opposed to the manually-crafted SORTING-ALGORITHM dataset. This metric may thus be useful to classify such ambiguity inherent in datasets in our field.

Observation 3: Prediction scores of CODE2VEC and CODE2SEQ steadily get worse with the increase of noise, that appears prevalent in the manually-crafted SORTING-ALGORITHM dataset but not consistently observed in the JAVA-SMALL dataset where true labels often correspond poorly, or ambiguously.

D. RQ4: Effects of dataset noise on the spread of loss

In prior work [9], only a subset of the training was found to have a high loss in original data, while the loss is high for virtually all examples in random data. Therefore, the spread of loss (computed with the Gini-coefficient) should be significantly higher in the original data than in random data. In order to observe the spread of loss in CI models as training progresses, we obtained the loss of each sample in the training set after each epoch without modifying the training procedure.

Figure 8 shows the result. For both models and datasets, we can see that the Gini-coefficient value decreases significantly with an increasing noise level. Note that a coefficient of 0 means all values are the same, while a coefficient of 1 means maximal inequality among values. Therefore, the diversity of observed losses is quite high in the original data, as the coefficient is closer to 1. Moreover, the difference among coefficient values

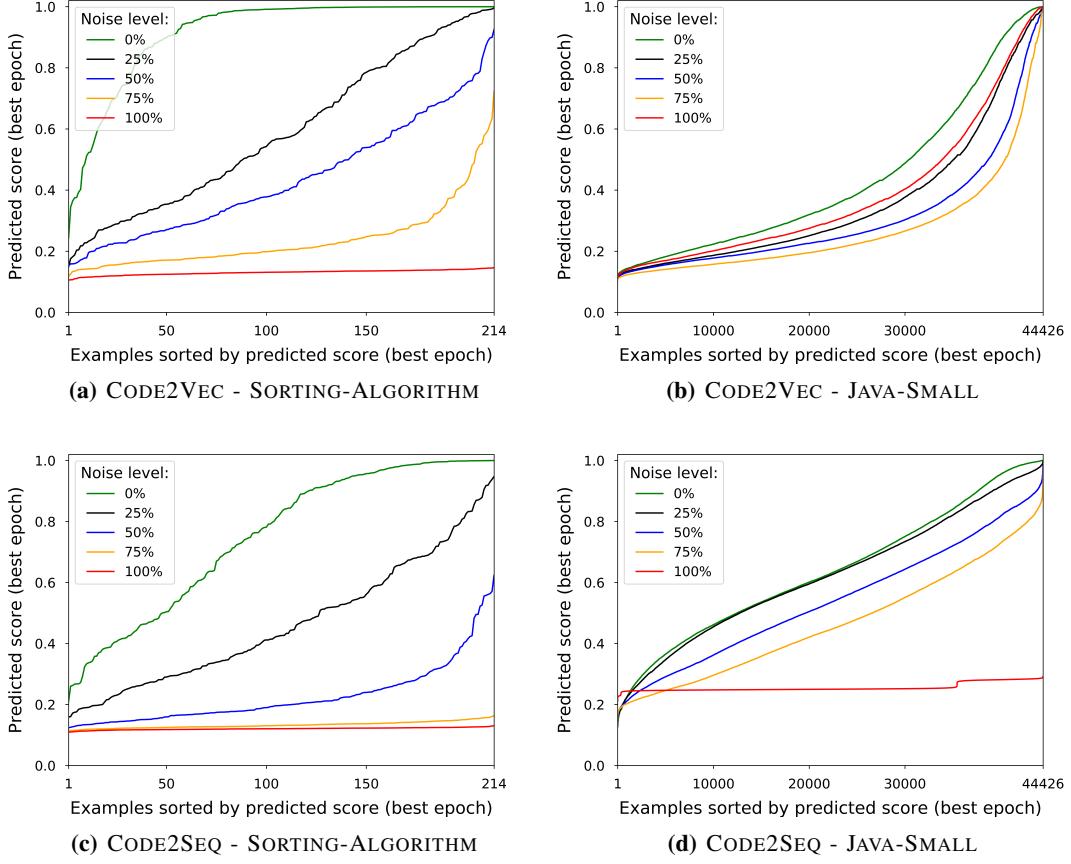


Fig. 6: Probability of predicted name for all examples after the best epoch of training.

at different noise levels is significantly higher in the SORTING-ALGORITHM dataset than the JAVA-SMALL dataset, for both CODE2VEC and CODE2SEQ, which indicates a greater spread in losses in the former.

