# An Empirical Investigation on the Performance of Domain Adaptation for T5 Code Completion

Daisuke Fukumoto[*], Yutaro Kashiwa[*], Toshiki Hirao[*], Kenji Fujiwara[†], Hajimu Iida[*]

[*]*Nara Institute of Science and Technology, Japan* — [†]*Tokyo City University, Japan*

***Abstract*—Code completion has the benefit of improving coding speed and reducing the chance of inducing bugs. In recent years, DL-based code completion techniques have been proposed. In particular, pre-trained models have shown outstanding performance because they can complete code by considering the context before and after it is completed. While the model can generate the set of candidate codes, some of those might need to be modified by developers because projects can have different coding rules.**

**In this study, to complete code that fits a specific project appropriately, we train the CodeT5 model with additional data from the target project. This fine-tuning approach is called domain adaptation, and is often used in neural machine translation. Our preliminary experiment observes that our domain-adapted model improves 5.3% of the perfect prediction rate and, 3.4% of the edit distance rate, compared to the fine-tuned model with the out-of-domain dataset. Furthermore, we discover that the improvement is greater with a larger repository size. The model that is trained with a small dataset, however, hardly improves or performs worse.**

***Index Terms*—code completion, CodeT5, domain adaptation, fine-tuning, transfer-learning**

## I. Introduction

Code completion is an integral function of modern Integrated Development Environments (IDEs) [1]. It helps developers speed up their coding work by suggesting what developers might want to enter next, such as class names, methods, or any variables in the valid scope. Because of the effectiveness, code completion has been widely studied over the decades [2]–[6]. Notably, DL-based unsupervised language models [7]–[9] show outstanding performance for code completion [5], [10]. Although these have the disadvantage of requiring large datasets [11], in recent years, a large quantity of source code has been made publicly available on code hosting systems such as GitHub, overcoming this disadvantage and accelerating the recent use of unsupervised language models.

In particular, many studies employ the pre-trained models extending the T5 [9] or BERT models [8], and they demonstrate state-of-the-art performance in code-related tasks [12]–[14]. These models fit code completion [5] because they can consider the context of the code both before and after the code that expects to be completed. It can take into account not only the code that developers are typing at the time but also the code before and after the target completion point.

Still, those pre-trained models do not always show the high performance due to the lack of awareness for project-specific context. For example, even for a function name of the implementation validates whether an argument is null or not null, there are often several types of method names used: "`notNull`", "`checkNotNull`", "`isNotNull`", "`ensureNotNull`", and "`parametersNotNull`". The pre-trained models are built with general datasets (i.e., out-of-domain datasets), thus, it might not be able to select the most appropriate candidate that fits a specific project.

This study tries to apply domain adaptation techniques [15] to address this challenge. Domain adaptation is a technique to develop a suitable model for a specific domain, by fine-tuning a pre-trained model with additional data from a specific domain (e.g., a project). This technique has the potential to complete code optimized for a specific project, understanding their coding rules. Still, it is not clear if the technique can be used for code completion so we formulate the first research question as follows:

RQ1. *Can domain adaptation improve the performance of code completion, and if so to what extent?*

In addition, the software engineering community has another specific challenge to apply domain adaptation. There are a lot of small projects and it is not certain if such a small amount of fine-tuning data can contribute to performance improvement. If not, it wastes the time and machine-resource for fine-tuning. Also, on the other hand, if fine-tuning with a large fine-tuning dataset shows similar performance to that with a small dataset, this would also be a waste of resources. Hence we formulate the second research question as follows:

RQ2. *What is the impact of different dataset sizes on the fine-tuned transformer's performance?*

To answer these two research questions, we conduct an empirical study that evaluates the impact of domain adaptation, using a state-of-the-art T5 model built with different data resources and data-sizes. Throughout our experiments, our work makes the following contributions:

1) An empirical result showing that domain adaptation can improve code completion performance.
2) An empirical evidence showing that fine-tuning with out-of-domain datasets does not improve the performance.
3) An empirical fact that domain adaptation with a larger dataset improves the performance more greatly but that with a small dataset hardly improves or performs worse.
4) A replication package including the studied datasets and the program for fine-tuning is available on https://github.com/sdlab-naist/DomainAdaptedCodeT5.

## II. Related Work

This section introduces prior work that has studied code completion models, and the techniques used in this study.

