Source Code Data Augmentation for Deep Learning: A Survey

Anonymous ACL submission

Abstract

The increasingly popular adoption of deep learning models in many critical source code tasks motivates the development of data augmentation (DA) techniques to enhance training data and improve various capabilities (e.g., robustness and generalizability) of these models. Although a series of DA methods have been proposed and tailored for source code models, there is a lack of comprehensive surveys and examinations to understand their effectiveness and implications. This paper fills this gap by conducting a comprehensive and integrative survey of data augmentation for source code, wherein we systematically compile and encapsulate existing literature to provide a comprehensive overview of the field. Complementing this, we present a continually updated GitHub repository that hosts a list of up-to-date papers on DA for source code modeling.[1]

1 Introduction

Data augmentation (DA) is a technique used to increase the variety of training examples without collecting new data. It has gained popularity in recent machine learning (ML) research, with methods like back-translation (Sennrich *et al.*, 2015), and Mixup (Zhang *et al.*, 2018) being widely adopted in natural language processing (NLP), computer vision (CV), and speech recognition. These techniques have significantly improved the performance of data-centric models in low-resource domains. However, DA has not yet been fully explored in source code modeling, which is the intersection of ML and software engineering (SE). Source code modeling is an emerging area that applies ML techniques to solve various source code tasks, such as code completion, by training models on a vast amount of data available in open-source repositories. Source code data typically has two modalities: the programming language (e.g., Python and Java code) and the natural language (e.g., doc-strings and code comments), which complement each other. Such dual-modality nature of source code data presents unique challenges in tailoring DA for NLP to source code models. For example, the context of a sentence can be relatively standalone or derived from a few surrounding sentences in many NLP tasks (Feng *et al.*, 2021). However, in source code, the context can span across multiple functions or even different files, due to the widespread use of function calls, object-oriented programming, and modular design. Therefore, we argue that DA methods for source code would need to take this extended context into account, to avoid introducing errors or changing the original program's behavior. In addition, source code follows strict syntactic rules that are specified using context-free grammar. Consequently, conventional NLP DA methods, such as token substitution with similar words, may make the augmented source code fail to compile and introduce erroneous knowledge for training models.

Despite such challenges, there has been increasing interest and demand for DA for source code

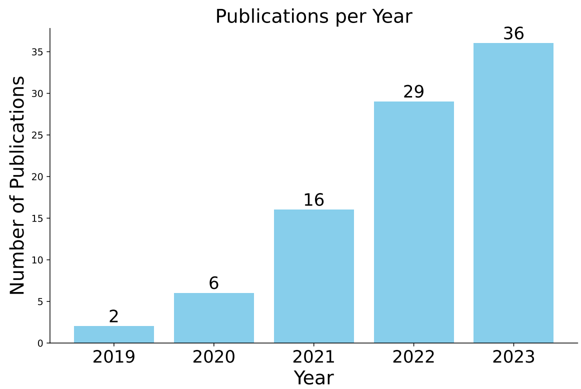

Figure 1: Yearly publications on the topic of "Source Code DA for Deep Learning". Data Statistics as of November 2023.

[1] https://anonymous.4open.science/r/ARR-DA4Code

modeling. With the growing accessibility of large, off-the-shelf, pre-trained source code models via learning from large-scale corpora, there is a growing focus on applying these models to real-world software development (Hou *et al.*, 2023). For instance, Husain *et al.* (2019) observe that many programming languages are low-resource, emphasizing the importance of DA to improve model performance and robustness on unseen data.

Our survey aims to bring attention from both ML and SE communities to this emerging field. As depicted in Figure 1, the relevant publications have been increasing in the recent five years. More precisely, we have compiled a list of core papers from the past five years, mainly from premier conferences and journals in both the ML and SE disciplines with most published in CORE Rank[2] A/A* venues. Given the escalating interest and rapidly growing research in this domain, it is timely for our survey to (1) provide a comprehensive overview of DA for source code models, and (2) pinpoint key challenges and opportunities to stimulate and guide further exploration in this emerging field. To the best of our awareness, our paper constitutes the first comprehensive survey offering an in-depth examination of DA techniques for source code models.

The structure of this paper is organized as follows:

- Section 2 offers a thorough review of three categories of DA for source code modeling: rule-based (2.1), model-based (2.2), and example interpolation-based (2.3) techniques.

- Section 3 provides a summary of prevalent strategies and techniques designed to enhance the quality of augmented data, encompassing method stacking (3.1) and optimization (3.2).

- Section 4 articulates various beneficial source code scenarios for DA, including adversarial examples for robustness (4.1), low-resource domains (4.2), retrieval augmentation (4.3), and contrastive learning (4.4).

- Section 5 delineates DA methodologies for common source code tasks, such as code authorship attribution (5.1), clone detection (5.2), defect detection and repair (5.3), code summarization (5.4), code search (5.5), code completion (5.6), code translation (5.7).

- Section 6 expounds on the challenges and future prospects in the realm of DA for source code modeling.

In addition, we provide more details in the Appendix to help readers have a more comprehensive understanding of source code data augmentation. Through this work, we hope to emulate prior surveys which have analyzed DA techniques for other data types, such as text (Feng *et al.*, 2021), time series (Wen *et al.*, 2020), and images (Shorten and Khoshgoftaar, 2019). Our intention is to pique further interest, spark curiosity, and encourage further research in the field of data augmentation, specifically focusing on its application to source code.

2 Source Code Data Augmentation Methods for Deep Learning

This section categorizes the mainstream DA techniques specifically designed for source code models into three families: rule-based, model-based, and example-interpolation techniques. We explain studies of different families as follows.

2.1 Rule-based Techniques

A large number of DA methods utilize *predetermined rules* to transform the programs without breaking syntax rules and semantics. Specifically, these rules mainly implicitly leverage ASTs to transform the code snippets. The transformations can include operations such as replacing variable names, renaming method names, and inserting dead code. Besides the basic program syntax, some code transformations consider deeper structural information, such as control-flow graph (CFG) and use-define chains (UDC) (Quiring *et al.*, 2019). Additionally, a small part of rule-based DA techniques focuses on augmenting the natural language context in the code snippets, including doc-strings and comments (Bahrami *et al.*, 2021).

