Learning code summarization from a small and local dataset

Toufique Ahmed
University of California, Davis
Davis, California, USA
tfahmed@ucdavis.edu

Premkumar Devanbu
University of California, Davis
Davis, California, USA
ptdevanbu@ucdavis.edu

ABSTRACT
Foundation models (e.g., CodeBERT, GraphCodeBERT, CodeT5) work well for many software engineering tasks. These models are pre-trained (using self-supervision) with billions of code tokens, and then fine-tuned with hundreds of thousands of labeled examples, typically drawn from many projects. However, software phenomena can be very project-specific. Vocabulary, and other phenomena vary substantially with each project. Thus, training on project-specific data, and testing on the same project, is a promising idea. This hypothesis has to be evaluated carefully, e.g., in a time-series setting, to prevent training-test leakage. We compare several models and training approaches, including same-project training, cross-project training, training a model especially designed to be sample efficient (and thus prima facie well-suited for learning in a limited-sample same-project setting) and a maximalist hybrid approach, fine-tuning first on many projects in many languages and then training on the same-project. We find that the maximalist hybrid setting provides consistent, substantial gains over the state-of-the-art, on many different projects in both Java and Python.

KEYWORDS
deep learning, same-project training, code summarization, transfer learning

1 INTRODUCTION
Machine learning applications in software engineering have been very successful in practice (e.g., Microsoft’s CoPilot) and also on a wide range of more advanced applications [23]. Recently, there has been a great deal of interest in foundation models [1, 9, 11, 17, 34] which subjects a very highly parametrized, high-capacity neural model to a two-phase training regime. The first unsupervised “pre-training” phase is done with an enormous corpus, using a simple fill-in-the-blanks or predict-the-next-token/sentence regime. This phase can be carried out on essentially any (unlabeled) code data harvested from the web. There is no task-specific goal here; the model simply learns the statistics of the input data. The second phase, fine-tuning, trains on-task, and requires carefully curated, consistently labeled data, consisting typically of input-output pairs reflecting good, consistent, on-task performance.

The challenges of creating well-curated, de-duplicated, and yet relevant training datasets have been described by several authors [4, 10]. Recent work by Ahmed et al [3] and Chen et al [7] explore some of the issues that arise with finding sufficient quantities of high-quality find-tuning data. Data availability may be limited because: first, some languages (e.g., Ruby) are relatively less popular than other languages, and available high-quality data may be limited. Second, the projects in a language may be skewed towards one application domain (e.g., Javascript for the web) and thus the performance of the trained model maybe somewhat uneven. Finally, and most interestingly, when curating software engineering datasets, for on-task fine-tuning, yet another strange, unique, wrinkle arises: project specificity.

It’s well known that developers in different projects do behave somewhat differently; they use different terminology, different algorithms, and even different coding practices. As far back as 2009 [37] it was observed that cross-project models don’t perform as well as in-project models on defect prediction tasks. These difficulties cross-over into language models; even the earliest paper on language modeling for code [13] noted application-specific effects. Subsequent work by Tu et al [32] and Hellendoorn [12] noted the highly local, project- and even file-specific vocabularies of source code, and proposed ways to handle them.

This phenomenon offers an entirely new opportunity: Can project-specific training data improve performance? On the plus side, since vocabulary, coding styles etc are notoriously project-specific, training and testing on the same project should give better performance. This seems like an easy, low-hanging fruit. However, there are a couple of traps. First, when working within project, one has to be careful in partitioning training and test data, so that we only use data that would be realistically available in practice. Second, within-project data may be quite substantially limited in comparison to cross-project data. For this reason, within-project training regimes would require models that learn well from fewer samples.

For this reason, we believe that it would be useful to investigate approaches that would improve sample efficiency for the fine-tuning phase of foundation model training. By “improving sample efficiency” we mean the general goal of increasing the ability of machine-learning models to learn to perform better, with fewer training samples. For example, a model A that reliably performs as well as model B with much fewer training samples is a more “sample efficient” model. Sample efficient model A both requires less data and potentially trains much faster: thus saving human effort, time, and energy usage. Most of all, in settings where high
quality training data is not as abundant, model A would be more attractive.

Finally, as a sort of “stress-testing” of the same-project tuning idea, we applied this to an extremely well-tuned code summarization model, to see if same-project training provided *any improvement at all*. For this we used the multilingual “PolyGlot” model, by Ahmed & Devanbu [3]. They found that cross-project, multilingual training, using a very large fine-tuning set in many languages provided best-in-class performance (this “PolyGlot” model was the chart-topper on the CodeXGlue Leaderboard1 for a while, although CodeT5 has since reported better performance). We wondered whether even this extensively well-tuned model could be further improved on a specific project by further fine-tuning on the same project. One might expect that it wouldn’t, since it is already so well trained... but actually, it worked!

In this paper we consider same-project fine-tuning, for the task of code summarization, and make the following contributions

(1) We investigate the benefits of within-project training, using a time-series scenario: we train only on ‘past’ data, and evaluate on ‘future’ data. In the code summarization setting, this reflects a realistic setting where a developer asks for the summary of a piece of code, and we train only data already available that specific time point in the history of the project.

We find that within-project training offers some advantages.

(2) Second, we adapt the GRAPHCodeBERT foundation model specifically to improve its sample efficiency for code summarization; the resulting GCBHybrid model, achieves high levels of sample-efficiency and can outperform the state of the art in some project-specific settings.

(3) We also found that the “maximalist stress test”, adding project-specific fine-tuning to the already extensively fine-tuned “PolyGlot” model, actually provides further benefits, and yields the best performance, comfortably beating the state of the art CodeT5 model overall, with statistical and practical significance (pairwise Wilcoxon test, with a difference in means, over all test samples, of about 3.7 BLEU-4; this is above the 2.0 BLEU-4 threshold difference that humans are experimentally [28] known to notice).