Observation 4: The spread of loss at different noise levels is significantly higher in the SORTING-ALGORITHM dataset than the JAVA-SMALL dataset, for both CODE2VEC and CODE2SEQ.

VI. DISCUSSION AND FUTURE WORK

In this section we discuss the results and its implication for training neural code intelligence models.

A. Memorization and Network Architecture

Our results suggest that all models are susceptible to memorization, and network architecture can influence the memorization behavior of the models. The results suggest that sequence-based models, i.e. CODE2SEQ, exhibit less memorization than CODE2VEC in our study. This observation is consistent with Maharaj et al.'s [26] observation in the field of vision that recurrent networks memorize less compared to multi-layer feed-forward neural networks. Neural code intelligence models use a variety of network architectures, e.g. graph neural networks [27] and convolutional neural networks [28]. Our

results highlight the need to further evaluate and compare memorization in such models and architectures.

B. Synthetic noise and Dataset quality

In our experiment, inducing noise has had a more pronounced impact on the F1-score of the clean dataset, i.e. SORTING-ALGORITHM, as the difference between training and testing accuracy increases with more noise. This observation requires further evaluation in a larger-scale study. If true, perhaps one can create a framework to compare the quality of various datasets by synthetically adding noise to the dataset and compare the training behavior of the models.

C. Study of Training in Code Intelligence Models

Neural networks are powerful tools for learning from arbitrarily large datasets; however, with that, they pose two main challenges: (1) what do they learn? and (2) how do they learn? Ideally, for the users, as long as neural networks learn correct patterns, the latter question is moot, or of lesser importance. However, datasets are noisy in reality. In such cases, answers to the latter question can provide insights about what is actually learned by the models. Moreover, a model that learns mostly based on memorization cannot generalize well to new unseen data and is also susceptible to adversarial examples. Noise

