

# The Adverse Effects of Code Duplication in Machine Learning Models of Code

Miltiadis Allamanis, Microsoft Research, Cambridge, UK

**Abstract**—The field of big code relies on mining large corpora of code to perform some learning task. A significant threat to this approach has been recently identified by Lopes et al. [19] who found a large amount of near-duplicate code on GitHub. However, the impact of code duplication has not been noticed by researchers devising machine learning models for source code. In this article, we study the effect of code duplication to machine learning models showing that reported metrics are sometimes inflated by up to 100% when testing on duplicated code corpora compared to the performance on de-duplicated corpora which more accurately represent how machine learning models of code are used by software engineers. We present an “errata” for widely used datasets, list best practices for collecting code corpora and evaluating machine learning models on them, and release tools to help the community avoid this problem in future research.

## 1 INTRODUCTION

MACHINE learning models of source code have recently received great attention from the research community. At the intersection of the research fields of software engineering, programming languages, machine learning and natural language processing, multiple communities have been brought together into the field of “Big Code” or “code naturalness” with many fruitful results [3]. Commonly, research in this area relies on large corpora of code which can be used as training and test sets, allowing these methods to learn about coding practice at a large scale.

However, there is a looming crisis in this newly-founded area, caused by a disproportionately large amount of code duplication. This issue — first observed by Lopes et al. [19] — refers to the fact that multiple clones or near-clones appear in large corpora of code, such as those mined from GitHub repositories. This is because software engineers often copy — partially or entirely — files from other projects [19, 11]. Despite the findings of Lopes et al. [19], the research community has not yet analyzed how code duplication impacts its research, the machine learning models it devises and the negative effects on the practical tools it creates.

In this article, we first analyze the impact that code duplication can have on machine learning models. We discuss the biases introduced when evaluating models under duplication and show that duplication can cause the evaluation to overestimate the performance of a model compared to the performance that actual users of the model observe. Then, we replicate the work of Lopes et al. [19] across ten corpora that have been used in “big code” research and we measure the impact of duplication across datasets and machine learning models showing that the performance observed by a user is up to 50% worse compared to reported results. We hope that this article will help researchers better understand the issue of code duplication and allow them to better measure its impact. At the same time, we provide tools and best practices, that can help overcome pitfalls when researching machine learning methods that employ source code data. Briefly, the contributions of our work are:

- a study of the effects of code duplication on machine learning models of source code;
- an open-source, cross-platform tool that detects near-duplicates in C#, Java, Python and JavaScript along with an “errata” for

existing datasets, listing existing duplicate files;

- a suggested list of best practices with respect to the code duplication problem and machine learning models.

## 2 CODE DUPLICATION & MACHINE LEARNING

Code duplication refers to the idea that a snippet of code appears multiple times with no or small differences within a corpus of code. Code clones [25] are a widely studied form of code duplication. The potentially adverse effects of code cloning to the software engineering process have been extensively studied in the literature, although there are cases where cloning is not considered bad [18]. The existence of duplicates has been noticed long time ago in empirical software engineering [27] but with the ability to create large corpora this issue became more noticeable [19]. In this study, we are interested on the effects of presence of near-duplicates in codebases specifically to machine learning models trained on code<sup>1</sup>. These requirements, sets different parameters for searching, understanding and classifying code duplication. To understand the effects of duplicates in code corpora, we first need to understand how machine learning models of code can be used in practice and how code duplication affects model performance.

At a high-level, the goal of machine learning models of source code is to train a model on existing code, such that the learned models capture the statistical properties of some particular aspect of coding practice, which can then be useful within a software engineering tool. Some examples of such models include:

- autocompletion models [15, 22, 13, 20] aiming to assist code construction in an editor when a developer is writing new code;
- type prediction models [23, 14] where the goal is to infer the types of variables (*e.g.* in JavaScript) of new, previously untyped, programs;
- code summarization [2, 16, 7, 5] where the goal is to summarize some code into a short natural language utterance.

In most models, like in the aforementioned examples, the goal is to use trained models to provide recommendations and insights on

1. We use the terms “duplicate” and “near-duplicate” interchangeably to refer to code that is highly similar but not necessarily identical.

new and unseen code when the software engineer is creating or maintaining it. Essentially, this necessitates that machine learning models *generalize* well to new source code or — in statistical machine learning terms — to *faithfully model the “true” distribution of the data as it will be observed by the particular use case of the tool*.

To achieve this, machine learning-based software engineering tools need to be evaluated on the true data distribution of their use-case, since it is the one that software engineers will observe when using the tool. For example, for a token-level code autocompletion tool the true data distribution refers to the predicted next token that the developer will actually type. However, it is very rare that researchers get to train their model and measure its performance by directly observing its use by engineers, *i.e.* the true data distribution. Instead, a common practice is to split any existing dataset into two parts: a training set that is used to train the machine learning model and a test set where the performance of the model is evaluated. However, for this evaluation methodology to be accurate, an important assumption is commonly made: each of the data points need to be *independent and identically distributed* (i.i.d) over the true distribution of data of the use-case. This is *not* an unreasonable assumption and is widely and successfully used in machine learning and data mining research and practice [21, §7.3]. It is exactly this assumption that code duplication strongly violates for many of the use cases of machine learning models of code.