(4) Finally, while “PolyGlot”+ same-project setting is most performant, we do find that same-project training is remarkably efficient; even the largest projects use less than 2.5% of the time taken for cross-project training, while attaining comparable performance to current state-of-the-art

The paper begins in the next section with some motivating explorations of project-specific phenomena relevant to the foundation model setting. Following that we present our methodology, followed by results, discussion, and related work. The paper ends with a brief speculation on future directions.

2 BACKGROUND & MOTIVATION

We begin with a brief overview of Foundation models [5], which are currently widely used in NLP and SE. Foundation models are trained in two stages (i.e., pre-training and fine-tuning). In the pre-training stage, we train the models with billions of unsupervised tokens to teach the models the statistics of the language in a self-supervised way, using simple tasks like auto-regressively predicting the next token, filling in a blank in context, completing the next sentence, denoising an input, etc. These tasks are performed in a multi-layer deep network; the intermediate layers thus learn a representation (“embedding”) of the salient patterns of input token sequences in the code. Later we take the embeddings of the input sequence learned in the pre-training stage and further train it with supervised data in the fine-tuning stage. This pre-training+fine-tuning paradigm was first introduced in NLP by Devlin et al. [8]. They proposed an encoder-only model BERT, pre-trained with two training objectives: Mask Language Modeling (MLM) and Next Sentence Prediction (NSP). MLM is the most effective pre-training objective for encoder-only models where the model randomly masks out a certain percentage of tokens and unmasks them. Liu et al. showed that RoBERTA outperforms BERT using only MLM as a pre-training objective with some new training strategies (e.g., dynamic masking instead of static masking of a sequence) and hyperparameters tuning [22]. BERT-style encoder-only models have inherent limitations for seq2seq generative tasks like Neural Machine Translations (NMT) because of the missing trained decoder. Two models, BART [20] and T5 [27] have well-trained decoders and perform well on seq2seq generative tasks.

These models are not designed for code; subsequent research has refined these models (retaining the same basic scheme) for code and code-related natural language description. CodeBERT [9] and GraphCodeBERT [11] are similar to the BERT model pre-trained with MLM and some code-specific pre-training objectives. PLBART [1] and CodeT5 [34] are replications of BART [20] and T5 [27] specially designed for SE tasks. Code-specific pre-trained models perform quite well on several SE tasks, including code summarization. The standard benchmark dataset CodeSearchNet [15] is used to evaluate these models. The CodeXGlUE includes a de-duplicated code summarization dataset prepared by modifying the CodeSearchNet [15] dataset. CodeXGlUE is a multilingual dataset (consisting of data from six languages) and has between 25K and 252K cross-project training samples for each language. Though some languages have relatively smaller samples (i.e., Ruby and JavaScript), other have very large training datasets (i.e., Java and Python).

Ahmed and Devanbu [3] have recently shown that multilingual training is beneficial for code summarization. Identifiers play a significant role in ML-based summarization, and they are mostly preserved across languages; this phenomenon enables cross-language training to work well. This prior work suggests that if methods from the same projects share similar identifiers, then same-project training can benefit the model. However, there are some issues when using same-project data. First, to be realistic, we can only use data as it becomes available; thus at any point time, only past data in the same project is available; we cannot use data on classes, methods, etc that haven’t been created yet. Thus we perform all our evaluations below in a “time-series” or “time-partitioned” setting. Second, and following from this time-partitioned train-and-test approach, sample sizes get limited. Some times there is no more than a few hundred samples for each project, which differs greatly from the cross-project, cross-language setting where hundreds of thousands of instances can be used to fine-tune the models. If the pre-trained models are sample-efficient, then same-project training

---

1See https://microsoft.github.io/CodeXGLUE/.
can be proven effective for code summarization task. We will briefly
look into the two preliminary, motivating questions (PMQs):

PMQ 1 Do the different samples in the same project share more
identifiers than samples drawn from random projects? If this
were the case, one might hope that training with same project
data would be especially advantageous.

PMQ 2 Are the high capacity pre-trained models sample effici-
ent? If these models were not especially sample-efficient,
then (because same-project data might be as abundant) we
might have difficulty exploiting any same-project data syn-
geries.

PMQ 1: Are identifiers preserved across same-project samples? We con-
jecture that the same-project samples have higher identifier sim-
ilarities because of domain and API usage similarity. Same-project
samples also will use overlapping sets of user-defined class objects and
identifiers. To evaluate this question, we perform a small ex-
periment using time partitioned data. We take five projects from the
Java CodeXGLUE code summarization dataset, each with at least
200 samples and sort them according to their creation date. We
perform the following steps.

(1) Divide the first 200 samples of each project into two groups
(i.e., 1-100 and 101-200).
(2) Take each group and find the unique case-insensitive identi-
fiers of the group. We repeat it for all five projects.
(3) Take group I of project A and calculate the Jaccard index
with group II of the same project.
(4) Now pair group I of project A with group II of all other
projects and calculate the Jaccard index.
(5) Repeat steps 3 and 4 for all projects and observe the Jaccard
Index difference.

Table 2 shows that we get the highest Jaccard indices (always 2-5
times higher than other positions) in the diagonal position where
the data of both groups are coming from the same projects. Hence,
same-project samples have higher identifier similarities.