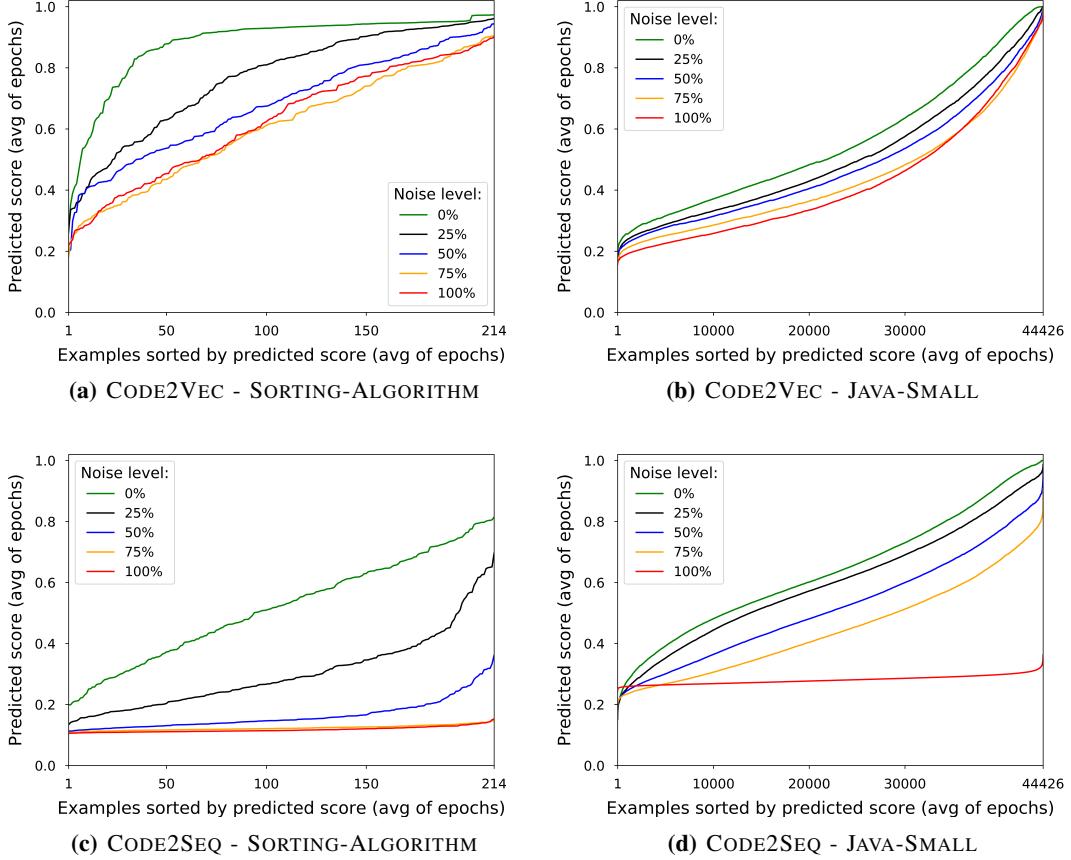


Fig. 7: Average probability of predicted name for all examples considering all epochs of training.

can skew the results and spuriously inflate the performance of code intelligence models. GitHub is a major data source used in training code intelligence models, and unfortunately, it can be noisy [29]. For effective and sound use of neural models in code intelligence applications, our community needs to develop rigorous frameworks for the evaluation and adoption of such models. We hope this work would enable researchers in developing such frameworks.

VII. THREATS TO VALIDITY

In this section, we discuss some of the limitations of our experiments. Our work is the first to investigate a series of metrics on code intelligence tools. In terms of implementation, we were able to rely on the public implementation of the underlying models (CODE2VEC and CODE2SEQ) and the description of the metrics by Arpit et al. [9]. As such, the primary threat to our work’s validity is external: our results are based on the evaluation of memorization and generalization on a case-study of a single type of task; while we analyzed these effects across several models and datasets, they may not generalize to entirely different kinds of code intelligence tasks. However, our results are consistent with similar studies in other domains, e.g. vision [9]; studying these effects in other settings

will likely be equally informative and is a worthwhile direction for future work.

VIII. RELATED WORK

In this section, we elaborate upon work that explores the behavior of Deep Neural Networks (DNNs) when trained with clean and unclean data. We intersperse our discussion by highlighting opportunities of building an infrastructure for inferring the state of the data with which a DNN was trained, based on the DNN’s behaviour while being trained on that data.

Zhang et al. [10] show that random data can be fit perfectly with DNNs (specifically, convolutional neural networks for image classification trained with stochastic gradient descent methods). This implies that DNNs have the potential to employ a high degree of memorization. Moreover, they observe that a NN with sufficient *effective capacity*, a notion that they propose, is capable of memorizing the training data, regardless of how noisy it is, without needing any significant additional training. For example, for an Alexnet-style CNN, they show that the difference between the relative convergence times to fit training data without label corruption and to do the same with data with the maximum label corruption is quite small. This observation was corroborated by similar experiments by Arpit et