**The True Data Distribution** The “true” data distribution depends largely on the target use case. For example, a token-level autocompletion model will never need to suggest duplicate code, since it is reasonable to assume that this is produced by copy-pasting rather than a developer writing duplicate code token-by-token using token-level code autocompletion. In this work, we are going to make two assumptions that will help us measure the impact of code duplication on machine learning models. First, the true data distribution of the target application contains no duplicates. Second, we assume that duplication happens only across different files, similar to Lopes et al. [19]. This means that smaller amounts of code duplication, such as clones that span only a few lines, are *not* duplicates. The last assumption allows us to efficiently find duplicates and addresses the possibility that a machine learning-based software engineering tool can still be useful when a few lines of code are cloned. For example, a type prediction tool may still be required to suggest types even when a few lines of code have been copy-pasted.

It should be emphasized that code duplication is harmful only when it does *not* reflect the distribution over which we expect the tool to be used. This means that there are use cases where learning from duplicated code would be the correct practice. For example, if we were interested in test coverage over a codebase, code duplicates within the codebase should be included in our study.

**Concepts and Definitions** Assume a dataset  $D$  of source code snippets that is split into a training and a test set (Figure 1). We distinguish three types of duplicates: (1) “in-train” duplicates, *i.e.* snippets duplicated within the training set; (2) “in-test” duplicates, *i.e.* duplicates within the test set; and (3) “cross-set” duplicates that are snippets that appear both in the training and test sets.

**Duplication Bias** In machine learning a quantity  $f$ , such as the loss function used for training or a performance metric, is usually estimated as the average of the metric computed uniformly

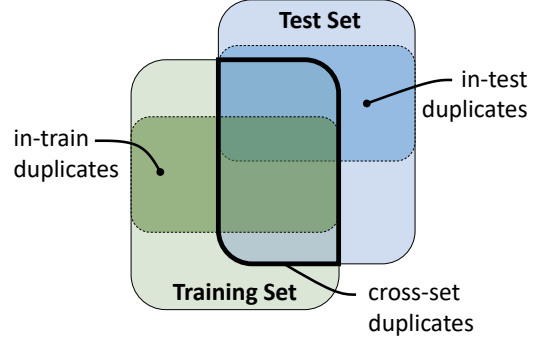


Fig. 1. Schematic description of types of duplicates. The dashed boxes indicate the subset of files that are duplicates within each set.

(because of the i.i.d. hypothesis) over the training or test set(s). Specifically, the estimate over a dataset  $D = \{x_i\}$  is computed as

$$\hat{f} = \frac{1}{|D|} \sum_{x_i \in D} f(x_i). \quad (1)$$

Duplication biases this estimate by overweighting some  $x_i$ . Specifically, we can equivalently rewrite  $D$  as a multiset  $X = \{(x_i, c_i)\}$  where  $c_i \in \mathbb{N}_+$  is the number of times that sample  $x_i$  is found in the dataset. Therefore, we can rewrite Equation 1 as

$$\hat{f} = (1-d) \underbrace{\frac{1}{|X|} \sum_{x_i \in X} f(x_i)}_{\text{unbiased estimate } \tilde{f}} + d \underbrace{\frac{1}{|X|} \sum_{x_i \in X} (c_i - 1)f(x_i)}_{\text{duplication bias}} \quad (2)$$

where  $d = \frac{|D|-|X|}{|D|} = \frac{\sum c_i - |X|}{|D|}$  is the duplication factor, *i.e.* the proportion of the samples in the dataset that are duplicated ( $c_i > 1$ ). As it can be seen, the larger the duplication factor, the larger the effect of the duplication bias.

Within the machine learning context, the duplication bias in the training loss can cause a model to overweighting some training samples (the in-train duplicates). During testing, the duplication bias skews the performance metric. Furthermore, we expect cross-set duplicates to improve any metric taking advantage of the fact that those samples were seen during training and give the illusion that the model generalizes, where in fact it memorizes duplicates.

### 3 MEASURING DUPLICATION

To measure code duplication we need a method that detects (near) duplicate files along a large corpus of code. As we discussed in the previous section, we are interested in file-level duplication and thus we follow SourcererCC’s [26] token-level duplication detection with some small modifications described next.<sup>2</sup>

**Detecting near-duplicates** Our goal is to use a high-precision method for detecting duplicates. Although detecting exact duplicates is somewhat straightforward, this misses a substantial number of near-exact matches that differ only in a few tokens. To achieve this, we follow the general principles in SourcererCC [26], we tokenize each file and extract all identifier and literal tokens. For each file, we build two “fingerprints”, a set  $T_0$  and a multiset  $T_1$  of all the identifiers and literals. We consider two files  $i$  and  $j$  to be duplicates, if the Jaccard similarities  $J(T_0^i, T_0^j)$  and

<sup>2</sup> SourcererCC is released under a non-permissive open-source license, which restricts some uses and therefore, we implemented a new tool, which we release under a permissive license.