**Observation 1.** Same-projects samples likely to exhibit more iden-
tifier similarity than the cross-project samples.

PMQ 2: What is the fine-tuning sample efficiency of foundation models?
To observe the sample efficiency of foundations models, we con-
sider two best performing models from each family of models
(i.e., GRAPHCODEBERT from BERT-type encoder-only models and
CodeT5 from seq2seq generative models). We also introduce a hy-
brid model GCBHYBRID in this paper (described in more detail
below), where we cascade the GRAPHCODEBERT encoder with a pre-
trained decoder. For this experiment, we use the java CodeXGLUE
code summarization dataset. Note that CodeT5 is the best perform-
ing model for this task achieving 20.32 BLEU-4, where GRAPHCODE-
BERT reaches 19.22 BLEU-4 We sample datasets of different sizes
(10-300 examples) and observe the cross-project performance of the
three models. Table 1 presents that with 300 code-project samples,
CodeT5 achieves 18.23 BLEU-4 which is only about 2 BLEU-4 lower
than what it performs with the complete dataset of 165k samples.
This, with about 550 times as much data! Therefore, we can con-
clude that models like CodeT5 are fairly sample-efficient and can
perform well with a few data samples. However, CodeT5 struggles
to summarize code well, when less than 150 training samples are
available.

In the same-project fine-tuning scenario, this situation is quite
common, in many projects, as we argue later. Happily, our GCB-
HYBRID model is highly sample efficient, and attains two-digit BLEU-
4 even with ten examples. This is because GCBHYBRID’s pre-trained
decoder is especially trained to generate (denoised) comments. GCB-
HYBRID dominates the CodeT5 until 150 samples become available.

| #of samples | GRAPHCODEBERT | GCBHYBRID | Codet5 |
|-------------|---------------|-----------|--------|
| 10          | 4.88          | 11.37     | 1.38   |
| 50          | 9.29          | 13.7      | 1.84   |
| 100         | 10.02         | 14.73     | 2.32   |
| 150         | 10.33         | 14.98     | 14.93  |
| 200         | 10.57         | 15.51     | 18.64  |
| 250         | 10.73         | 15.63     | 18.81  |
| 300         | 10.58         | 15.71     | 18.23  |

| Complete (~165k) | 19.22 | 19.97 | 20.32 |

**Observation 2.** Pre-trained models can be adapted to be fine-
tuning sample-efficient; such models are competitive with State-
of-the-art for the code summarization task when samples are lim-
ited.

The following sections will discuss same-project training for
code summarization using time series data and observe whether it
can outperform the cross-project training performance with a few
examples.

3 METHODOLOGY
This section briefly describes the dataset preparation and founda-
tion models we used for the evaluation.

3.1 Dataset Preparation
To evaluate the potential of same-project, sample-efficient training,
we prepare a new dataset from CodeXGLUE benchmark dataset.
There are three reasons for choosing CodeXGLUE

(1) The dataset is known to be appropriately de-duplicated, thus
avoiding issues raised in prior work [4, 29]
(2) We can more easily baseline our approach; most foundation
models have been evaluated on this dataset.
(3) This dataset provides the complete path to all the functions
with commit ID and line number. Using this information, we
can find out the creation date of that particular function, and
perform time-series partitioning.

Preparing a same-project dataset, and then partitioning for train-
ing and test, has to be done carefully, to avoid the risk of possible
data leakage from future to past instances during evaluation. There-
fore, we perform time-series partitioning: we sort the samples from
each project according to the creation date and perform an 80:20
split. 80% of data are used for training, and later 20% are randomly

\[ \text{Jaccard Index is calculated as} \ \frac{|X \cap Y|}{|X \cup Y|} \]
As already mentioned, while creating the CodeBERT is one of the first BERT-type encoder–decoder models. The approach we are following to extract the first commit date of a specific line. Note that most of the functions are multi-line, and we use start and end line numbers in the command to get the creation dates for all the lines. We consider the earliest date as the creation date of the complete function. We follow such a strategy because creation dates differ by line. Consider the following code snippet.

```java
public static BeanCopy from(final Object source) {
    BeanCopy beanCopy = new BeanCopy(source);
    beanCopy.isSourceMap = source instanceof Map;
    return beanCopy;
}
```

**Figure 1: Example for assigning creation date**

For the example presented above (Fig. 1), we get the same time-stamp (2015-08-26 12:57:28) for all the lines except the first one (2018-01-13 01:41:10). The approach we are following to extract the creation date has some limitations. "git blame -ignore-rev" reports the earliest commit that changes a specific line. The first line was rewritten/edited from "public static BeanCopy fromMap(Map source) {" to "public static BeanCopy from(final Object source) {" on "2018-01-13 01:41:10", almost 2.5 years later than original function creation date (2015-08-26 12:57:28). We are interested in the original creation date instead of the last commit that changed a single program line. Considering the change made to the first line doesn’t introduce any major change into the program. It does not even introduce or remove any identifier from the code. If we consider the times-tamps of all the function lines, that may help us get the original creation date. We consider the earliest commit time-stamp (2015-08-26 12:57:28) as the function creation date, for example presented in Fig. 1.

There is one case when this approach will fail. If all the program lines have been modified over time at least once, we will fail to predict the actual creation date of the program because "git blame -ignore-rev" will not be able to report the original creation date for any line of the program. However, this is very unlikely to happen. Another challenge is that we can only track down the history recorded on GitHub. Our approach will fail if the programs are created and edited on a local machine and then dumped into GitHub.

Table 2: Intra and inter project identifier overlap

| Projects               | Group I | Group II |
|------------------------|---------|----------|
| oblac/jodd             | 0.16    | 0.06     |
| wildfly/wildfly        | 0.06    | 0.16     |
| orientechnologies/orientdb | 0.07    | 0.17     |
| Unidata/thredds        | 0.07    | 0.05     |
| ngageoint/geopackage-android | 0.05    | 0.05     |

Table 3: Number of projects from each category

| Category | Language | Java | Python |
|----------|---------|------|--------|
| Category I | 10 | 7 |
| Category II | 6 | 7 |
| Category III | 2 | 2 |
| Total     | 18 | 16 |

**3.2 Foundation Models**

In this section, we briefly describe the foundation models that we use to compare the performance of cross-project and same-project training.