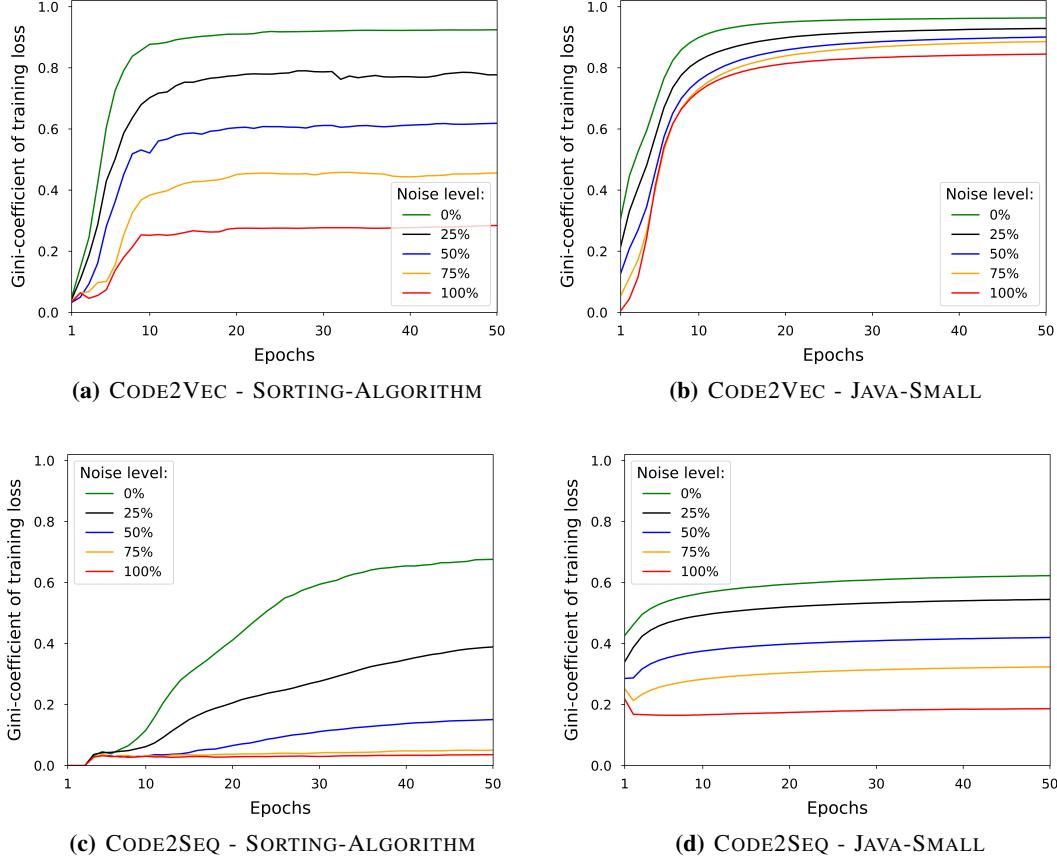


Fig. 8: Spread of loss as training progress after each epoch.

al. [9] on the CIFAR10 and MNIST image datasets; they found their Alexnet-style CNN to achieve maximum training accuracy within only about 50 epochs of training, for all label noise-levels of training sets. In contrast, from among the noisy datasets there were no clear winners towards reaching convergence. During the very first epochs, each model’s progress in training accuracy corresponds directly to how clean the dataset is. This is clearly reflected in our experiments with JAVA-SMALL and SA datasets, and also with Arpit et al.’s experiments. It is only after a some time that models on different noise levels can arbitrarily overtake one another in this progression. From these observations we therefore see that the total time to converge is a useful metric in identifying the cleanest dataset, but may not be useful in determining the degree of noise among noisy datasets. Instead, for estimating the latter, the models’ training accuracy during the initial epochs may be useful.

Hacohen et al. [30] observed that there is order in the data that the network memorizes: different NNs *memorize* data in different order, yet, in contrast, when training a NN with real data, different NNs (with same architecture) *learn* the data in the same order. They observed this behavior on a text classification benchmark and several image classification benchmarks.

Morcos et al. [31], show that multiple NNs that generalize from the same data are more mutually similar (as in, converge

to similar representations) than networks that memorize. Similar work [32], [33] investigates generalization behavior through the lens of network architectures.

Fui et al. [34] study the generalization behavior in the NLP domain with a task-centric focus. They cite the growing gap between NLP task performance and the understanding of model generalization behavior [35], [36], and thus characterize generalization with respect to a specific NLP task (Named Entity Recognition). They find that the performance of existing models is significantly influenced by the degree to which training set entities have been set with the same label. Rosa et al. [37] propose a method to measure memorization effects in neural networks while they are trained for NLP tasks. Their method is based on a symmetric selection of comparable sets of seen versus unseen test words in training. Belinkov et al. [38] study learning patterns in neural network based natural language translation models and find that the inherent morphological structure of source and target languages may affect the generalization/memorization behavior of the neural models. They also show that different layers of the network learn different constructs of a language’s structure. While we do not conduct a full analysis of how code constructs of the samples in our datasets influence NN learning behavior, we show results of how the overall quality of datasets can have an effect in this regard.