Zhang *et al.* proposed MHM (2020a), a method of iteratively renaming identifiers in the code snippets. Considered as the approach to generate examples for adversarial training, MHM greatly improves the robustness of source code models. Later, Srikant *et al.* (2021) consider program obfuscations as adversarial perturbations, where they rename program variables in an attempt to hide the program's intent from a reader. By applying these perturbed examples to the training stage, the source code models become more robust to the adversarial attack. Instead of just renaming identifiers,

[2]We refer to the venues listed at `http://portal.core.edu.au/conf-ranks/` and `http://portal.core.edu.au/jnl-ranks/`.

BUGLAB-Aug (Allamanis *et al.*, 2021) contains more rules to augment code snippets, emphasizing both the programming language and natural language, such as comment deletion, comparison expression mirroring, and if-else branch swapping.

Brockschmidt *et al.* (2019) present a generative source code model by augmenting the given AST with additional edges to learn diverse code expressions. Instead of the direct augmentation on AST, Quiring *et al.* (2019) propose three different augmentation schemes via the combination of AST and CFG, UDC and declaration-reference mapping (DRM), named as Control Transformations, Declaration Transformations and API Transformations.

Another line of work is augmenting the natural language context in source code. QRA (Huang *et al.*, 2021) augments examples by rewriting natural language queries when performing code search and code question answering. It rewrites queries with minor rule-based modifications that share the same semantics as the original one. Specifically, it consists of three modifications: randomly deleting a word, randomly switching the position of two words, and randomly copying a word. Inspired by this approach, Park *et al.* (2023) devise KeyDAC with an emphasis on the query keywords. KeyDAC augments on both natural language and programming language. For natural language query, it follows the rules in QRA but only modifies non-keywords. In terms of programming language augmentation, KeyDAC simply uses ASTs to rename program variables, similar to the aforementioned work.

2.2 Model-based Techniques

A series of DA techniques for source code target training various models to augment data. Intuitively, Mi *et al.* (2021) utilize Auxiliary Classifier Generative Adversarial Networks (AC-GAN) to generate augmented programs. To increase the training data for code summarization, CDA-CS (Song *et al.*, 2022) uses the pre-trained BERT model to replace synonyms for non-keywords in code comments, which benefits the source code downstream tasks.

While these methods largely adapt the existing model-based DA techniques for general purposes, most DA approaches are specifically designed for source code models. Li *et al.* (2022e) introduce IRGen, a genetic-algorithm-based model using compiler intermediate representation (LLVM IR) to augment source code embeddings, where IRGen generates a piece of source code into a range of semantically identical but syntactically distinct IR codes for improving model's contextual understanding. Studies like (Roziere *et al.*, 2021) have investigated the suitability of the multilingual generative source code models for unsupervised programming language translation via Back-translation, in the similar scope of the one for NLP. However, unlike the one in NLP that commonly uses English as the intermediate language, Back-translation here is defined as translating between two programming languages via the natural language as an intermediate language. Pinku *et al.* (2023) exploit another generative source code model, Transcoder, to perform source-to-source translation for augmenting cross-language source code.

2.3 Example Interpolation Techniques

Another category of data augmentation (DA) techniques, originated by Mixup (Zhang *et al.*, 2018), involves interpolating the inputs and labels of two or more actual examples. For instance, given that a binary classification task in CV and two images of a dog and a cat, respectively, these DA approaches like Mixup can blend these two image inputs and their corresponding labels based on a randomly selected weight. This collection of methods is also termed Mixed Sample Data Augmentation. Despite trials in the context of text classification problems, such methods are hard to deploy in the realm of source code, as each code snippet is constrained by its unique program grammar and functionality.

In contrast to the aforementioned surface-level interpolation, the majority of example-interpolation DA methods are enhanced to fuse multiple real examples into a single input via model embeddings (Feng *et al.*, 2021). Dong *et al.* (2023b) merge rule-based techniques for source code models with Mixup to blend the representations of the original code snippet and its transformation. This approach is commonly regarded as the linear interpolation technique deployed in NLP classification tasks.

3 Strategies and Techniques

In real-world applications, the design and efficacy of DA techniques for source code models are influenced by a variety of factors, such as computing cost, example diversity, and models' robustness. This section highlights these factors, offering insights and techniques for devising and optimizing suitable DA methods.

3.1 Method Stacking

As discussed in Section 2, numerous DA strategies are proposed concurrently in a single work, aiming to enhance the models' performance. Typically, the combination entails two types: same-type DA or a mixture of different DA methods. The former is typically applied in rule-based DA techniques, stemming from the realization that a single code transformation cannot fully represent the diverse code style and implementation found in the real world.

Several works (Shi *et al.*, 2023; Huang *et al.*, 2021) demonstrate that merging multiple types of DA techniques can enhance the performance of source code models. Mi *et al.* (2021) combine rule-based code transformation schemes with model-based DA using AC-GAN to create an augmented corpus for model training. Instead of augmenting on programming language, CDA-CS (Song *et al.*, 2022) encompasses two kinds of DA techniques: rule-based non-keyword extraction and model-based non-keyword replacement.

3.2 Optimization

In certain scenarios such as enhancing robustness and minimizing computational cost, optimally selecting specific augmented example candidates is crucial. We denote such goal-oriented candidate selections in DA as *optimization*. Subsequently, we introduce three types of strategies: probabilistic, model-based, and rule-based selection. Probabilistic selection is defined as the optimization via sampling from a probability distribution, while model-based selection is guided by the model to select the most proper examples. In terms of rule-based selection, it is an optimization strategy where specific predetermined rules or heuristics are used to select the most suitable examples.

3.2.1 Probabilistic Selection

We introduce three representative probabilistic selection strategies, MHM, QMDP, and BUGLAB-Aug. MHM (Zhang *et al.*, 2020a) adopts the Metropolis-Hastings probabilistic sampling method, which is a Markov Chain Monte Carlo technique, to choose adversarial examples via identifier replacement. Similarly, QMDP (Tian *et al.*, 2021) uses a Q-learning approach to strategically select and execute rule-based structural transformations on the source code, thereby guiding the generation of adversarial examples. In BUGLAB-Aug, Allamanis *et al.* (2021) model the probability of applying a specific rewrite rule at a location in a code snippet similar to the pointer net.