**GraphCodeBERT.** CodeBERT is one of the first BERT-type encoder–decoder models specially designed for Software Engineering tasks. CodeBERT pre-trained with two objectives: i) MLM and ii) Replaced Token Detection (RTD). Though CodeBERT is successful...
in many downstream tasks, it does not use any code-related property. Guo et al. [11] describe an encoder-only foundation model, GraphCodeBERT, which uses two additional pre-training objectives (i.e., edge prediction and node alignment) to MLM. The first additional task is to predict code structure edges, and the second aligns representations between source code and code structure. These two objectives incorporate data flow in the pre-training stage, a semantic-level code structure that encodes the relation of “where-the-value-comes-from” between variables. We evaluate GraphCodeBERT in this paper to evaluate the effectiveness of same-project training because GraphCodeBERT outperforms CodeBERT in all downstream tasks, including code summarization. CodeT5 [34] is a unified pre-trained encoder-decoder Transformer model well-suited for the seq2seq generative task. This model is pre-trained with three objectives: i) Masked Span Prediction (MSP), ii) Identifier Tagging (IT), and iii) Masked Identifier Prediction (MIP). CodeT5 learns improved embedding by leveraging the code semantics conveyed from the developer-assigned identifiers. It also achieves the state-of-the-art performance in CodeXGLUE code summarization task. Both GraphCodeBERT and CodeT5 are primarily pre-trained with CodeSearchNet dataset. However, CodeT5 is pre-trained with some additional C and C# datasets.

**GCBHYBRID**

![Figure 2: Steps for preparing GCBHYBRID](image)

In the sample-efficiency experiment, presented earlier (Table 1), GraphCodeBERT reached only 10.58 BLEU-4 even after fine-tuning with 300 samples. Even the current SOTA, CodeT5, underperforms until we fine-tune with at least 200 samples. However, both models do relatively well on the complete Java dataset, reaching 19.22 and 20.32 BLEU-4, respectively. Why do the pre-trained models underperform, with smaller fine-tuning datasets, even after training with billions of unsupervised tokens? GraphCodeBERT does not have a pre-trained decoder; it learns to generate comments only during fine-tuning. Thus, it cannot produce good comments until it’s seen a large number of samples. On the other hand, CodeT5’s pre-training also trains the decoder; however, it’s trained to “refill” masked-out span of code, rather than a complete natural language description. The PLBART model is trained to denoise code and natural language description; yet it failed to outperform GraphCodeBERT, with its trained decoder, on Java code summarization (18.45 BLEU-4). We propose a hybrid model GCBBHYBRID where we cascade the pre-trained GraphCodeBERT model with a specialized decoder pre-trained to denoise natural language description. Such a decoder helps the model to do well on code summarization task by incorporating prior knowledge for generating natural language description.

Like GraphCodeBERT and CodeT5, we use CodeSearchNet dataset for training the decoder. We use only the given training partition of the CodeSearchNet to prevent any data leakage in the fine-tuning stage because our final test dataset is taken from the test partition of CodeSearchNet. We get approximately 2M natural language descriptions to proceed. Following BART, we implement five noising modes. Note that we apply two different types of noise modes to each sample to enhance the dataset. In the next segment of the paper, we will briefly explain the noise modes.

- **Comment permutation** With this noise mode, we take one code-related natural language description/comment at a time and shuffle the tokens in random order.
- **Comment rotation** We randomly choose one token at a time and rotate the comment to bring that token to the first position of the statement. Repairing such noise helps to model to pick the starting of the comments.
- **Token deletion** We randomly choose 15% of the tokens and drop them from the comment. The model’s task is to recover those dropped tokens and generate natural comments.
- **Token masking** Like token deletion, we randomly mask out 15% of the tokens and ask the model to recover them and generate the comment using the decoder. Token masking is a comparatively easier task than token deletion. In token deletion, the model needs to learn both position and content of the missing token.
- **Token infilling** We select a random span with span lengths drawn from a Poisson distribution ($\lambda = 3$). We replace the span with a single token `<mask>`. The model will recover the complete missing span and learn about predicting the number of missing tokens in the span.

| Sequence Type          | Sequence                                             |
|------------------------|-------------------------------------------------------|
| Original               | Return next line with tag masked with whitespace .    |
| Comment permutation     | with line . masked whitespace next tag with Return   |
| Comment rotation        | masked with whitespace . Return next line with tag    |
| Token deletion          | Return line tag masked with whitespace .              |
| Token masking           | Return next `<mask>` with `<mask>` masked with whitespace . |
| Token infilling         | Return `<mask>` tag masked with whitespace .          |

**Table 4: Denoising natural language description**

Table 4 illustrates a comment mutated with the 5 noise modes. For training the decoder, we cascade a RoBERTa encoder with a newly created, 12 layers transformer decoder model. To make this hybrid model work, we need to ensure both encoder and decoder use the same vocabulary. Therefore, we use the original GraphCodeBERT vocabulary for this denoising task. To accelerate the training process, GraphCodeBERT was initialized with CodeBERT, and CodeBERT was loaded with the weights from the natural language.
RoBERTa. We also initialized our encoder with GraphCodeBERT and continued the denoising task for three epochs. Figure 2 depicts the steps involved in preparing GCbHybrid. We start with (a) the pretrained GraphCodeBERT model and (b) untrained 12 layers transformer decoder. These are adjoined (c) together and trained together (d) for the de-noising task described above. After sufficient de-noising performance is achieved, the decoder has become pretty good at generating comments. Now we detach just the decoder (e) and adjoin it with the pre-trained original GraphCodeBERT encoder, to create the (f) GCbHybrid model. We drop the fine-tuned encoder because it is subject to “catastrophic forgetting” of the knowledge learned in the base model [18].