IX. CONCLUSION

In this work, we quantitatively study the memorization and generalization behavior in several popular code intelligence models through adapting and replicating several experimental settings from the broader ML community [9], [10]. To the best of our knowledge, this is the first such work in the Software Engineering domain. We observe that, much as in other domains, memorization and generalization in CI models depend heavily on the architecture of the model and the dataset it has been trained on. A model trained on a “clean”, largely noise-free dataset shows better generalization and lower memorization than a model trained on noisy data – the latter have a more complex decision boundary and are thus more vulnerable to adversarial samples.

REFERENCES

- [1] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis, “Deep learning type inference,” in *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2018, pp. 152–162.
- [2] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, “Suggesting accurate method and class names,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 38–49. [Online]. Available: <https://doi.org/10.1145/2786805.2786849>
- [3] M. Allamanis, H. Peng, and C. A. Sutton, “A convolutional attention network for extreme summarization of source code,” in *Proceedings of the 33rd International Conference on Machine Learning, ICML*, 2016.
- [4] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, “Toward deep learning software repositories,” in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 334–345.
- [5] X. Gu, H. Zhang, D. Zhang, and S. Kim, “Deep api learning,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 631–642.
- [6] M. Allamanis, M. Brockschmidt, and M. Khademi, “Learning to represent programs with graphs,” in *International Conference on Learning Representations*, 2018. [Online]. Available: <https://openreview.net/forum?id=BJOFETxR>
- [7] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner *et al.*, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
- [8] M. Allamanis, “The adverse effects of code duplication in machine learning models of code,” in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 143–153. [Online]. Available: <https://doi.org/10.1145/3359591.3359735>
- [9] D. Arpit, S. Jastrzebski, N. Ballas, D. Krueger, E. Bengio, M. S. Kanwal, T. Maharaj, A. Fischer, A. Courville, Y. Bengio *et al.*, “A closer look at memorization in deep networks,” in *International Conference on Machine Learning*. PMLR, 2017, pp. 233–242.
- [10] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, “Understanding deep learning requires rethinking generalization,” in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24–26, 2017*, 2017.
- [11] H. J. Kang, T. F. Bissyandé, and D. Lo, “Assessing the generalizability of code2vec token embeddings,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2019, pp. 1–12.
- [12] M. R. I. Rabin, K. Wang, and M. A. Alipour, “Testing neural program analyzers,” *34th IEEE/ACM International Conference on Automated Software Engineering (Late Breaking Results-Track)*, 2019. [Online]. Available: <https://arxiv.org/abs/1908.10711>
- [13] R. Compton, E. Frank, P. Patros, and A. Koay, “Embedding java classes with code2vec: Improvements from variable obfuscation,” in *17th International Conference on Mining Software Repositories (MSR 2020, Seoul, Republic of Korea)*, 2020.
- [14] M. R. I. Rabin, N. D. Bui, K. Wang, Y. Yu, L. Jiang, and M. A. Alipour, “On the generalizability of neural program models with respect to semantic-preserving program transformations,” *Information and Software Technology*, p. 106552, 2021. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0950584921000379>
- [15] N. Yefet, U. Alon, and E. Yahav, “Adversarial examples for models of code,” *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: <https://doi.org.ezproxy.lib.uh.edu/10.1145/3428230>
- [16] M. R. I. Rabin, A. Mukherjee, O. Gnawali, and M. A. Alipour, “Towards demystifying dimensions of source code embeddings,” in *Proceedings of the 1st ACM SIGSOFT International Workshop on Representation Learning for Software Engineering and Program Languages*, ser. RL+SE&PL 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 29–38. [Online]. Available: <https://dl.acm.org/doi/10.1145/3416506.3423580>
- [17] Y. Wang, F. Gao, and L. Wang, “Demystifying code summarization models,” *arXiv preprint arXiv:2102.04625*, 2021.