### *A. Deep Learning Based Code Completion*

In recent years, many studies have proposed code completion models using deep learning [3], [4]. Kim *et al.* [3] have developed transformer models for code completion and they observe that the models outperform RNN-based systems [16]. Karampatsis and Sutton [4] tackled a major problem where new vocabularies tend to occur frequently as new identifier names are created in code, which causes the difficulty for code completion. Thus, an open-vocabulary neural language model was proposed by employing machine translation techniques.

These approaches are single-token-based and would not be sufficient support for developers in actual development scenarios such as completing a function name with its arguments. To address this challenge, prior work has proposed multiple token-based approaches. Matteo *et al.* [5] have proposed an approach that targets the completion of multiple tokens using T5 [9] and BERT [8]. They compared those models and found that T5 performs better code completion. Alexey *et al.* [10] have also developed a multiple tokens code completion model using GPT-2 [7] and show that the model outperforms n-gram language model approaches.

However, they might not be able to suggest the appropriate candidate that fits the code of those projects due to the lack of awareness for project-specific context (e.g., coding rules).

### *B. Domain Adaptation for Code Completion*

Domain adaptation is a technique to fill the data distribution gap between training and testing datasets, which also overcomes the problem where there is no sufficient data in specific domains [15]. This technique is a subcategory of transfer learning and is often used in Deep learning studies such as neural machine translation (NMT). This technique has two steps; training on out-of-domain datasets, and then fine-tuning the model on a scarce in-domain dataset. The second step allows the model to bias its list of candidates toward the terms in the target domain, which can improve the accuracy.

In software engineering, several studies train models with data from similar projects and have reported a certain improvement. Matteo *et al.* [5] compare the code completion model using Android projects data with the model using Java projects data, and show that the former model outperforms the other model. Furthermore, Sharma *et al.* [6] report that a model trained on UNITY3D projects data can complete code more accurately than a model trained on C# projects data.

These studies use finer-grained data (i.e., one-platform data) but still, they are not the finest-grained. Even in a platform (e.g., Android, UNITY3D), each project has different coding rules so the model is not optimized for a specific project and not be able to complete a project's specific code. Whereas, our study explores the possibility to apply the finest-grained level of domain adaptation to create an optimized model for a project and evaluate its performance.

## III. Study Design

This section describes the data creation and training process for code completion models.

### *A. Overview*

This study fine-tunes CodeT5 [13], which is a pre-trained T5 model using multilingual programming languages and shows a state-of-the-art performance. To investigate the performance of domain adaptation, this study uses different resources to create three types of CodeT5 models; CodeT5, CodeT5+, and Domain-Adapted CodeT5 (DA-CodeT5). Note that, DA-CodeT5 is fine-tuned with four different sizes of datasets (collected from four data points of a project) in order to investigate what amount of data the domain adaptation requires (i.e., RQ2).

This study trains the CodeT5 with a dataset that contains a large amount of Java methods (i.e., CodeSearchNet). To create the Domain-Adapted CodeT5 (DA-CodeT5), CodeT5 is fine-tuned using data from a specific Java project. Specifically, the data was collected from the Spring Framework project because it is a well-organized project (e.g., it has their own coding rule[1]) and has a large amount of Java files.

However, comparing CodeT5 with DA-CodeT5 is not a fair comparison because DA-CodeT5 has been fine-tuned with an additional training dataset. To eliminate the threat of the data size unfairness, we prepare another model (CodeT5+), whose training dataset is at the same size of DA-CodeT5. Note that, to expand the dataset of CodeT5+, an additional subset is randomly selected from the GitHub repositories with specific conditions, which is described in the next subsection.

### *B. Dataset Creation*

Our study creates six datasets that are made from different resources or are different size. Table I summarize the dataset used in this study.

**Data collection:** We collect methods from different resources in order to fine-tune CodeT5, CodeT5+, and DA-CodeT5. For CodeT5, we extract methods from CodeSearchNet [17] that is a platform to provide source codes for scientific research. The dataset for CodeT5+ is made from GitHub repositories. To select appropriate projects, we use the GitHub search platform [18], which can specify non-forked Java projects, as well as more than 100 commits, 100 stars, and 1 fork.

As for the dataset for DA-CodeT5, we collected methods from four data points of the Spring Framework repository. The first data point is the latest revision of the Spring Framework repository on October 2022. The dataset in this point is used in both RQ1 and RQ2. The second, third, and fourth points are the revisions when the number of the methods is 50%, 10%, and 1% compared with that of the latest revision (i.e., 100%), respectively.[2]

[1]https://github.com/spring-projects/spring-framework/wiki/Code-Style

[2]There are no revisions that have the exact number of methods of 50%, 10%, and 1%. So we used the revisions that have the closest numbers.