$J(T_1^i, T_1^j)$  are above the thresholds  $t_0$  and  $t_1$  respectively. In this work, we set  $t_0 = 0.8$  and  $t_1 = 0.7$  based on experimentation on a small C# dataset, but we notice that duplicate detection is fairly robust to these thresholds. Files with less than 20 identifiers are not considered duplicates and are excluded from our analysis. Finally, to improve the speed of the tool, we make the simplifying assumption that similarity is transitive. Although this does not generally hold, we found that this does not impact the accuracy of the tool. Finally, since computing the Jaccard similarities is embarrassingly parallel, we simply compare all combinations of files for similarity.

Our tool is quite fast. For example, on an Azure F16 machine (16 cores), our method detects duplicates among 112k files in the JavaScript-150k corpus (discussed next) in 5 hours. We open-source the duplication-detection code online under a permissive license at <https://github.com/Microsoft/near-duplicate-code-detector>. It contains tokenizers for Java, JavaScript, C# and Python but can easily be extended to other languages. The deduplication tool simply accepts a JSONL file (*i.e.* a file containing a valid JSON per line) containing an id of each file (*e.g.* its filepath) and a list of identifier and literal tokens within that files. It returns a JSON file with the near-duplicate clusters. We also provide a faster, but approximate Python tool that works on the same principles within the `dpu-utils` package at <https://github.com/Microsoft/dpu-utils>.

**Duplication Statistics** Armed with a reasonable method for detecting duplication, we now report code duplication statistics for ten datasets that have been used for machine learning on code. It should be noted that for the studied datasets all authors have taken significant steps to remove exact file-level clones. However, this process missed a large number of (near) duplicate files, that may differ in minor aspects, such as whitespace, code comments and other small code modifications. Table 1 reports the results. We note that for the JavaScript-150k dataset our tool was able to process only 112k files and therefore we report results on those files. The rest of the files are ignored. The results show that only 79.8% of files in Java-Large and 79.3% in JavaScript-150k are unique and removing the rest of the files would create a deduplicated dataset. We additionally show the expected proportion of the test set that is made of cross-set duplicates. Note that these statistics are when datasets are split across files. When splitting across projects, this percent is most commonly reduced. For example, splitting the Java-Large dataset across projects, following the author-provided split, only 8.9% of the test set is made of cross-set duplicates (compared to the average of 24.1% when splitting across files). This suggests that splitting across projects — when possible — is a helpful strategy. Why are cross-set duplicates so common? Given a dataset split where each file is added to the training set with probability  $k$  and to the test set with probability  $1 - k$ . A cluster of  $v \geq 2$  near-duplicate files has a probability of  $1 - k^v - (1 - k)^v$  of introducing a cross-set duplicate. For a 60-40 split ( $k = 0.6$ ), and a file that is duplicated twice ( $v = 2$ ) the probability of these two files being cross-set duplicates is 48%. Thus despite that each file has a probability of  $k = 40\%$  to be included in the testset, the probability of having a cross-set duplicate is larger.

As expected, smaller datasets, such as those collected over a small and curated set of projects suffer less from duplication. The Concode dataset [17] seems to be the one suffering the most from duplication, by having about 68.7% of its methods be duplicates. However, it should be appreciated that Concode and the Python

docstring datasets are datasets where a per-function granularity has been used, due to the way the data was made available. It might be the case that if we performed a per-function analysis in the other datasets, that their duplication statistics also worsen. Note that once the data is split into training-test sets, the percent of cross-set duplicates is smaller than the full dataset duplication factor, since a noticable proportion of duplicates become in-train or in-test duplicates. Finally, we note that **the duplication in all datasets is significantly smaller than that reported by Lopes et al. [19]**. This should be attributed to the fact that the corpus collected by Lopes et al. [19] is orders of magnitude larger than any of the datasets in Table 1. Authors of the datasets examined here have made efforts to deduplicate and filter the collected corpora by removing most low popularity projects and some number of exactly duplicated files. We release the near-duplicates files at <https://dpupublicdata.blob.core.windows.net/duplicates/errata.zip> We hope that these lists can be used as dataset “errata” in future work.

**Human Evaluation** Our near-duplicate detection algorithm makes some approximations to make the search computationally efficient. This raises the question about its precision. The author of this paper inspected 100 random pairs of duplicates for the Javascript-150k dataset [24] and 100 random pairs from the Java-Large dataset [5] and annotated each pair as a true or false positive. Overall, the duplicate detection achieves perfect precision for both datasets. Looking at the duplicates, we make a few qualitative, empirical observations. First, we observe that a large majority of duplicates share the same file name. For the JavaScript-150k, the majority of near-duplicates is of two kinds: (a) different versions of the same file (b) configuration-like files that differ mostly on the configuration values. In contrast, in the Java-Large dataset we find more exact clones, duplicates of the same file but of a different version and boilerplate code. For the C# corpus [4], we note that near-duplicates were mostly found within projects and largely include autogenerated files. This is because the creator of that dataset — and author of this work — had explicitly checked for and removed duplicates when creating the dataset, but only across projects and under stricter thresholds.