Stress test: the “PolyGlot” Model Finally, as a stress-test of the same-project training approach, we wondered if even a very extensively fine-tuned model such as PolyglotGraphCodeBERT [3], which fine-tuned on enormous (more than 900K) sample dataset, incorporate a diverse sample of project in many languages, could actually benefit from fine-tuning on relatively small number of same-project samples. For this, we took the published PolyglotGraphCodeBERT model, and ran a few epochs of fine-tuning on same project data, and evaluated in on same-project test data. We could do this without fear of training-test data overlap, because the data partitions provided by CodeXGLUE for pre-training and fine-tuning guaranteed that this “PolyGlot” model had not seen these projects during its previous pre-training and fine-tuning.

Complete pipeline and baselines We are primarily investigating performance relative to the cross-project models, that are fine-tuned with hundreds of thousands of instances, with same-project models fine-tuned only a few hundred samples. Figure 3 presents the complete pipeline of our approach. In the pre-processing stage, we separate the same project data using a “segmenter” and convert it to time-series data following the approach described in Section 3.1 in “creation date retriever” stage. After preparing data, we fine-tune four foundation models (i.e., GraphCodeBERT, GCbHybrid, “PolyGlot”, and CodeT5) for the code summarization task and compare them with the performance achieved by those three models in the cross-project setup (where there is no shortage of data).

4 RESULTS

In this section, we evaluate same-project training for the code summarization task, in different settings.

Fine-tuning cross-project baselines & same-project models As mentioned earlier (section 3), we compare our approach with the models fine-tuned with abundant of cross-project data. We need to fine-tune the baseline models because we look for the BLEU-4 of a subset of test data while evaluating same-project data.

The repository of the baseline models (e.g., GraphCodeBERT3 and CodeT5) only provide cumulative (corpus) BLEU-4, which we cannot map into results on the (test) subset within the same project. For GraphCodeBERT, we fine-tune the model with 32 batch size, as recommended by CodeXGLUE repository. However, we fine-tune the CodeT5 model with 24 batch size instead of 48 to fit the model into our Nvidia Titan RTX GPUs. We keep the other parameters unchanged. We also train our proposed GCbHybrid model with the cross-project data for Java (≈ 165k training samples) and Python (≈ 251k training samples). For same-project training, we replace the cross-project samples with same-project data and fine-tune the models using the same codebases used for fine-tuning the baselines.

During fine-tuning, the cross-project models stop improving after 10 epochs, but same-project models continue improving even after 20 epochs, because of the smaller training set. Therefore, we fine-tune the same-project models for 30 epochs. Following all the code relevant foundation models [1, 9, 34], we use smooth BLEU-4 [21] as the evaluation metric.

4.1 Effectiveness of same-project training on Category I projects

Table 5 presents the results of the Category I projects (with 150+ training samples) for Java. We achieve 18.65, 18.83, 19.52, and 19.72 on average with GraphCodeBERT, GCbHybrid, CodeT5 and the “PolyGlot” models on the Java dataset in the cross-project fine-tuning setup. PolyGlot very slightly outperforms the other two models. In the same-project setup, we can see the encoder-only GraphCodeBERT lags, because the untrained decoder has too few samples from which to learn. However, same-project CodeT5 and GCbHybrid perform really well, achieving 22.71 and 22.69 BLEU-4; the “PolyGlot” model excels, refinings it’s already extensive multilingual fine-tuning to reach 25.87 BLEU-4. All models significantly improves over their cross-project counterpart (20.6% for GCbHybrid and 16.2% for CodeT5). Roy et al. [28] reported that less than 2 points do not guarantee systematic improvements in summarization quality and are not trustworthy as proxies of human evaluation. In this category, our best model shows over 6 BLEU-4 improvement over CodeT5 and the “PolyGlot” model, which is 300% the Roy et al. threshold. Hence, same-project training introduces systematic improvement in code summarization task.

Same-project training also works for Category I python projects. As per Table 6, same-project “PolyGlot”, on average, beats the next best prior work (CodeT5) by a solid 5.7 BLEU-4. Note that CodeT5 and GCbHybrid also improve on same-project training, while GraphCodeBERT does not.

4.2 Effectiveness of same-project training on Category II projects

Table 7 presents the results of the Category II projects (with 100-150 training samples) for Java. We measure 17.72, 17.83, 19.18, 18.93 BLEU-4 on average with GraphCodeBERT, GCbHybrid, CodeT5, and “PolyGlot” models on the Java dataset in the cross-project setting. The performance of the cross-project setting is consistent with the results we observe with Category I projects. However, CodeT5 underperforms with same-project training and scores only 5.33 BLEU-4 on average. In section 2, we found that CodeT5 generally performs worse with less than 150 training samples. On the other hand, our decoder-enhanced model GCbHybrid outperforms all the cross-project models and achieves 21.34 BLEU-4, which is 2.16 higher
Learning code summarization from a small and local dataset

GraphCodeBERT GCBhybrid

Pre-trained model

Creation date retriever

Segmenter

Test set from CodeXGLUE

Project name

Same-project data

Time series data

Fine-tuning

Fine-tuned model

Figure 3: Complete pipeline for data generation and model training.