TABLE I
SUMMARY OF THE STUDIED MODELS AND DATASETS.

| Model (Data source) | Size | Dataset | #Methods | #Lines |
|---|---|---|---|---|
| CodeT5 (CodeSearchNet) | | training | 400448 | 1222431 |
| | | evaluation | 50056 | 153630 |
| CodeT5+ (GitHub repositories) | | training | 7805 | 43203 |
| | | evaluation | 975 | 5460 |
| DA-CodeT5 (Spring Framework) | 100% | training | 7805 | 40863 |
| | | evaluation | 975 | 5072 |
| | | test | 975 | 5458 |
| | 50% | training | 3705 | 19723 |
| | | evaluation | 463 | 2413 |
| | | test | 463 | 2416 |
| | 10% | training | 1332 | 7253 |
| | | evaluation | 166 | 904 |
| | | test | 166 | 898 |
| | 1% | training | 120 | 701 |
| | | evaluation | 14 | 68 |
| | | test | 14 | 93 |

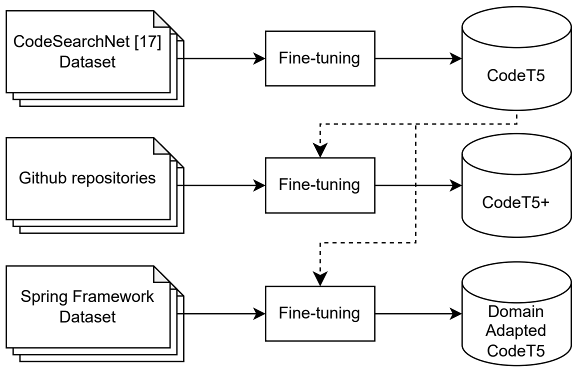

Fig. 1. Overview of generating three different models. CodeT5 was fine-tuned with CodeSearchNet dataset, CodeT5+ was fine-tuned with dataset made from GitHub repositories, and DA-CodeT5 was fine-tuned with the Spring Framework dataset.

**Data processing:** Then, following a previous study [5], we conduct four types of filtering. First, we remove methods that contain '*test*' or '*toString*', methods that have fewer than 3 lines, or duplicated methods. Also, we eliminate methods that have more than 100 tokens because variations in the length of the dataset can affect the accuracy of the model.

Next, we randomly extract methods from these repositories so that the number of the methods in CodeT5+ is the same as that in DA-CodeT5 (100%).

Once six datasets (i.e., different resources and sizes) are created, the collected methods in each dataset are divided into three groups: training, evaluation, and testing. The ratio between the numbers of methods in these datasets is 8:1:1. Finally, in each dataset, $n$ tokens in each line of each method are replaced with masked tokens (i.e., "<MASK>"), where $n$ is randomly chosen and ranges from 1 to $min(10, line\ length)$.

## C. Fine-tuning

Figure 1 illustrates the overview of the fine-tuning process. First, we fine-tune the CodeT5 with CodeSearchNet dataset. We employ the pre-trained model and tokenizer provided by the authors of CodeT5 [13]. Specifically, the authors provide several version of CodeT5 and we use the small version of CodeT5[3] because the calculation cost is very high as the previous study also mentioned [5]. The batch size is 1024 and the learning rate is $1e^{-4}$ as a constant. The hyper-parameters should be optimized for each dataset in the fine-tuning phase, but we do not optimize them because it is very computationally expensive. For each epoch, the evaluation dataset is predicted with fine-tuning model and the perfect match is recorded. If the perfect match does not update the maximum value for four consecutive epochs, it is considered over-learning and stops the learning (i.e., early stops).

The fine-tuning of CodeT5 for 50 epochs eventually took about 138 hours on a Linux server with two NVIDIA A100 GPUs. Furthermore, we fine-tuned CodeT5 with the additional source from the GitHub repositories and Spring Framework in order to create CodeT5+ and DA-CodeT5, respectively. These processes consumed 2 hours for each DA-CodeT5 and CodeT5+ (50 epochs at maximum) on a Linux server with two NVIDIA A100 GPUs.

## D. Model Evaluation

To evaluate those three models, we employed three quantitative and qualitative performance measures used in the previous study [5], which are Perfect Prediction Rate, Edit Distance, and BLEU-n.