3.2.2 Model-based Selection

Several DA techniques employing this strategy use the model's gradient information to guide the selection of augmented examples. A representative approach is the DAMP method (Yefet *et al.*, 2020), which optimizes based on the model loss to select and generate adversarial examples via variable renaming. Another variant, SPACE (Li *et al.*, 2022b), performs selection and perturbation of code identifiers' embeddings via gradient ascent, targeting to maximize the model's performance impact while upholding semantic and grammatical correctness of the programming language. A more complex technique, ALERT (Yang *et al.*, 2022b), uses a genetic algorithm in its gradient-based selection strategy. It evolves a population of candidate solutions iteratively, guided by a fitness function that calculates the model's confidence difference, aiming to identify the most potent adversarial examples.

3.2.3 Rule-based Selection

Rule-based selection stands as a powerful approach, featuring predetermined fitness functions or rules. This method often relies on evaluation metrics for decision-making. For instance, IRGen (Li *et al.*, 2022e) utilizes a Genetic-Algorithm-based optimization technique with a fitness function based on IR similarity. On the other hand, ACCENT (Zhou *et al.*, 2022) and RADAR apply evaluation metrics such as CodeBLEU, respectively, to guide the selection and replacement process, aiming for maximum adversarial impact. Finally, STRATA (Springer, 2021) employs a rule-based technique to select high-impact subtokens that significantly alter the model's interpretation of the code.

4 Scenarios

This section delves into several commonplace source code scenarios where DA approaches can be applied.

4.1 Adversarial Examples for Robustness

Robustness presents a critical and complex dimension of software engineering, necessitating the creation of semantically-preserved adversarial examples to discern and mitigate vulnerabilities within

Table 1: Comparing a selection of DA methods by various aspects relating to their applicability, dependencies, and requirements. *PL*, *NL*, *EI*, *Prob*, *Tok*, *KWE*, *TA*, and *LA* stand for **P**rogramming **L**anguage, **N**atural **L**anguage, **E**xample **I**nterpolation, **Prob**ability, **Tok**enization, **K**ey**W**ord **E**xtraction, **T**ask-**A**gnostic, and **L**anguage-**A**gnostic. *PL* and *NL* determine if the DA method is applied to the programming language or natural language context. *Preprocess* denotes preprocessing required besides the program parsing. *Parsing* refers to the type of feature used by the DA method during program parsing. *Level* denotes the depth at which data is modified by the DA. *TA* and *LA* represent whether the DA method can be applied to different tasks or programming languages. As most papers do not clearly state if their DA methods are *TA* and *LA*, we subjectively denote the applicability.

DA Method	Category	PL	NL	Optimization	Preprocess	Parsing	Level	TA	LA
ComputeEdge (Brockschmidt *et al.*, 2019)	Rule	✓	✗	—	—	AST	AST	✓	✓
RefineRepresentation (Bielik and Vechev, 2020)	Rule	✓	✗	Model	—	AST	AST	✓	✓
Control Transformations (Quiring *et al.*, 2019)	Rule	✓	✗	Prob	—	AST+CFG+UDC	Input	✓	✗
Declaration Transformations (Quiring *et al.*, 2019)	Rule	✓	✗	Prob	—	AST+DRM	Input	✓	✗
API Transformations (Quiring *et al.*, 2019)	Rule	✓	✗	Prob	—	AST+CFG+DRM	Input	✓	✗
DAMP (Yefet *et al.*, 2020)	Rule	✓	✗	Model	—	AST	Input	✓	✓
IBA (Huang *et al.*, 2021)	Rule	✗	✓	—	Tok	—	Embed	✗	✓
QRA (Huang *et al.*, 2021)	Rule	✓	✗	—	Tok	—	Input	✗	✓
MHM (Zhang *et al.*, 2020a)	Rule	✗	✓	Prob	—	AST	Input	✓	✗
AugmentedCode (Bahrami *et al.*, 2021)	Rule	✓	✗	—	Tok	—	Input	✗	✓
QMDP (Tian *et al.*, 2021)	Rule	✓	✗	Prob	Tok	AST	Input	✓	✗
Transpiler (Jain *et al.*, 2021)	Rule	✓	✗	Prob	—	AST	Input	✓	✗
BUGLAB-Aug (Allamanis *et al.*, 2021)	Rule	✓	✗	Prob	Tok	AST	Input	✗	✓
SPAT (Yu *et al.*, 2022)	Rule	✓	✗	Model	—	AST	Input	✓	✗
RoPGen (Li *et al.*, 2022c)	Rule	✓	✗	Model	—	AST	Input	✓	✗
ACCENT (Zhou *et al.*, 2022)	Rule	✓	✗	Rule	—	AST	Input	✓	✓
SPACE (Li *et al.*, 2022b)	Rule	✓	✗	Model	Tok	AST	Embed	✓	✓
ALERT (Yang *et al.*, 2022b)	Rule	✓	✗	Model	Tok	AST	Input	✓	✓
IRGen (Li *et al.*, 2022e)	Rule	✓	✗	Rule	—	AST+IR	IR	✓	✓
Linear Extrapolation (Li *et al.*, 2022a)	EI	✓	✓	—	—	—	Embeb	✓	✓
Gaussian Scaling (Li *et al.*, 2022a)	Rule	✓	✓	Model	—	—	Embeb	✓	✓
CodeTransformator (Zubkov *et al.*, 2022)	Rule	✓	✗	Rule	—	AST	Input	✓	✗
RADAR (Yang *et al.*, 2022a)	Rule	✓	✗	Rule	—	AST	Input	✓	✗
AC-GAN (Mi *et al.*, 2021)	Model	✓	✗	—	—	—	Input	✓	✓
CDA-CS (Song *et al.*, 2022)	Model	✗	✓	Model	KWE	—	Input	✗	✓
srcML-embed (Li *et al.*, 2022d)	Rule	✓	✗	—	—	AST	Embed	✓	✗
ProgramTransformer (Rabin and Alipour, 2022)	Rule	✓	✗	—	—	AST	Input	✓	✗
Back-translation (Ahmad *et al.*, 2023)	Model	✓	✗	—	Tok	—	Input	✗	✓
MixCode (Dong *et al.*, 2023b)	Rule+EI	✓	✓	—	—	—	Embed	✓	✓
NP-GD (Shen *et al.*, 2023)	Model	✓	✗	Model	Tok	—	Embed	✓	✓
ExploitGen (Yang *et al.*, 2023)	Rule	✗	✓	—	—	—	Input	✓	✗
SoDa (Shi *et al.*, 2023)	Model	✓	✓	—	—	AST	Input	✓	✓
Transcompiler (Pinku *et al.*, 2023)	Model	✓	✗	—	—	—	Input	✓	✗
STRATA (Springer, 2021)	Rule	✓	✗	Model	Tok	AST	Input	✓	✓
KeyDAC (Pack *et al.*, 2023)	Rule	✓	✓	—	KWE	AST	Embed	✗	✓
Simplex Interpolation (Zhang *et al.*, 2022)	EI	✓	✗	—	—	AST+IR	Embed	✗	✓