- [18] M. R. I. Rabin, V. J. Hellendoorn, and M. A. Alipour, “Understanding neural code intelligence through program simplification,” in *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021, 2021.
- [19] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “Code2vec: Learning distributed representations of code,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, Jan. 2019. [Online]. Available: <http://doi.acm.org/10.1145/3290353>
- [20] U. Alon, O. Levy, and E. Yahav, “code2seq: Generating sequences from structured representations of code,” in *International Conference on Learning Representations*, 2019.
- [21] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, “A survey of machine learning for big code and naturalness,” *ACM Comput. Surv.*, vol. 51, no. 4, Jul. 2018. [Online]. Available: <https://doi.org/10.1145/3212695>
- [22] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. L. Traon, “Learning to spot and refactor inconsistent method names,” in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE ’19. IEEE Press, 2019, p. 1–12. [Online]. Available: <https://doi.org/10.1109/ICSE.2019.00019>
- [23] N. D. Q. Bui, Y. Yu, and L. Jiang, “Bilateral dependency neural networks for cross-language algorithm classification,” in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 422–433.
- [24] C. Gini, “On the measure of concentration with special reference to income and statistics,” *Colorado College Publication, General Series*, vol. 208, no. 1, pp. 73–79, 1936.
- [25] P. Bielik and M. Vechev, “Adversarial robustness for code,” in *International Conference on Machine Learning*. PMLR, 2020, pp. 896–907.
- [26] T. Maharaj, D. Krueger, and T. Cooijmans, “Memorization in recurrent neural networks,” in *Workshop on Principled Approaches to Deep Learning*, 2017.
- [27] A. LeClair, S. Haque, L. Wu, and C. McMillan, “Improved code summarization via a graph neural network,” in *Proceedings of the 28th International Conference on Program Comprehension*, ser. ICPC ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 184–195. [Online]. Available: <https://doi.org/10.1145/3387904.3389268>
- [28] X. Huo, M. Li, and Z.-H. Zhou, “Learning unified features from natural and programming languages for locating buggy source code,” in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, ser. IJCAI’16. AAAI Press, 2016, p. 1606–1612.
- [29] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, “Curating github for engineered software projects,” *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.
- [30] G. Hacohen, L. Choshen, and D. Weinshall, “Let’s agree to agree: Neural networks share classification order on real datasets,” in *International Conference on Machine Learning*. PMLR, 2020, pp. 3950–3960.
- [31] A. Morcos, M. Raghu, and S. Bengio, “Insights on representational similarity in neural networks with canonical correlation,” in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018. [Online]. Available: <https://proceedings.neurips.cc/paper/2018/file/a7a3d70c6d17a73140918996d03c014f-Paper.pdf>

- [32] S. Baluja and I. Fischer, "Learning to attack: Adversarial transformation networks," in *Proceedings of AAAI-2018*, 2018. [Online]. Available: <https://aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16529>
- [33] L. Schmidt, S. Santurkar, D. Tsipras, K. Talwar, and A. Madry, "Adversarially robust generalization requires more data," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 5019–5031.
- [34] J. Fu, P. Liu, and Q. Zhang, "Rethinking generalization of neural models: A named entity recognition case study," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 05, pp. 7732–7739, Apr. 2020. [Online]. Available: <https://ojs.aaai.org/index.php/AAAI/article/view/6276>
- [35] P. Rajpurkar, R. Jia, and P. Liang, "Know what you don't know: Unanswerable questions for squad," 2018.
- [36] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.
- [37] R. Rosa, T. Musil, and D. Mareček, "Measuring memorization effect in word-level neural networks probing," in *Text, Speech, and Dialogue*. Cham: Springer International Publishing, 2020, pp. 180–188.
- [38] Y. Belfinkov, N. Durrani, F. Dalvi, H. Sajjad, and J. Glass, "What do neural machine translation models learn about morphology?" in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 861–872. [Online]. Available: <https://www.aclweb.org/anthology/P17-1080>