**Perfect Prediction Rate (PPR)** is the rate indicating what percent of predicted completions are perfectly matched with the code written by actual developers [5]. This measure is very strict because the predicted completion is treated as wrong even if one token (i.e., even one character) is different from the code written by actual developers. If lines that have fully-masked tokens (e.g., 10 tokens) account for the larger part of the dataset, a lower perfect prediction rate may be achieved.

**Edit Distance** is a measure of the similarity between two code, so-called Levenshtein Distance [19]. The distance represents the minimum number of times when each token of a code is removed, added, or replaced until it becomes the same as the other code. We normalize the distance which is divided by the length of the longer code. A shorter edit distance indicates a better performance by the model.

**Bilingual Evaluation Understudy (BLEU-n)** is a measure of the quality of neural machine translation and is used in many software engineering studies [5], [12], [20]. BLEU-n is calculated based on what percent of n-gramed tokens of the generated code match the original code. BLEU-n ranges from 0 to 1, where 1 means that the generated code and the original code are perfectly matched. If the translated sentences are smaller than n-gramed tokens, the proportion cannot be calculated so we exclude them from calculation. Similar to the previous studies [5], this study employs BLEU-1, BLEU-2, BLEU-3, and BLEU-4 to compare models.

[3]https://huggingface.co/Salesforce/codet5-small

TABLE II
CODE COMPLETION PERFORMANCE BY THE GENERATED MODELS

| Model | PPR | Edit distance | BLEU-n n=1 | n=2 | n=3 | n=4 |
|---|---|---|---|---|---|---|
| CodeT5 | 0.572 | 0.234 | 0.759 | 0.656 | 0.582 | 0.510 |
| CodeT5+ | 0.577 | 0.230 | 0.763 | 0.663 | 0.594 | 0.528 |
| DA-CodeT5 | **0.630** | **0.196** | **0.800** | **0.707** | **0.644** | **0.585** |

* A higher PPR and BLEU-n show a better performance whereas a shorter edit distance shows a better performance

TABLE III
IMPROVEMENT BY EACH DA-CODET5 SIZE IN COMPARISON TO CODET5.

| Model | PPR | Edit distance | BLEU-n n=1 | n=2 | n=3 | n=4 |
|---|---|---|---|---|---|---|
| 100% | 0.058 | **0.038** | **0.040** | **0.052** | **0.062** | 0.075 |
| 50% | **0.065** | 0.033 | 0.035 | 0.043 | 0.059 | **0.076** |
| 10% | 0.037 | 0.020 | 0.020 | 0.030 | 0.034 | 0.032 |
| 1% | 0.000 | 0.004 | 0.003 | 0.010 | -0.009 | -0.014 |

* A higher values show a better performance of DA-CodeT5 than CodeT5

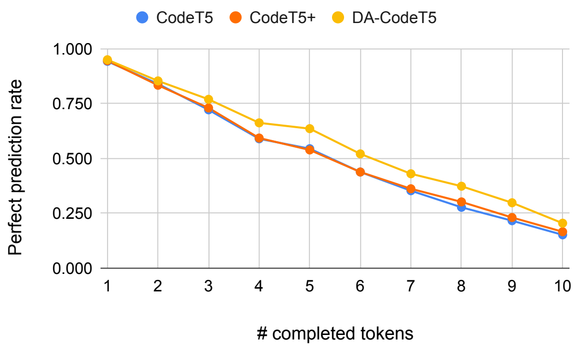

Fig. 2. Perfect prediction rate across each number of completion tokens

## IV. RESULTS

*RQ1 Can domain adaptation improve the performance of code completion, and if so to what extent?*

Using the testing dataset of the Spring Framework, Table II displays the perfect prediction rate, the edit distance rate, and the BLEU-n for three models. The completed tokens produced by the baseline model (i.e., CodeT5) had the worst performance for the perfect prediction rate (i.e., 0.572), which was 0.5% lower than CodeT5+ (i.e., 0.577). Similarly, the CodeT5 basic model's edit distance from the original developers' code was the highest (i.e., the worst performance). CodeT5's performance (i.e., 0.234) was 0.4% worse than CodeT5+ (i.e., 0.230). Also for BLEU-n, CodeT5+ performs slightly better than CodeT5 (0.4%–1.8%). Overall, the performance difference between these two models was not significant even though the fine-tuning process took 2 hours.