source code models. There is a surge in designing more effective DA techniques for generating these examples in recent years. Several studies (Yefet *et al.*, 2020; Li *et al.*, 2022c; Srikant *et al.*, 2021; Li *et al.*, 2022b; Anand *et al.*, 2021) have utilized various DA methods for testing and enhancing model robustness. Wang *et al.* (2023) have gone a step further to consolidate universally accepted code transformation rules to establish the first benchmark for source code model robustness.

4.2 Low-Resource Domains

In the domain of software engineering, the resources of programming languages are severely imbalanced (Orlanski *et al.*, 2023). While some of the most popular programming languages like Python and Java play major roles in the open-source repositories, many languages like Rust are starkly low-resource. As source code models are trained on open-source repositories and forums, the programming language resource imbalance can adversely impact their performance on the resource-scarce programming languages. Furthermore, the application of DA methods within low-resource domains is a recurrent theme within the CV and NLP communities (Shorten and Khoshgoftaar, 2019; Feng *et al.*, 2021). Yet, this scenario remains underexplored within the source code discipline.

In order to increase data in the low-resource domain for representation learning, Li *et al.* (2022e) tend to add more training data to enhance source code model embeddings by unleashing the power of compiler IR. Ahmad *et al.* (2023) propose to use source code models to perform Back-translation DA, taking into consideration the scenario of low-resource programming languages. Meanwhile, (Chen and Lampouras, 2023) underscore the fact that source code datasets are markedly smaller than their NLP equivalents, which often encompass millions of instances. As a result, they commence investigations into code completion tasks under this context and experiment with Back-translation and variable renaming. Shen *et al.* (2023) contend that the generation of bash comments is hampered by a dearth of training data and thus explore model-based DA methods for this task.

4.3 Retrieval Augmentation

Increasing interest has been observed in the application of DA for retrieval augmentation within NLP and source code (Lu *et al.*, 2022). These retrieval augmentation frameworks for source code models incorporate retrieval-augmented examples from the training set when pre-training or fine-tuning source code models. This form of augmentation enhances the parameter efficiency of models, as they are able to store less knowledge within their parameters and instead retrieve it. It is shown as a promising application of DA in various source code downstream tasks, such as code summarization (Zhang *et al.*, 2020b) and program repair (Nashid *et al.*, 2023).

4.4 Contrastive Learning

Another source code scenario to deploy DA methods is contrastive learning, where it enables models to learn an embedding space in which similar samples are close to each other while dissimilar ones are far apart (Wang *et al.*, 2022; Zhang *et al.*, 2022). As the training datasets commonly contain limited sets of positive samples, DA methods are preferred to construct similar samples as the positive ones. Liu *et al.* (2023b) make use of contrastive learning with DA to devise superior pre-training paradigms for source code models, while some works study the advantages of this application in some source code tasks like defect detection (Cheng *et al.*, 2022) and clone detection (Zubkov *et al.*, 2022).

5 Downstream Tasks

While many aforementioned DA methods are deemed task-agnostic, most of them have been only applied to specific tasks. Therefore, we share an overview of how these methods work for common source code tasks and evaluation datasets.

5.1 Code Authorship Attribution

Code authorship attribution is the process of identifying the author of a given code, usually achieved by source code models. Yang *et al.* (2022b) initially investigate generating adversarial examples on the *Google Code Jam* (GCJ) dataset, which effectively fools source code models to identify the wrong author of a given code snippet. By training with these augmented examples, the model's robustness can be further improved. Li *et al.* (2022c) propose another DA method called RoPGen for the adversarial attack and demonstrate its efficacy on GCJ. Dong *et al.* (2023a) empirically study the effectiveness of several existing DA approaches for NLP on several source code tasks, including authorship attribution on *GCJ*.

5.2 Clone Detection

Code clone detection refers to the task of identifying if the given code snippet is syntactically or semantically similar to the original sample Jain *et al.* (2021) propose correct-by-construction DA via compiler information to generate many variants with equivalent functionality of the training sample and show its effectiveness of improving the model robustness on *BigCloneBench* and a self-collected JavaScript dataset. Pinku *et al.* (2023) later use Transcompiler to translate between limited source code in Python and Java and increase the training data for cross-language code clone detection.

5.3 Program Repair

Program repair, in other words, bug or vulnerability fix, captures the bugs in given code snippets and generates repaired versions. Allamanis *et al.* (2021) implement BUGLAB-Aug, a DA framework of self-supervised bug detection and repair. BUGLAB-Aug has two sets of code transformation rules, one is a bug-inducing rewrite and the other one is rewriting as DA. Their approach boosts the performance and robustness of source code models simultaneously. Cheng *et al.* (2022) present a path-sensitive code embedding technique called ContraFlow, which uses self-supervised contrastive learning to detect defects based on value-flow paths. ContraFlow utilizes DA to generate contrastive value-flow representations of three datasets (namely *D2A*, Fan and *FFMPeg+Qemu*) to learn the (dis)-similarity among programs. Ding *et al.* (2021) present a novel self-supervised model focusing on identifying (dis)similar functionalities of source code, which outperforms the state-of-the-art models on *REVEAL* and *FFMPeg+Qemu*. Specifically, they design code transformation heuristics to automatically create bugged programs and similar code for augmenting pre-training data.

5.4 Code Summarization

Code summarization is considered as a task that generates a comment for a piece of the source code, and is thus also named code comment generation. Zhang *et al.* (2020c) apply MHM to perturb training examples and mix them with the original ones for adversarial training, which effectively improves the robustness of source code models in summarizing the adversarial code snippets. Zhang *et al.* (2020b) develop a retrieval-augmentation framework for code summarization, relying on similar code-summary pairs to generate the new summary on *PCSD* and *JCSD* datasets. Based on this framework, Liu *et al.* (2020) leverage Hybrid GNN to propose a novel retrieval-augmented code summarization method and use it during model training on the self-collected CCSD dataset. Zhou *et al.* (2022) generate adversarial examples of a Python dataset (Wan *et al.*, 2018) and *JSCD* to evaluate and enhance the source code model robustness.