| Projects                  | Number of Samples | Cross-project | Same-project |
|---------------------------|-------------------|---------------|--------------|
| oblac/jodd                | 913               | 17.98         | 17.63        |
| wildfly/wildfly           | 356               | 17.93         | 14.26        |
| orienter/technologies/orientdb | 346         | 16.98         | 20.54        |
| UmdData/threads            | 1341              | 17.66         | 20.71        |
| ngagenet/geo-package-android | 239          | 16.16         | 14.92        |
| RestComm/jain-slee        | 184               | 15.37         | 14.92        |
| OpenEstate/OpenEstate-EO  | 196               | 14.21         | 14.92        |
| tfaces/TfFaces            | 281               | 14.21         | 14.92        |
| jboss/jboss-common-core   | 209               | 15.89         | 14.92        |
| supersmith/lojix          | 336               | 15.85         | 14.92        |
| Average                   | 18.65             | 17.29         | 17.16        |

Table 5: Effectiveness of same-project fine-tuning for code summarization task on category I Java projects

| Projects                  | Number of Samples | Cross-project | Same-project |
|---------------------------|-------------------|---------------|--------------|
| apache/airflow            | 435               | 16.16         | 18.64        |
| tensorflow/probability    | 425               | 16.41         | 20.12        |
| h2oai/h2o3                | 215               | 16.11         | 20.12        |
| Qiskit/qiskit-terra       | 376               | 16.29         | 21.24        |
| chaos/ghreiolab-perceval  | 188               | 15.03         | 14.92        |
| PyCQA/pylint              | 271               | 23.33         | 20.12        |
| SmokinCaterpillar/pypet   | 277               | 19.91         | 18.36        |
| Average                   | 17.50             | 18.59         | 20.22        |

Table 6: Effectiveness of same-project fine-tuning for code summarization task on category I Python projects

| Projects                  | Number of Samples | Cross-project | Same-project |
|---------------------------|-------------------|---------------|--------------|
| oblac/jodd                | 913               | 19.12         | 18.64        |
| wildfly/wildfly           | 356               | 19.12         | 20.12        |
| orienter/technologies/orientdb | 346         | 16.98         | 20.54        |
| UmdData/threads            | 1341              | 17.66         | 20.71        |
| ngagenet/geo-package-android | 239          | 16.16         | 14.92        |
| RestComm/jain-slee        | 184               | 15.37         | 14.92        |
| OpenEstate/OpenEstate-EO  | 196               | 14.21         | 14.92        |
| tfaces/TfFaces            | 281               | 14.21         | 14.92        |
| jboss/jboss-common-core   | 209               | 15.89         | 14.92        |
| supersmith/lojix          | 336               | 15.85         | 14.92        |
| Average                   | 18.65             | 17.29         | 17.16        |

than the best performing cross-models. The “PolyGlot” model surpasses even this model, reaching 23.39 BLEU, over 4.2 BLEU-4 better than the best conventionally fine-tuned (cross-project) model. We find similar performance with Python also (Table 8)

4.3 Effectiveness of same-project training on Category III projects

Foundation models are pre-trained with billions of tokens. However, they are not trained to do code summarization task. They need enough samples to fine-tune the objective of the models. Though GCBHYBRID scored 11.37 BLEU-4 (Table 1) with only 10 samples, it is much lower than we usually achieve with Cross-project models. Therefore, same-project training has some limitations. It requires a certain number of samples to achieve comparable results to cross-project models. We found that the models need at least 100 samples to compete with cross-project models from Category II and Category III projects. Table 9 has 4 projects in total (the first two from

Observation 4. With 100-150 same-projects samples, GCBHYBRID outperforms all the cross-project models. Again, “PolyGlot” does best overall. However, CodeT5 underperforms in this sample-range.
Table 7: Effectiveness of same-project fine-tuning for code summarization task on category II Java projects

| Projects                  | Number of Samples | Cross-project | Same-project |
|---------------------------|-------------------|---------------|--------------|
|                           | Training          | Validation    | Test         | GraphCodeBERT | GCBhybrid | CodeT5 | GraphCodeBERT | GCBhybrid | CodeT5 | PolyGlot | GraphCodeBERT | GCBhybrid | CodeT5 | PolyGlot | GraphCodeBERT | GCBhybrid | CodeT5 | PolyGlot | GraphCodeBERT | GCBhybrid | CodeT5 |
| nic30/hwt                 | 124               | 15            | 16           | 9.64         | 15.81       | 14.18   | 16.93       | 5.79     | 12.8   | 4.36     | 17.05       |
| vaexio/vaex               | 124               | 15            | 16           | 15.9         | 17.23       | 15.62   | 17.08       | 9.2      | 13.6   | 2.2      | 18.19       |
| assemblerflow/flowcraft   | 113               | 14            | 15           | 14.01        | 14.03       | 14.39   | 18.56       | 9.04     | 14.24  | 1.1      | 25.42       |
| funlry5/PyFunccle         | 104               | 13            | 13           | 16.65        | 25.82       | 22.97   | 27.05       | 19.11    | 31.06  | 5.67     | 38.65       |
| pycs/pypropsal            | 100               | 13            | 13           | 19.81        | 27.47       | 23.69   | 23.1        | 15.22    | 24.87  | 17.24    | 25.4        |
| LionelAuroux/pyser        | 102               | 11            | 12           | 16.33        | 16.91       | 17.53   | 13.75       | 7.23     | 13.98  | 3.22     | 20.79       |
| OpenKMIP/PkKMIP           | 150               | 18            | 18           | 14.57        | 15.66       | 17.03   | 18.8        | 31.38    | 39.11  | 41.17    | 42.59       |
| **Average**               | **15.27**         | **18.99**     | **17.92**    | **19.32**    | **13.85**   | **21.38**| **10.71**   | **26.87**|