**Finding 1:** Fine-tuning with an out-of-domain dataset does not improve the performance of the CodeT5 model.

In comparison to the CodeT5+ model, the DA-CodeT5 performed 5.3% better in terms of the perfect prediction rate. Additionally, the DA-CodeT5 performed about 3.4% better than CodeT5+ in terms of the edit distance. The DA-CodeT5 shows consistently better performance for all the BLEU-n (3.7% to 5.7% higher performance).

We also looked at performance according to the amount of completed tokens because performance could differ depending on how many completed tokens there are. For each amount of completed tokens, the perfect prediction rate is displayed in Figure 2. The perfect prediction rate tends to decline with increasing token count for all models. In contrast to CodeT5 and CodeT5+, DA-CodeT5 shows higher or equivalent performance, regardless of the completed token size.

Surprisingly, we discovered that DA-CodeT5 correctly adheres to their coding rules when looking at the perfect match samples. A coding rule for the Spring Framework, for instance, states that the attribute variable must always be accessed with "`this.`"[4] However, CodeT5 and CodeT5+ violate this rule and complete "`(Component pathComponent : pathComponents)`{". In fact, only DA-CodeT5 correctly completed, which is "`(Component pathComponent :`**`this`**`.pathComponents)`{". Such perfect completions will save the time needed by revisions in code reviews [21].

**Finding 2:** The domain-adapted model (DA-CodeT5) achieved a 5.3% higher perfect prediction rate and 3.4% shorter edit distance than the other two models.

*RQ2 What is the impact of different dataset sizes on the fine-tuned transformer's performance?*

Table III compares the four DA-CodeT5 models to the CodeT5 that was used in RQ1 and demonstrates the improvement made by the four models using different size datasets. Note that, we show the relative performance (i.e., the improvement from CodeT5) because we are more interested in whether it is worthwhile to use smaller datasets to apply the domain adaptation to CodeT5. Also, we cannot simply compare the absolute performance of each DA-CodeT5 model (i.e., 100%, 50%, 10%, and 1%) because we collected the data from different four data points of the Spring Framework (i.e., the testing datasets are different).

Except for PPR and BLEU-4, 100% DA-CodeT5 had the best performance, and there was a tendency for the improvement to be greater as repository size increases. However, the magnitudes of the 100% and 50% models' improvements are equivalent, implying that the degree of domain adaptation's improvement may be constrained as dataset sizes increase.

It is also worth noting that the 1% model hardly improves most of the performance measures, or rather worsens BLEU-3 and BLEU-4. The 10% model, however, improves the performance by 2.0% to 3.7%. This demonstrates how identifying the tipping point is crucial for domain adaptation.

**Finding 3:** Domain adaptation with the full-size dataset shows the best performance whereas that with the smallest dataset does not improve the performance.

[4]https://github.com/spring-projects/spring-framework/wiki/Code-Style#field-and-method-references

## V. Discussion

### A. Future Research Direction

Finding 3 shows that DA-CodeT5 fine-tuned with the largest dataset (100%) shows the largest improvement over CodeT5. More specifically, the improvement increases with dataset size. However, as Finding 1 shows, there should be a limitation where the performance can be improved. As fine-tuning with a very large dataset will be computationally expensive, the cost may not be worth it for a small performance improvement. Thus, our future study plans to predict the point where the performance is achieved at the limitation, which helps model creators to save time for fine-tuning.

Additionally, RQ2 shows that a small-size dataset (i.e., 1%) does not improve the performance compared with CodeT5. This implies that small projects should use the basic model (i.e., CodeT5) to save the time and machine-resources for fine-tuning. This motivates our future work to explore the minimum size of the dataset that can improve the performance.

We are also interested in the trade-off between retraining costs and accuracy declines caused by data drift [22]. Software development projects conceptually vary over time. As a result, the built models may no longer be able to complete code correctly after a certain period of time. We are planning to evaluate the models with testing datasets from different future time points in order to investigate the impact of data drift.

### B. Threats to Validity

**External validity.** As this study investigated only the Spring Framework. Therefore, we cannot demonstrate the generality of the domain adaptation on the code completions. However, as each project has their own coding practice, we believe that domain adaptation is suitable for other projects.

**Internal validity.** This study used the hyper-parameters used by previous studies [5], [13]. It is possible to increase the performance by comprehensive hyper-parameter tuning.

**Construct validity.** The performance measures used in this study as proxy measures might not represent the effect of the actual code completion by developers. While these measures are often used by many studies [5], to investigate the real effectiveness, we need to monitor the actual development.