5.5 Code Search

Code search, or code retrieval, is a text-code task that searches code snippets based on the given natural language queries. The source code models on this task need to map the semantics of the text to the source code (Li *et al.*, 2022a, 2023; Huang *et al.*, 2023; Ma *et al.*, 2023). Bahrami *et al.* (2021) increase the code search queries by augmenting the natural language context such as doc-string, code comments and commit messages. Shi *et al.* (2022) use AST-focused DA to replace the function and variable names of the data in *CodeSearchNet* and *CoSQA* (Huang *et al.*, 2021). Specifically, Shi *et al.* introduce soft data augmentation (SoDa), without external transformation rules on code and text. With SoDa, the model predicts tokens based on dynamic masking or replacement when processing *CodeSearchNet*. Instead of applying rule-based DA techniques, Li *et al.* (2022a) manipulate the representation of the input data by interpolating examples of *CodeSearchNet*.

5.6 Code Completion

Code completion requires source code models to generate lines of code to complete given programming tasks. Anand *et al.* (2021) suggest that source code models are vulnerable to adversarial examples which are perturbed with transformation rules. Lu *et al.* (2022) propose a retrieval-augmented code completion framework composed of the rule-based DA module to generate on *PY150* and *GitHub Java Corpus* datasets (Allamanis and Sutton, 2013). Wang *et al.* (2023) customize over 30 transformations specifically for code on docstrings, function and variable names, code syntax, and code format and benchmark generative source code models on *HumanEval* and *MBPP*. Yang *et al.* (2022a) devise transformations on functional descriptions and signatures to attack source code models and show that their performances are susceptible.

5.7 Code Translation

Similar to neural machine translation in NLP (Stahlberg *et al.*, 2020), the task is to translate source code written in a specific programming language to another one. Ahmad *et al.* (2023) apply data augmentation through back-translation to enhance unsupervised code translation. They use pre-trained sequence-to-sequence models to translate code into natural language summaries and then back into code in a different programming language, thereby creating additional synthetic training data to improve model performance. Chen *et al.* (2023) utilize Back-translation and variable augmentation techniques to yield the improvement in code translation on *CodeTrans* (Lu *et al.*, 2021).

6 Challenges and Opportunities

When it comes to source code, DA faces significant challenges. Nonetheless, it's crucial to acknowledge that these challenges pave the way for new possibilities and exciting opportunities in this area of work.

Discussion on theory. Currently, there is a noticeable gap in the in-depth exploration and theoretical understanding of DA methods in source code. Most existing research on DA is centered around image processing and natural language fields, viewing data augmentation as a way of applying pre-existing knowledge about data or task invariance (Wu *et al.*, 2020). When shifting to source code, much of the previous work introduces new methods or demonstrates how DA techniques can be effective for subsequent tasks. However, these studies often overlook why and how particularly from a mathematical perspective. By exploring DA in this way, we can better understand its underlying principles without being solely dependent on experimental validation.

More study on pre-trained models. In recent years, pre-trained source code models have been widely applied in source code, containing rich knowledge through self-supervision on a huge scale of corpora (Feng *et al.*, 2020; Guo *et al.*, 2021). Numerous studies have been conducted utilizing pre-trained source code models for the purpose of DA, yet, most of these attempts are confined to mask token replacement (Shi *et al.*, 2023), and direct generation after fine-tuning (Ahmad *et al.*, 2023; Pinku *et al.*, 2023). An emergent research opportunity lies in exploring the potential of DA in the source code domain with the help of large language models (LLMs) trained on a large amount of text and source code. LLMs have the capability of context generation based on prompted instructions and provided examples, making them a choice to automate the DA process in NLP (Yoo *et al.*, 2021; Wang *et al.*, 2021a). Different from the previous usages of pre-trained models in DA, these works open the era of "prompt-based DA". In contrast, the exploration of prompt-based DA in source code domains remains a relatively untouched research area. Another direction is to harness the internal knowledge encoded in pre-trained source code models. For example, previous work (Karmakar and Robbes, 2021; Wan *et al.*, 2022) shows that ASTs and code semantics can be induced from these models without the static analysis tools.

More exploration on project-level source code and low-resource programming languages. The existing methods have made sufficient progress in function-level code snippets and common programming languages. The emphasis on code snippets at the function level fails to capture the intricacies and complexities of programming in real-world scenarios, where developers often work with multiple files and folders simultaneously. Therefore, we highlight the importance of exploring DA approaches on the project level. The DA on source code projects can be distinct from the function-level DA, as it may involve more information such as the interdependencies between different code modules, high-level architectural considerations, and the often intricate relationship between data structures and algorithms used across the project (Mockus *et al.*, 2002). At the same time, limited by data resources (Husain *et al.*, 2019; Orlanski *et al.*, 2023), augmentation methods of low-resource languages are scarce, although they have more demand for DA. Exploration in these two directions is still limited, and they could be promising directions.

Lack of unification. The current body of literature on data augmentation (DA) for source code presents a challenging landscape, with the most popular methods often being portrayed in a supplementary manner. A handful of empirical studies have sought to compare DA methods for source code models (Rodrigues *et al.*, 2023; Dong *et al.*, 2023a). However, none of these works leverages most of the existing advanced DA methods for source code models. Whereas there are well-accepted frameworks for DA for CV and DA for NLP, a corresponding library of generalized DA techniques for source code models is conspicuously absent. Furthermore, as existent DA methods are usually evaluated with various datasets, it is hard to determine the efficacy. Therefore, we posit that the progression of DA research would be significantly facilitated by the establishment of standardized and unified benchmark tasks, along with datasets, for the purpose of contrasting and evaluating the effectiveness of different augmentation methods. This would pave the way towards a more systematic and comparative understanding of the benefits and limitations of these methods.