## 4 IMPACT ON MACHINE LEARNING MODELS

So far, we have established that code duplication can — theoretically — have adverse effects to the way machine learning models of code are trained and evaluated. But is this actually the case? Measuring the effect of duplication on machine learning models in a generalized way is not entirely possible. This is because machine learning models differ widely in their characteristics and we expect different models and tasks to be affected differently by code duplication. To study the impact of code duplication, we create experimental settings that illuminate separate aspects of the problem. In Section 4.1 and Section 4.2 we focus on code autocompletion through language modeling. This allows us to do an in-depth study of a single model and a few factors of variation. Then in Section 4.3 we train state-of-the-art models on other tasks. In all cases, we assume a random 50-10-40 train-validation-test split over the full dataset. If a model does not use a validation set, we use the validation samples in the training set.

**Terminology** In the absence of existing terms for the code duplication problem, we introduce a few new terms and annotate them with a mnemonic symbol to help the reader. Given a training-test split and by interpreting Equation 2, we have two possible types of training:

TABLE 1  
Duplication Statistics across Existing Corpora over all files (across any provided splits) with more than 20 identifier and literal tokens.

Name	Relevant Publications	# Files ( $\times 1000$ )	Unique Duplicate Clusters ( $\times 1000$ )	Duplicate Files – $d$ (%)	Duplicate Average	Duplicate Clusters Size Median	% Expected Cross-Set Duplicate Files within Test (6:4 split)
C#-19	[4]	28.3	0.9	10.6	4.4	2	11.7
Concode – Java*	[17]	229.3k	30.8	68.7	6.1	3	77.8
Java GitHub Corpus	[1]	1853.7	682.7	24.8	2.1	2	29.6
Java-Small	[5], [2]	79.8	2.4	4.7	2.6	2	5.7
Java-Large	[5]	1863.4	195.0	20.2	2.9	2	<sup>†</sup> 24.1
JavaScript-150k	[24]	112.0	8.6	20.7	3.7	2	24.1
Python-150k	[24]	126.0	5.4	6.6	2.6	2	8.0
Python docstrings v1*	[7]	105.2	17.0	9.2	2.3	2	11.2
Python docstrings v2*	[7]	194.6	24.2	31.5	3.5	2	37.4
Python Autocomplete*	[12]	70.4	8.9	20.3	2.6	2	24.5

\*We place one method per file, since the corpus is split across methods. <sup>†</sup>When the dataset is split across projects, as in the author provided split, this falls to 8.9%.

TABLE 2  
Terminology for Measuring Performance based on Kinds of Duplicates in Training and Test Sets

Training	Test Set		
	no dups	w/ cross-set dups	w/ all dups
Biased	Unbiased Test	Cross-Set Biased	Fully Biased
Unbiased	Fully Unbiased	–	–

- **Unbiased Training** All duplicates are removed ( $c_i = 1, \forall i$ ) and an unbiased loss function  $\bar{f}$  is employed during training;
- **Biased Training** All in-train duplicates are retained and the biased loss function  $\hat{f}$  is used. Since existing work does not adequately de-duplicate its datasets, it employs biased training.

We now turn our attention to the testing terminology. Within a testset we distinguish two types of duplicates: the cross-set duplicates, and the in-test duplicates (Figure 1). This leads to four types of metrics, summarized in Table 2 and discussed next. The mnemonic symbols can be interpreted as Venn diagrams of the training and test sets. When a set contains duplicates it is shaded (indicating bias on that set), otherwise it is left blank. Finally, we note that when we remove duplicates, we keep exactly one file from each cluster of near-duplicates, such that any duplicate file is used exactly once.

- **Fully Unbiased** that represents an “ideal world”, where all duplicates are removed both from training and test sets and the training and test sets are completely disjoint, allowing us to perform unbiased training and testing.
- **Unbiased Test** that represents the performance when the test set contains no duplicates. This is equivalent to the performance observed by a user who is using a machine learning model under the true data distribution, but the model has been trained in a biased way.
- **Cross-set Biased Test** which is the performance measured when performing a biased training and using a test set that only contains cross-set duplicates, but no in-test duplicates.
- **Fully Biased Test** where training and testing happens on the duplicated (original) dataset. This is the metric that is reported by existing work. Compared to the cross-set biased test () these metric is additionally biased by the in-test duplicates. Because this bias is arbitrary, it inhibits us from measuring the exact effect of code duplication. For this reason, we do *not*

report this metrics, but note that empirically it is always very close to the cross-set biased test metrics ().

It should be noted that for estimating the impact of duplication on machine learning models it is technically incorrect to directly compare the fully unbiased performance () with the unbiased test () to measure the effect of code duplication since model training is performed on somewhat different datasets and we have no way to distinguish between a model’s capacity to learn from more (but duplicated) data and the effect of duplication. In practice we observe small differences between deduplicated () and unbiased testing () and we report both.

#### 4.1 Biased vs. Unbiased Performance

As we discussed in Section 2, code duplication can result in reporting better performance compared to the one that a user would actually observe. In this and next section, we focus on the effects of duplication on a single task, namely language modeling. By focusing on a single task and model we can do a deep-dive on various aspects of code duplication and measure more subtle effects. Later, in Section 4.3 we measure the impact of code duplication on other models and on other tasks.