Table 8: Effectiveness of same-project fine-tuning for code summarization task on category II Python projects

| Projects                  | Number of Samples | Cross-project | Same-project |
|---------------------------|-------------------|---------------|--------------|
|                           | Training          | Validation    | Test         | GraphCodeBERT | GCBhybrid | CodeT5 | GraphCodeBERT | GCBhybrid | CodeT5 | PolyGlot | GraphCodeBERT | GCBhybrid | CodeT5 | PolyGlot | GraphCodeBERT | GCBhybrid | CodeT5 | PolyGlot | GraphCodeBERT | GCBhybrid | CodeT5 |
| Nic30/hwt                 | 124               | 15            | 16           | 9.64         | 15.81       | 14.18   | 16.93       | 5.79     | 12.8   | 4.36     | 17.05       |
| vaexio/vaex               | 124               | 15            | 16           | 15.9         | 17.23       | 15.62   | 17.08       | 9.2      | 13.6   | 2.2      | 18.19       |
| assemblerflow/flowcraft   | 113               | 14            | 15           | 14.01        | 14.03       | 14.39   | 18.56       | 9.04     | 14.24  | 1.1      | 25.42       |
| funlry5/PyFunccle         | 104               | 13            | 13           | 16.65        | 25.82       | 22.97   | 27.05       | 19.11    | 31.06  | 5.67     | 38.65       |
| pycs/pypropsal            | 100               | 13            | 13           | 19.81        | 27.47       | 23.69   | 23.1        | 15.22    | 24.87  | 17.24    | 25.4        |
| LionelAuroux/pyser        | 102               | 11            | 12           | 16.33        | 16.91       | 17.53   | 13.75       | 7.23     | 13.98  | 3.22     | 20.79       |
| OpenKMIP/PkKMIP           | 150               | 18            | 18           | 14.57        | 15.66       | 17.03   | 18.8        | 31.38    | 39.11  | 41.17    | 42.59       |
| **Average**               | **15.27**         | **18.99**     | **17.92**    | **19.32**    | **13.85**   | **21.38**| **10.71**   | **26.87**|

Observation 5. Same-project fine-tuning does help “PolyGlot” model dominate again; however, by itself it provides only modest gains with less than 100 samples, and may have to be used together with other refinements.

5 DISCUSSION

In this section, we will discuss the feasibility of applying same-project training and the computational cost needed for such training. We will also present one motivating example at the end of this section.

5.1 Feasibility of same-project training

So far, we have discussed the possible benefits of same-project training; higher BLEU-4 scores can be attained with just 100 samples. Our experiments suggest that an already extensively fine-tuned model like PolyGlot can still benefit from a few epochs same-project fine-tuning.

However, how long must we wait, after a project starts, to get 100 samples? This matters, because if it takes a long time to get enough samples, the benefit of same-project training is perhaps lower; one might just use cross-project training from older/existing projects. We look into the project-wise lifespans, and time it takes to net 100 fine-tuning samples for our Category I and II projects. We track this data for both Python and Java, for each project in the CodeSearchNet data, from project inception to the date the data was collected. Many of these projects are still active, and the lifespans in Table 4 show the distribution of the project time spans, in days, for both Java and Python projects. We show both the Complete lifespan of the project (until last activity, or the current time, if still active and the time elapsed until 100 fine-tuning samples are available. It’s evident that Java projects “live” longer than Python projects in our dataset. The median lifespan for Java projects is 2872 days (almost 8 years); however, the time required to generate 100 fine-tuning samples is 335 days (less than 1 year). That means for nearly 7 years (85% of the total lifespan), these projects could benefit from same-project fine-tuning. We have observed a similar situation for Python also. The median lifespan for Python projects is 1365 days (almost 4 years), and the time required to generate 100 comments is 496 days (less than 1.5 years). Python projects could also benefit
from same-project training for 64% of their lifespan. Therefore, we can assume that the sample sizes sufficient for same-project training become available reasonably early and can be used for the remaining lifespan of the project.

### 5.2 Computational cost of same-project training

Same-project training is computationally much cheaper! In fact, in our experiments same-project training run is persisted for three times as many epochs as the cross-project run; we do get this better convergence with fewer samples; even so, we see significant gains in fine-tuning costs. In the cross-project setting, we have 164,923 and 251,820 samples for Java and Python, respectively. To compare with the same-project setting, we choose the projects with the largest in-project fine-tuning datasets: 1341 for Java, and 435 for Python. Even these big projects, same-project training takes just 2.43% (Java) and 0.52% (Python) of the time of cross-project training.

Considering all the Java and Python Category I and II projects, the total training samples will be 5,122 and 3,004, across all projects. Cumulatively, for all projects, same-project training takes just 9.31% and 3.57% of the cross-project training time for Java and Python, respectively. Note that we compare the computational efficiency with respect to sample counts instead of time because the time required for training a certain number of samples varies due to server load.

### 5.3 Motivating Example

Now we present one illustrative example where our same-project polyglot GraphCodeBERT model outperforms all the cross-project and same-project models. Table 10 presents the results from different models for the examples presented in Figure 5. We can see cross-project GraphCodeBERT, GCBHybrid, and polyglot GraphCodeBERT produce meaningful summaries for example I. Same-project GCBHybrid also generates a complete sentence (0.36 BLEU-4). However, our polyglot GraphCodeBERT fine-tuned with the same project data gives the closest summary achieving 0.49 BLEU-4.