7 Conclusion

Our paper comprehensively analyzes data augmentation techniques in the context of source code. We first explain the concept of data augmentation and its function. We then examine the primary data augmentation methods commonly employed in source code research and explore augmentation approaches for typical source code applications and tasks. Finally, we conclude by outlining the current challenges in the field and suggesting potential directions for future source code research. In presenting this paper, we aim to assist researchers in selecting appropriate data augmentation techniques and encourage further exploration and advancement in this field.

Limitations

While the work presented in this paper has its merits, we acknowledge the several limitations. Firstly, our work only surveys imperative programming languages used for general-purpose programming. Therefore, some DA methods for declarative languages (Zhuo *et al.*, 2023b) or minor downstream tasks like cryptography misuse detection (Rodrigues *et al.*, 2023), including SQL. Secondly, our focus has been primarily on function-level DA within the source code context. As such, future development in project-level DA methods remains needed. Nonetheless, this paper offers a valuable collection of general-purpose DA techniques for source code models, and we hope that it can serve as an inspiration for further research in this area. Thirdly, given the page limits, the descriptions presented in this survey are essentially brief in nature. Our approach has been to offer the works in meaningful structured groups rather than unstructured sequences, to ensure comprehensive coverage. This work can be used as an index where more detailed information can be found in the corresponding works. Lastly, it is worth noting that this survey is purely qualitative and does not include any experiments or empirical results. To provide more meaningful guidance, it would be helpful to conduct comparative experiments across different DA strategies. We leave this as a suggestion for future work.

References

Wasi Uddin Ahmad *et al.* 2023. Summarize and generate to back-translate: Unsupervised translation of programming languages. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 1528–1542, Dubrovnik, Croatia. Association for Computational Linguistics.

Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *2013 10th working conference on mining software repositories (MSR)*, pages 207–216. IEEE.

Miltiadis Allamanis *et al.* 2017. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51:1 – 37.

Miltiadis Allamanis *et al.* 2021. Self-supervised bug detection and repair. *Advances in Neural Information Processing Systems*, 34:27865–27876.

Uri Alon *et al.* 2019. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*.

Mrinal Anand *et al.* 2021. Adversarial robustness of program synthesis models. In *Advances in Programming Languages and Neurosymbolic Systems Workshop*.

Mehdi Bahrami *et al.* 2021. Augmentedcode: Examining the effects of natural language resources in code retrieval models. *arXiv preprint arXiv:2110.08512*.

Pavol Bielik and Martin Vechev. 2020. Adversarial robustness for code. In *International Conference on Machine Learning*, pages 896–907. PMLR.

Marc Brockschmidt *et al.* 2019. Generative code modeling with graphs. In *International Conference on Learning Representations*.

Pinzhen Chen and Gerasimos Lampouras. 2023. Exploring data augmentation for code generation tasks. In *Findings of the Association for Computational Linguistics: EACL 2023*, pages 1497–1505.

Xiao Cheng *et al.* 2022. Path-sensitive code embedding via contrastive learning for software vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 519–531.

Michael L Collard *et al.* 2013. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. pages 516–519. IEEE.

Yangruibo Ding *et al.* 2021. Towards learning (dis)-similarity of source code from program contrasts. In *Annual Meeting of the Association for Computational Linguistics*.

Zeming Dong *et al.* 2023a. Boosting source code learning with data augmentation: An empirical study. *arXiv preprint arXiv:2303.06808*.

Zeming Dong *et al.* 2023b. Mixcode: Enhancing code classification by mixup-based data augmentation. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 379–390. IEEE.

Steven Y Feng *et al.* 2021. A survey of data augmentation approaches for nlp. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 968–988.

Zhangyin Feng *et al.* 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547.

Xiaodong Gu *et al.* 2016. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 631–642.

Daya Guo *et al.* 2021. Graphcode{bert}: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Hossein Hajipour *et al.* 2022. Simscood: Systematic analysis of out-of-distribution behavior of source code models. *arXiv preprint arXiv:2210.04802*.

Xiaoshuai Hao *et al.* 2023. Mixgen: A new multi-modal data augmentation. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 379–389.

Vincent J Hellendoorn *et al.* 2018. 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*, pages 152–162.

Xinyi Hou *et al.* 2023. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620*.

Qiang Hu *et al.* 2022. Codes: A distribution shift benchmark dataset for source code learning. *arXiv preprint arXiv:2206.05480*.

Junjie Huang *et al.* 2021. Cosqa: 20,000+ web queries for code search and question answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5690–5700.

Qiao Huang *et al.* 2018. Api method recommendation without worrying about the task-api knowledge gap. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 293–304.

Xiangbing Huang *et al.* 2023. Towards better multilingual code search through cross-lingual contrastive learning. In *Proceedings of the 14th Asia-Pacific Symposium on Internetware*, pages 22–32.

Hamel Husain *et al.* 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Paras Jain *et al.* 2021. Contrastive code representation learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5954–5971.

Anjan Karmakar and Romain Robbes. 2021. What do pre-trained code models know about code? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1332–1336. IEEE.

Haochen Li *et al.* 2022a. Exploring representation-level augmentation for code search. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 4924–4936, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Haochen Li *et al.* 2023. Rethinking negative pairs in code search.

Yiyang Li *et al.* 2022b. Semantic-preserving adversarial code comprehension. In *Proceedings of the 29th International Conference on Computational Linguistics*, pages 3017–3028.

Zhen Li *et al.* 2022c. Ropgen: Towards robust code authorship attribution via automatic coding style transformation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1906–1918.

Zhiming Li *et al.* 2022d. Cross-lingual transfer learning for statistical type inference. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 239–250.

Zongjie Li *et al.* 2022e. Unleashing the power of compiler intermediate representation to enhance neural program embeddings. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2253–2265.

Fangyu Liu *et al.* 2023a. Matcha: Enhancing visual language pretraining with math reasoning and chart derendering. In *Proceedings of the 61th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Shangqing Liu *et al.* 2020. Retrieval-augmented generation for code summarization via hybrid gnn. In *International Conference on Learning Representations*.

Shangqing Liu *et al.* 2023b. Contrabert: Enhancing code pre-trained models via contrastive learning.

Yan Liu *et al.* 2023c. Uncovering and quantifyingsocialbiases incodegeneration.

Shuai Lu *et al.* 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Shuai Lu *et al.* 2022. Reacc: A retrieval-augmented code completion framework. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6227–6240.

Yingwei Ma *et al.* 2023. Mulcs: Towards a unified deep representation for multilingual code search. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 120–131. IEEE.