**Autocompletion via Language Modeling** has been extensively studied both in natural language and in source code. The goal of language models is to capture the statistical characteristics of a language such that the output appears to be “natural”. Language models have been used for autocompletion and variable naming [3] and it would be unreasonable to assume that the true distribution of those use cases of language models contains duplicate code.

To demonstrate the effects of code duplication we employ a relatively simple, yet powerful neural language model. The goal is to show how even relatively simple models are severely impacted by duplication and draw conclusions that generalize to other models. We follow the early work of Bengio et al. [8] for token-level language modeling. Our neural language model (NLM) is described as




$$P(t_i) = \text{softmax}(E_o \sigma(W_c [E_i h(t_{i-1}) \dots E_i h(t_{i-c})]) + \mathbf{b}) \quad (3)$$

where  $E_o \in \mathbb{R}^{|V| \times K}$  and  $E_i \in \mathbb{R}^{D \times |V|}$  are the output and input embedding matrices of tokens,  $W_c \in \mathbb{R}^{K \times cD}$  is a matrix,  $\mathbf{b}$  is a bias vector, and  $h(\cdot)$  is a function that take a token and converts it to an one-hot vector. All parameters are learned. We train our model to minimize the empirical cross-entropy on the training set, and pick the model that achieves the best performance on the validation set. For simplicity, in this work we set  $K = D$ . Throughout this section,










TABLE 3

Impact of Duplicates on Evaluation Performance on a simple Language Modeling Task on the JavaScript-150k [24] dataset.

Metric	Performance			
			$\Delta(\text{unbiased}, \text{cross-set biased})$	
Acc (%)	49.1	55.1	-10.9%	49.2
Acc-ID (%)	8.6	17.7	-51.4%	8.3
MRR	0.674	0.710	-5.1%	0.674
MRR-ID	0.136	0.224	-39.3%	0.132
PPL	9.4	7.5	+25.3%	9.4
PPL-ID	76.1	55.4	+37.4%	82.3

we set  $D = 128$ , train with RMSProp [28] and early stopping. As a vocabulary  $V$ , we use the top 10k most frequent tokens. All results are averaged across 5 runs on random splits of the data.

**Performance** To accurately measure the impact of duplication we need to be able to make a fair comparison on the evaluated results. To achieve this, we replicate the conditions of existing work, *i.e.* we perform biased training on our models. We then compute the unbiased () and cross-set biased () performance metrics. Table 3 shows the measured effect of duplication on the JavaScript-150k [24] dataset. Specifically, it highlights the relative difference between the unbiased-test () and cross-set biased () metrics, which can directly measure the effect of code duplication on the metrics. We also report the fully-unbiased metrics (). The metrics computed are (a) the accuracy of correctly predicting the next token (Acc; higher is better), (b) the mean reciprocal rank (MRR; higher is better) over the tokens and (c) the perplexity (PPL; lower is better) assigned by the neural language model. Unknown tokens are counted as incorrect when computing accuracy and MRR. We also compute focused metrics on identifiers since they have been proven to be the hardest to predict [1, 9, 20]. We note that we also computed the fully biased () metrics. On average, the NLM’s performance is similar to the cross-set biased () performance. This is expected, since the in-test bias is mostly random.

Based on the results, we notice that *all* metrics are affected to a different extent by code duplication. The difference ( $\Delta(\text{unbiased}, \text{cross-set biased})$ ) ranges from a few percentage points to halved performance. This suggests the seriousness of the code duplication problem. Furthermore, we observe that the identifier-related metrics are those that are more severely affected by code duplication. This is expected, since code duplication makes identifiers, which would otherwise appear sparsely, appear more frequently and predictably.

Thus, it should be appreciated that *not all metrics and tasks are equally affected by code duplication*. For example, if an application requires predicting code’s non-identifier tokens (*e.g.* as in Campbell et al. [10]), duplication would have a much smaller effect compared to an autocomplete application for predicting identifiers.

## 4.2 Model Capacity and Impact on Code Duplication

Duplication has an observable impact on the performance of machine learning models of source code. However, not all models are impacted in the same way. Indeed, some models may be more prone to memorizing code duplicates than others. Since we cannot measure capacity across different models, we focus on the NLM model and show that model capacity plays a crucial role by varying the model’s capacity.

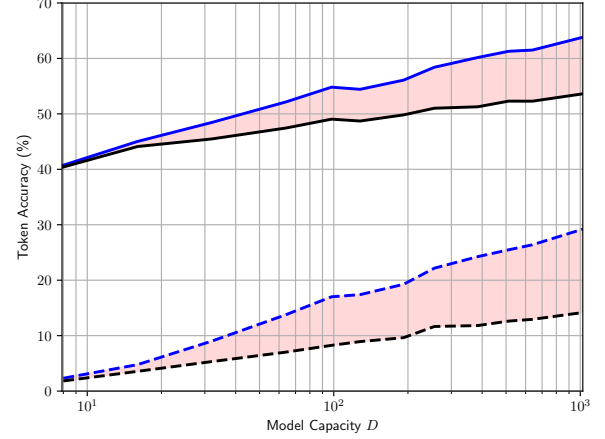


Fig. 2. The impact of code duplication on models of different capacity trained on JavaScript-150k. The solid lines show the accuracy of the NLM model when predicting all tokens, whereas the dashed lines show the accuracy of predicting only identifiers. Blue lines indicate the cross-set biased accuracy, and black ones show the unbiased test accuracy. The larger the capacity of the model, the more severe the impact of code duplication (red shaded area).