### 6 RELATED WORK

**Code summarization** Code summarization is a widely studied problem in software engineering. Developers spend around 59% of their time on activities somewhat relevant to program comprehension [35], and good comments can ease the development and maintenance process by helping developers more quickly understand the meaning of code under maintenance [30]. However, misaligned and outdated comments are prevalent in SE projects. Automatic code summarization can help provide more faithful & current comments. Code summarization can also help write new comments.

We can closely relate the code summarization task to Neural Machine Translation (NMT) (e.g., translating English to German). In NMT, an encoder-decoder-based framework is used to do the translation task. Researchers in the SE domain have also adopted such a framework for code summarization tasks. Systems like CodeNN [16], DeepCom [14], Astattg [19], Rencos [36], NCS [2] and many more applied different kinds of deep learning architecture (e.g., LSTM [31] and Transformers [33]) on encoder-decoder framework and show good performance on code summarization task. Prior work [10, 28, 29] discuss the evaluation metrics and datasets that have been used for code summarization task. **Foundation models for code summarization** Foundation models [1, 9, 22, 24–26, 34] are currently state-of-the-art for the code summarization task. Pre-training is key for foundation models, and helps them learn the language’s statistical properties well. Since the model already knows about the language, a few examples are enough to train the model for a downstream task like code summarization. In this paper, we show that the pre-trained models are indeed sample-efficient and can outperform the models trained with cross-project data. Note that there are more than 30 papers that have been published in the last five years that follow some form of encoder-decoder architecture for code summarization [28].

Table 9: Effectiveness of same-project fine-tuning for code summarization task on category III projects

| Projects                  | Number of Samples | Cross-project | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlot | GraphCodeBERT | GCBHybrid | Code/TS | PolyGlo
We explore some of them below.

The main threats to our results arise from our evaluation approach. We discuss augmenting the dataset using multilingual training and help models perform better. In contrast, we propose to reduce the sample count and perform well using same-project data. Auto-regressive generative language models, such as GPT-3 [6] have shown strong on-task performance, even after very limited fine-tuning; however, in our setting, without custom pre-training, as was done for GraphCodeBERT, and GCBHybrid, it’s difficult to ensure that the (enormously sized) pre-training data used in these models was not already pre-polluted with the data we use for same-project testing; these enormously sized models are too costly to pre-train, except for the wealthiest organizations, so we omit these from our evaluation.

## 7 Threats

The main threats to our results arise from our evaluation approach. We explore some of them below.

### Data Pollution

For external validity, and stability of results, it is important ensure that we never test on data we used for pre-training or fine-tuning. The CodeXglue dataset for code summarization is split very carefully to avoid risk of data pollution; the pre-training data is separate from the fine-tuning data, and the test data is distinct from both of these. Our evaluation of the GraphCodeBERT and GCBHybrid models adheres to this protocol. The “polyglot” model is first fine-tuned on a large, multilingual dataset, of around a million samples, from CodeXGlue, before project-specific fine-tuning and same-project evaluation, on held-out set of projects.

### Data Duplication

As described by Allamanis [4], duplication lead to poor estimates of performance that don’t generalize. Fortunately CodeXglue [23] is very carefully de-duplicated, and thus the performance numbers we report here can be expected to be fairly good.

### External Validity

External validity threats may arise from size of samples used to estimate performance, as well as whether the representativeness of the samples. First, our results have statistical significance: we have compare the performance of our best model (“polyglot” model) with the SOTA (CodeT5) using a paired, non-parametric 2-sample test, and can reject the null hypothesis that SOTA is the same or better than our best model. Second, our average improvement on the BLEU-4 score is well above the 2 BLEU-4 threshold reported to be the barrier for humans to detect.

On the other hand, we have tested on a total of 18 Java projects and 16 Python projects. In almost every setting our best model beats CodeT5; but in some cases, it does not. Therefore, some caution is warranted in assessing the external validity of our results.

## 8 Conclusion

The existence, and impact, of Project-specific phenomena in software projects has been known for quite a while. The advent of foundation models, which can be fine-tuned on-task, offers a possible direction to exploit project-specificity for better on-task performance. We explore this direction, for the code summarization task, with several popular foundation models. We find that same-project training helps models exhibit competitive performance in several settings. In particular, we develop a new kind of GraphCodeBERT model, named GCBHybrid, which combines GraphCodeBERT with a specially trained decoder. GCBHybrid exhibits very high sample-efficiency, which further enables exploitation of project specificity; except “polyglot”, GCBHybrid does achieve state-of-the-art in some realistic same-project time-series settings. We also find that same-project training offers substantial savings in computational cost. In addition to code summarization, project-specific fine-tuning is a general idea that could well prove an useful adjunct for other tasks, such as defect prediction, fault localization, de-obfuscation, or automated patching. Finally, the same-project code summarization dataset and GCBHybrid source code are made available anonymously at https://doi.org/10.5281/zenodo.6523229.

| Original | Cross-project or same-project? | Model name | Prediction | BLEU-4 |
|----------|--------------------------------|------------|------------|--------|
| Instantiates a shutdown hook of the given class and sets its name. | Cross-project | GraphCodeBERT | Initialize the shutdown hook. | 0.11 |
| | | GCBHybrid | Initialize the shutdown hook. | 0.11 |
| | | CodeT5 | This method is called at the beginning of the action. | 0.13 |
| | | PolyGlot | Parses a shutdown hook. | 0.14 |
| Same-project | | GraphCodeBERT | Get the the the. | 0.07 |
| | | GCBHybrid | Instantiate a hook of the given class. | 0.36 |
| | | CodeT5 | } addInfo ( "About to instantiate shutdown hook of type [" + ... more random tokens | 0.02 |
| | | PolyGlot | Instantiates a new shutdown hook of the given class. | 0.49 |

Table 10: Predictions for the example presented in Fig. 5