Qing Mi *et al.* 2021. The effectiveness of data augmentation in code readability classification. *Information and Software Technology*, 129:106378.

Audris Mockus *et al.* 2002. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3):309–346.

Noor Nashid *et al.* 2023. Retrieval-based prompt selection for code-related few-shot learning. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*.

Erik Nijkamp *et al.* 2023. Codegen: An open large language model for code with multi-turn program synthesis. *ICLR*.

Gabriel Orlanski *et al.* 2023. Measuring the impact of programming language distribution. *arXiv preprint arXiv:2302.01973*.

Shinwoo Pack *et al.* 2023. Contrastive learning with keyword-based data augmentation for code search and code question answering. In *Conference of the European Chapter of the Association for Computational Linguistics*.

Subroto Nag Pinku *et al.* 2023. Pathways to leverage transcompiler based data augmentation for cross-language clone detection. *arXiv preprint arXiv:2303.01435*.

Maryam Vahdat Pour *et al.* 2021. A search-based testing framework for deep neural networks of source code embedding. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 36–46. IEEE.

Erwin Quiring *et al.* 2019. Misleading authorship attribution of source code using adversarial learning. In *USENIX Security Symposium*, pages 479–496.

Md Rafiqul Islam Rabin and Mohammad Amin Alipour. 2022. Programtransformer: A tool for generating semantically equivalent transformed programs. *Software Impacts*, 14:100429.

Md Rafiqul Islam Rabin *et al.* 2021. On the generalizability of neural program models with respect to semantic-preserving program transformations. *Information and Software Technology*, 135:106552.

Arijit Ray *et al.* 2019. Sunny and dark outside?! improving answer consistency in vqa through entailed question generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5860–5865.

de Paula Rodrigues *et al.* 2023. Detecting cryptography misuses with machine learning: Graph embeddings, transfer learning and data augmentation in source code related tasks. *IEEE Transactions on Reliability*.

Baptiste Roziere *et al.* 2021. Leveraging automated unit tests for unsupervised code translation. In *International Conference on Learning Representations*.

Rico Sennrich *et al.* 2015. Improving neural machine translation models with monolingual data. *arXiv preprint arXiv:1511.06709*.

Yiheng Shen *et al.* 2023. Bash comment generation via data augmentation and semantic-aware codebert. *Available at SSRN 4385791*.

Ensheng Shi *et al.* 2023. Cocosoda: Effective contrastive learning for code search. In *Proceedings of the 45th International Conference on Software Engineering*.

Zejian Shi *et al.* 2022. Cross-modal contrastive learning for code search. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 94–105. IEEE.

Connor Shorten and Taghi M. Khoshgoftaar. 2019. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6:1–48.

Zixuan Song *et al.* 2022. Do not have enough data? an easy data augmentation for code summarization. In *2022 IEEE 13th International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, pages 1–6. IEEE.

Jacob M. Springer. 2021. Strata: Simple, gradient-free attacks for models of code. *Preprint*, arXiv:2009.13562.

Shashank Srikant *et al.* 2021. Generating adversarial computer programs using optimized obfuscations. In *International Conference on Learning Representations*.

Felix Stahlberg *et al.* 2020. Neural machine translation: A review. *Journal of Artificial Intelligence Research*, 69:343–418.

Dídac Surís *et al.* 2023. Vipergpt: Visual inference via python execution for reasoning. *arXiv preprint arXiv:2303.08128*.

Ruixue Tang *et al.* 2020. Semantic equivalent adversarial data augmentation for visual question answering. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XIX 16*, pages 437–453. Springer.

Junfeng Tian *et al.* 2021. Generating adversarial examples of source code classification models via q-learning-based markov decision process. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 807–818. IEEE.

Christoph Treude and Martin P Robillard. 2016. Augmenting api documentation with insights from stack overflow. In *Proceedings of the 38th International Conference on Software Engineering*, pages 392–403.

Yao Wan *et al.* 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 397–407.

Yao Wan *et al.* 2022. What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2377–2388.

Shiqi Wang *et al.* 2023. Recode: Robustness evaluation of code generation models. In *Proceedings of the 61th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.

Shuohang Wang *et al.* 2021a. Want to reduce labeling cost? gpt-3 can help. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 4195–4205.

Xiao Wang *et al.* 2022. Heloc: Hierarchical contrastive learning of source code representation. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 354–365.

Zeyu Wang *et al.* 2021b. Plot2api: recommending graphic api from plot via semantic parsing guided neural network. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 458–469. IEEE.

Qingsong Wen *et al.* 2020. Time series data augmentation for deep learning: A survey. In *International Joint Conference on Artificial Intelligence*.

Sen Wu *et al.* 2020. On the generalization effects of linear transformations in data augmentation. In *International Conference on Machine Learning*, pages 10410–10420. PMLR.

Frank F Xu *et al.* 2020. Incorporating external knowledge through pre-training for natural language to code generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6045–6052.

Guang Yang *et al.* 2022a. How important are good method names in neural code generation? a model robustness perspective. *arXiv preprint arXiv:2211.15844*.

Guang Yang *et al.* 2023. Exploitgen: Template-augmented exploit code generation based on codebert. *Journal of Systems and Software*, 197:111577.

Zhou Yang *et al.* 2022b. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1482–1493.

Noam Yefet *et al.* 2020. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450.

Kang Min Yoo *et al.* 2021. Gpt3mix: Leveraging large-scale language models for text augmentation. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2225–2239.

Shiwen Yu *et al.* 2022. Data augmentation by program transformation. *Journal of Systems and Software*, 190:111304.

He Zhang *et al.* 2011. Identifying relevant studies in software engineering. *Information and Software Technology*, 53(6):625–637.

Hongyi Zhang *et al.* 2018. mixup: Beyond empirical risk minimization. In *International Conference on Learning Representations*.

Huangzhao Zhang *et al.* 2020a. Generating adversarial examples for holding robustness of source code processing models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1169–1176.

Jian Zhang *et al.* 2020b. Retrieval-based neural source code summarization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1385–1397.

Xiaoqing Zhang *et al.* 2020c. Training deep code comment generation models via data augmentation. pages 185–188.

Yifan Zhang *et al.* 2022. Combo: Pre-training representations of binary code using contrastive learning. *arXiv preprint arXiv:2210.05102*.