Figure 2 plots the NLM accuracy of predicting tokens (solid lines) or only identifiers (dashed lines). As a proxy for measuring the capacity of the model, we vary the dimensionality  $D$  of the vector representations. Although there are other methods to increase the capacity of the model (*e.g.* by adding more layers), increasing the dimensionality is a reasonable option for observing the effect of code duplication. The shaded (red) area in Figure 2 shows, as expected, that the (negative) effect of duplication increases as model capacity increases. This can be attributed to the fact that additional capacity is probably used to memorize duplicated code. Therefore, we observe that *models that have larger capacity tend to be more heavily impacted by code duplication*.

This suggests an additional and very important observation: *Comparison of different models under code duplication may not be indicative of their real performance*. This is because some models, having more capacity, can take better “advantage” of code duplication and report improved results only because they are able to better memorize the duplicated cross-set samples.




## 4.3 Other Models and Tasks

In the previous sections, we evaluated the impact of code duplication over a relatively simple neural language modeling task where we could control various factors of variation and observe how different aspects of a model are affected by code duplication. Although the reader probably already suspects that code duplication affects many other models, here we select a few state-of-the-art models and tasks to evaluate the impact of code duplication.

**Tasks and Models** We select four reasonably well-known tasks in the literature. Our goal here is *not* to be comprehensive but to illustrate the adverse effects of duplication across a diverse set of models and tasks where code duplication is *not* part of the true data distribution. Note that we re-split the datasets randomly assigning each file to a set. This represents cases where a model can be used within projects, which is often a realistic scenario in machine learning-based software engineering tools. Splitting across projects,

TABLE 4

Impact of Code Duplication on Performance over a Series of Methods/Tasks.  $\Delta$  refers to the relative improvement (worsening). Note that some of the evaluated methods are evaluated on different datasets compared to those used in the original works.

Metric	Performance			
			$\Delta(\text{code2vec}, \text{JSNICE})$	
<u>Task</u> : Method Naming <u>Model</u> : code2vec [6]				
<u>Dataset</u> : Reshuffled Java-Large [5]				
F1 (%)	44.71	50.98	-12.3%	46.04
Precision (%)	53.00	58.92	-10.5%	54.51
Recall (%)	38.67	44.93	-13.9%	39.85
<u>Task</u> : Variable Naming <u>Model</u> : JSNICE [23]				
<u>Dataset</u> : Reshuffled & Slightly Reduced JavaScript-150k [24]				
Accuracy (%)	34.44	55.04	-37.4%	29.41
<u>Task</u> : Code Autocompletion <u>Model</u> : PHOG [9]				
<u>Dataset</u> : Reshuffled & Slightly Reduced JavaScript-150k [24]				
Accuracy (%) – Types (all)	71.80	75.69	-5.1%	72.95
Accuracy (%) – Values (all)	71.19	77.75	-8.4%	71.35
– Identifiers	48.94	61.43	-20.3%	49.05
– String Literal	25.62	43.89	-41.6%	24.51
<u>Task</u> : Docstring Prediction <u>Model</u> : Seq2Seq [7]				
<u>Dataset</u> : Python Docstrings v1 [7]				
BLEU	12.32	13.86	-11.1%	—

can slightly reduce the impact of code duplication, depending on the characteristics of each dataset.

- The **method naming** task of predicting the name of a method (function) given the body of the function (*i.e.* summarization). Here we run the open-source state-of-the-art code2vec model [6] on the Java-Large corpus [5].
- **Variable Naming** which is the task of predicting the names of variables of a snippet of possibly obfuscated code. Note that we assume that the task is to deobfuscate new, previously unseen code rather than code whose deobfuscated form is known, as discussed in Raychev et al. [23].<sup>3</sup> We run the state-of-the-art non-neural JSNICE model of Raychev et al. [23] on the JavaScript-150k [24] dataset using the author-provided data extraction utility. Note that the split differs from the original one and some of the files are missing as discussed in Section 3.
- **Code Autocompletion** which is the language modeling task used in the previous section. Instead of using the neural model of Section 4.1, we employ the PHOG model of Bielik et al. [9] another non-neural model. Since the code is not open-source yet, Pavol Bielik kindly helped with training and testing on that model. We provided the split on the JavaScript-150k [24] dataset for this task.
- **Documentation Prediction** which is the task of predicting the documentation (*e.g.* docstring) of a method/function using its implementation. Here, the most recent approach is that of Barone and Sennrich [7] that use neural machine translation to




3. This excludes multiple usage scenarios that Raychev et al. [23] have observed in practice when they deployed JSNICE as a service. Specifically, in personal correspondence they mentioned that submissions to the public JSNICE service often contain large bundled parts of various projects and libraries. As developers use different versions of many common libraries, JSNICE needs to train on all these versions, despite being approximately duplicate. For all these use cases that are observed in practice, removing duplicates would be wrong since a deduplicated dataset does *not* match the true data distribution (Section 2) of the use case.