## VI. Conclusion

This paper investigated the performance of domain adaptation for code completion tasks. Our experiment result demonstrated that fine-tuning with the dataset from a specific project fits the project and improves the code completion performance (i.e., 2.5% in the edit distance rate and 5.3% in the perfect prediction rate), rather than fine-tuning with the out-of-domain datasets. Also, our results showed that the larger repositories, the greater improvement due to domain adaptation. The result implies that domain adaptation is ineffective for small projects.

## Acknowledgment

We gratefully acknowledge the financial support of JSPS for the KAKENHI grants (JP21H03416, JP21K17725, JPJSJRP20191502) and JST for the PRESTO grant (JP-MJPR22P3).

## References

[1] S. Amann, S. Proksch, S. Nadi, and M. Mezini, “A study of visual studio usage in practice,” in *Proc. of the 23rd Intl. Conf. on Software Analysis, Evolution, and Reengineering*, 2016, pp. 124–134.

[2] P. Roos, “Fast and precise statistical code completion,” in *Proc. of the 37th IEEE/ACM International Conference on Software Engineering*, vol. 2, 2015, pp. 757–759.

[3] S. Kim, J. Zhao, Y. Tian, and S. Chandra, “Code prediction by feeding trees to transformers,” in *Proc. of 2021 IEEE/ACM 43rd Intl. Conf. on Software Engineering*, 2021, pp. 150–162.

[4] R.-M. Karampatsis and C. Sutton, “Maybe deep neural networks are the best choice for modeling source code,” *ArXiv*, 2019.

[5] M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Poshyvanyk, M. Di Penta, and G. Bavota, “An empirical study on the usage of transformer models for code completion,” *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[6] M. Sharma, T. K. Mishra, and A. Kumar, “Source code auto-completion using various deep learning models under limited computing resources,” *Complex & Intelligent Systems*, vol. 8, no. 5, pp. 4357–4368, Oct 2022.

[7] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” 2019.

[8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *Proc. of the 2019 Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1*, 2019, pp. 4171–4186.

[9] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020.

[10] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, “Intellicode compose: Code generation using transformer,” in *Proc. of the 28th ACM Joint European Software Engineering Conf. and Symposium on the Foundations of Software Engineering*, 2020, p. 1433–1443.

[11] L. Pérez-Mayos, M. Ballesteros, and L. Wanner, “How much pretraining data do language models need to learn syntax?” in *Proc. of the 2021 Conf. on Empirical Methods in Natural Language Processing*, 2021, pp. 1571–1582.

[12] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, “Studying the usage of text-to-text transfer transformer to support code-related tasks,” in *Proc. of the 43rd International Conference on Software Engineering*, 2021, p. 336–347.

[13] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, “CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” in *Proc. of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 8696–8708.

[14] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, “CodeBERT: A pre-trained model for programming and natural languages,” in *Findings of the Association for Computational Linguistics: EMNLP 2020*, Nov. 2020, pp. 1536–1547.

[15] C. Chu and R. Wang, “A survey of domain adaptation for neural machine translation,” in *Proc. of the 27th International Conference on Computational Linguistics*, 2018, pp. 1304–1319.

[16] F. Gers, J. Schmidhuber, and F. Cummins, “Learning to forget: continual prediction with lstm,” in *Proc. of the 1999 Intl. Conf. on Artificial Neural Networks*, vol. 2, 1999, pp. 850–855 vol.2.

[17] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “CodeSearchNet challenge: Evaluating the state of semantic code search,” *arXiv preprint arXiv:1909.09436*, 2019.

[18] O. Dabic, E. Aghajani, and G. Bavota, “Sampling projects in github for MSR studies,” in *Proc. of the 18th IEEE/ACM International Conference on Mining Software Repositories*, 2021, pp. 560–564.

[19] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” *Soviet physics.*, vol. 10, pp. 707–710, 1965.

[20] Y. Zhou, J. Shen, X. Zhang, W. Yang, T. Han, and T. Chen, “Automatic source code summarization with graph attention networks,” *Journal of Systems and Software*, vol. 188, p. 111257, 2022.

[21] C. Bird and A. Bacchelli, “Expectations, outcomes, and challenges of modern code review,” in *Proc. of the 35th International Conference on Software Engineering*, May 2013.

[22] S. McIntosh and Y. Kamei, “Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction,” *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 412–428, 2018.