“translate” code to documentation. Since the authors provided the output of their model, we use it directly to compute the performance on biased-training, instead of performing our own training.

Additionally, we considered the Variable Misuse task [4] which is the task of predicting which type-correct, in-scope variable to use at a given variable usage location. The only dataset that is available here is that of Allamanis et al. [4]. However, within the variable misuse sites only 0.5% of the datapoints are duplicated. This is due to the fact that the C#-19 dataset [4] duplicates are mostly files that are semi-auto-generated, such as assembly information files and resource files that contain very few candidate variable misuse sites. Given the duplication of 0.5% we will *not* consider this task. Note that for all the tasks considered above, it would be unreasonable to assume that the true distribution reflecting the particular use case of each tool to contain any duplicates.

We train/test all these models with the default parameters as provided by the authors in their open-source releases of their code. It should be noted that none of these results should be interpreted as negative results for any of the existing methods. Our study merely illustrates how different tasks and state-of-the-art models are also affected by code duplication. For example, the simple neural language model of Section 4.1 still has a significantly worse performance compared to PHOG, even after removing code duplicates.

**Analysis of Results** Overall, we observe in Table 4 that removing code duplicates noticeably reduces the measured performance of all methods ( $\Delta(\text{code2vec}, \text{JSNICE})$ ). Although all metrics worsen, the effect differs. For example, JavaScript-150k and Java-Large have very similar (file-level) duplication but the impact of duplication on the evaluation metrics of PHOG [9] and code2vec [6] is quite different. This can be attributed to two factors (a) different models are affected differently (*e.g.* because of their inductive biases) (b) different tasks are affected differently by code duplication.

An interesting observation is that training models with a biased dataset () almost always result to worse performance than training them in an unbiased fashion (*e.g.* without duplicates, ). This is probably due to the fact that part of each model’s capacity is spent on learning about duplicates hindering the performance of the model on the deduplicated test set. Thus, *training on a biased dataset usually has negative effects on model performance as observed by end-users* (). JSNICE, a non-neural method, seems to be an exception. This may be attributed to the fact that the reduced size of the deduplicated dataset harms performance more than code duplicates. Finally, as we already observed, different metrics are affected differently. A consistent theme has been that identifier-related metrics (*e.g.* accuracy of identifiers of PHOG and of the NLM) are the most severely impacted. Generalizing this, we can conclude that this can be attributed to the sparsity [3] of some code constructs (*e.g.* identifier names): *Rare elements of code are hard to predict. Metrics and methods heavily relying on such constructs are those most severely affected by code duplication.*

## 5 MITIGATING DUPLICATION: BEST PRACTICES

In the previous sections, we believe that we were able to document and sufficiently study the negative impact of code duplication on machine learning models of code. We observed that:

- Code duplication affects all metrics and the performance observed by end-users is often significantly worse than the one reported by evaluation metrics.

- Different metrics and applications are affected differently by code duplication.
- Powerful models that have larger capacity are impacted more by code duplication.
- Comparing different models with duplicated code corpora can be unfair to models with smaller capacity.

**Best Practices** Through this article, a set of best practices arise that we recommend researchers and practitioners to follow:

- **Understanding the True Data Distribution** for the target use case should be considered first. Does the distribution over which we expect the tool to be used contain duplicates? If so, then de-duplication needs to be performed.
- **Data Collection** Collecting large datasets in batch should be done carefully and deduplication methods — like the one proposed by Lopes et al. [19] or the one used in this work<sup>4</sup> — should be used to deduplicate the collected corpus. Simply removing exact matches and forks is a reasonable but clearly insufficient first step. **Splitting the dataset across different projects**, when possible, usually helps a lot, but duplication often still exists.
- **Use of Existing Datasets** This work demonstrates varying levels of duplication for different datasets. However, duplication exists to some extent in all existing datasets. When using existing datasets, we suggest using the errata files provided in this work to remove duplicates.
- **Model Capacity** Models that have a large capacity to memorize training data, suffer the most from the duplication problem and special attention should be given when evaluating such methods. Furthermore, researchers should include naïve memorization methods in their baselines (e.g. *k*-means). If these baselines perform “too well” compared to other widely-used models, this can indicate a duplication issue.

## 5.1 Conclusions

We hope that this work informs the research community about the negative effects of code duplication on the evaluation of machine learning models and informs practitioners about potential pitfalls when deploying such tools in practice. Removing exact and near duplicates will allow for more accurate comparison of machine learning models and methods and will lead to better machine learning-based software engineering tools.

Finally, despite code duplication’s negative effects many interesting research opportunities arise. Code duplication across code is a fact of software engineering life and therefore methods should embrace it. The work of Hashimoto et al. [12] that combine retrieval methods that find similar snippets within a database of code and then perform edits over those examples is an interesting example of such a direction. Interesting research questions such as “Can new machine learning tools be created that are robust to code duplication?” and “Can we usefully exploit near-duplicates to produce better software engineering tools?” seem to arise as interesting research problems.

4. The tool can be found at <https://github.com/Microsoft/near-duplicate-code-detector> and an approximate version within the `dpu-utils` Python package at <https://github.com/Microsoft/dpu-utils>.

## ACKNOWLEDGMENTS

The author would like to thank Marc Brockschmidt for useful discussions and suggesting the mnemonic symbols, Patrick Fernandes for first noticing the severity of the duplication problem in existing datasets and bringing it to the attention of the author and an anonymous reviewer of some other work of the author that insisted that code duplication is not an important issue with existing datasets. Finally, the author would like to thank Pavol Bielik for running the evaluation on PHOG [9], Uri Alon for useful discussions on the Java-Large corpus and useful comments on a draft of this work, Charles Sutton and Earl Barr for helpful discussions, suggestions and corrections.

## REFERENCES

- [1] M. Allamanis and C. Sutton, “Mining source code repositories at massive scale using language modeling,” in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. IEEE Press, 2013, pp. 207–216.
- [2] M. Allamanis, H. Peng, and C. Sutton, “A convolutional attention network for extreme summarization of source code,” in *Proceedings of the International Conference on Machine Learning (ICML)*, 2016, pp. 2091–2100.
- [3] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, “A survey of machine learning for big code and naturalness,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 81, 2018.
- [4] M. Allamanis, M. Brockschmidt, and M. Khademi, “Learning to represent programs with graphs,” in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.
- [5] U. Alon, O. Levy, and E. Yahav, “code2seq: Generating sequences from structured representations of code,” *arXiv preprint arXiv:1808.01400*, 2018.
- [6] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “code2vec: Learning distributed representations of code,” *arXiv preprint arXiv:1803.09473*, 2018.
- [7] A. V. M. Barone and R. Sennrich, “A parallel corpus of Python functions and documentation strings for automated code documentation and code generation,” in *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, vol. 2, 2017, pp. 314–319.
- [8] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, “A neural probabilistic language model,” *Journal of Machine Learning Research (JMLR)*, vol. 3, no. Feb, pp. 1137–1155, 2003.
- [9] P. Bielik, V. Raychev, and M. Vechev, “PHOG: Probabilistic model for code,” in *Proceedings of the International Conference on Machine Learning (ICML)*, 2016, pp. 2933–2942.
- [10] J. C. Campbell, A. Hindle, and J. N. Amaral, “Syntax errors just aren’t natural: improving error reporting with language models,” in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. ACM, 2014, pp. 252–261.
- [11] M. Gharehyazie, B. Ray, M. Keshani, M. S. Zavoost, A. Heydarnoori, and V. Filkov, “Cross-project code clones in GitHub,” *Empirical Software Engineering*, pp. 1–36, 2018.
- [12] T. B. Hashimoto, K. Guu, Y. Oren, and P. Liang, “A retrieve-and-edit framework for predicting structured outputs,” in *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, 2018.

- [13] V. J. Hellendoorn and P. Devanbu, “Are deep neural networks the best choice for modeling source code?” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 763–773.
- [14] 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*. ACM, 2018, pp. 152–162.
- [15] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, “On the naturalness of software,” in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 837–847.
- [16] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, “Summarizing source code using a neural attention model,” in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, 2016, pp. 2073–2083.
- [17] —, “Mapping language to code in programmatic context,” in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018, pp. 1643–1652.
- [18] C. J. Kapsner and M. W. Godfrey, ““Cloning considered harmful” considered harmful: patterns of cloning in software,” *Empirical Software Engineering (ESEM)*, vol. 13, no. 6, p. 645, 2008.
- [19] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, “DéjàVu: a map of code duplicates on GitHub,” *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 84, 2017.
- [20] C. Maddison and D. Tarlow, “Structured generative models of natural source code,” in *Proceedings of the International Conference on Machine Learning (ICML)*, 2014, pp. 649–657.
- [21] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.
- [22] V. Raychev, M. Vechev, and E. Yahav, “Code completion with statistical language models,” in *Proceedings of the Symposium on Programming Language Design and Implementation (PLDI)*, vol. 49, no. 6. ACM, 2014, pp. 419–428.
- [23] V. Raychev, M. Vechev, and A. Krause, “Predicting program properties from Big Code,” in *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, vol. 50, no. 1. ACM, 2015, pp. 111–124.
- [24] V. Raychev, P. Bielik, M. Vechev, and A. Krause, “Learning programs from noisy data,” in *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, vol. 51, no. 1. ACM, 2016, pp. 761–774.
- [25] C. K. Roy and J. R. Cordy, “A survey on software clone detection research,” *Queen’s School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.
- [26] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, “SourcererCC: scaling code clone detection to big-code,” in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 1157–1168.
- [27] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, “The qualitas corpus: A curated collection of java code for empirical studies,” in *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*. IEEE, 2010, pp. 336–345.
- [28] T. Tieleman and G. Hinton, “Lecture 6.5-RMSProp: Divide the gradient by a running average of its recent magnitude,”

*COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.