Yu Zhou *et al.* 2022. Adversarial robustness of deep code comment generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(4):1–30.

Terry Yue Zhuo *et al.* 2023a. Exploring ai ethics of chatgpt: A diagnostic analysis. *arXiv preprint arXiv:2301.12867*.

Terry Yue Zhuo *et al.* 2023b. On robustness of prompt-based semantic parsing with large pre-trained language model: An empirical study on codex. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 1090–1102.

Maksim Zubkov *et al.* 2022. Evaluation of contrastive learning with various code representations for code clone detection. *arXiv preprint arXiv:2206.08726*.

A Literature Selection

we employ the "Quasi-Gold Standard" (QGS) (Zhang *et al.*, 2011) approach for paper search. We conduct a manual search to identify a set of relevant studies and extracted a search string from them. This search string is then used to perform an automated search, and subsequently,

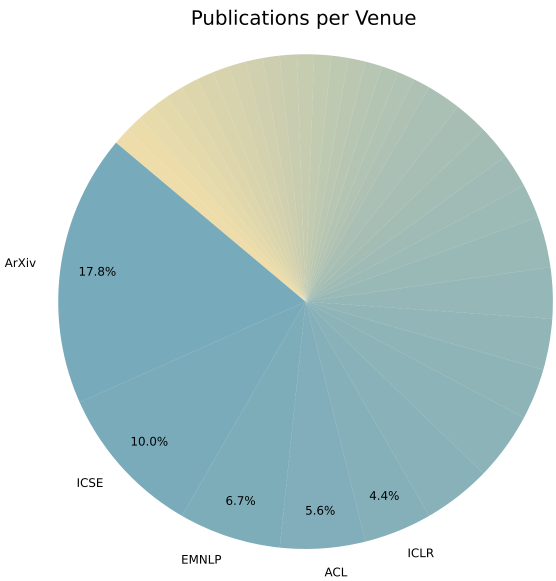

Figure 2: Venue Distribution of the collected publications.

a snowballing search is employed to further supplement the search results. This approach ensures both search efficiency and maximum coverage, minimizing the risk of omission.

During the manual search, we manually verify the papers containing two sets of keywords: one pertaining to software engineering, and the other related to deep learning. The complete set of search keywords is as follows:

- *Keywords related to software engineering*: Program Transformation, Robustness, Adversarial Robustness, Adversarial Attack.
- *Keywords related to deep learning*: Code Model, Code Language Model, Data Augmentation, Augmented, Contrastive Learning, Low Resource.

To this end, we have compiled a list of 89 core papers from the past five years, mainly from premier conferences and journals in both the ML and SE disciplines as shown in Figure 2 (with 62 out of 89 papers published in Core Rank A/A* venues[3]).

B Background

B.1 What are source code models?

Source code models are trained on large-scale corpora of source code and therefore able to model the contextual representations of given code snippets (Allamanis *et al.*, 2017). In the early stage, researchers have attempted to leverage deep learning architectures like LSTM (Gu *et al.*, 2016) and Seq2Seq (Yin and Neubig, 2017) to model the source code like plain text, and shown that these models can achieve great performance on specific downstream tasks of source code. With the development of pre-trained language models in NLP, many pre-trained source code models are proposed to enhance the source code representations and efficiently be scaled to any downstream tasks (Feng *et al.*, 2020; Guo *et al.*, 2021; Nijkamp *et al.*, 2023). Some of these models incorporate the inherent structure of code. For example, instead of taking the syntactic-level structure of source code like ASTs, Guo *et al.* (2021) consider program data flow in the pre-training stage, which is a semantic-level structure of code that encodes the relation of "where-the-value-comes-from" between variables. In this survey, we focus DA methods designed for all the deep-learning-based source code models.

B.2 What is data augmentation?

Data augmentation (DA) techniques aim to improve the model's performance in terms of various aspects (e.g., accuracy and robustness) via increasing training example diversity with data synthesis. Besides, DA techniques can help avoid model overfitting in the training stage, which maintains the generability of the model. In CV, DA techniques with predefined rules are commonly adopted when training models, such as image cropping, image flipping, and color jittering (Shorten and Khoshgoftaar, 2019). These techniques can be classified as *rule-based* DA. Furthermore, some attempts like Mixup have been made to create new examples by fusing multiple examples together, which is categorized as *example interpolation* DA. Compared to CV, DA techniques for NLP greatly rely on language models that can help paraphrase the given context by word replacing or sentence rewriting (Feng *et al.*, 2021). As most of these language models are pre-trained and can capture the semantics of inputs, they serve as reasonable frameworks to modify or paraphrase the plain text. We denote such DA methods as *model-based* DA.

B.3 How does data augmentation work in source code?

Compared to images and plain texts, source code is less flexible to be augmented due to the nature

[3] We refer to the venues listed at `http://portal.core.edu.au/conf-ranks/` and `http://portal.core.edu.au/jnl-ranks/`.

of strict programming syntactic rules. Hence, we observe that most DA approaches in source code must follow the predetermined transformation rules in order to preserve the functionality and syntax of the original code snippets. To enable the complex processing of the given source code, a common approach is to use a parser to build a concrete syntax tree from the code, which represents the program grammar in a tree-like form. The concrete syntax tree will be further transformed into an abstract syntax tree (AST) to simplify the representation but maintain the key information such as identifiers, if-else statements, and loop conditions. The parsed information is utilized as the basis of the *rule-based* DA approaches for identifier replacement and statement rewrite (Quiring *et al.*, 2019). From a software engineering perspective, these DA approaches can emulate more diverse code representation in real-world scenarios and thus make source code models more robust by training with the augmented data (Yefet *et al.*, 2020).

C More Scenarios

C.1 Method Name Prediction

The goal of method name prediction is to predict the name of a method given the program. Yefet *et al.* (2020) attack and defense source code models by using variable-name-replaced adversarial programs on the *Code2Seq* dataset (Alon *et al.*, 2019). Pour *et al.* (2021) propose a search-based testing framework specifically for adversarial robustness. They generate adversarial examples of Java with ten popular refactoring operators widely used in Java. (Rabin *et al.*, 2021) and (Yu *et al.*, 2022) both implement data augmentation frameworks and various transformation rules for processing Java source code on the *Code2Seq* dataset.

C.2 Type Prediction

Type prediction, or type interference, aims to predict parameter and function types in programs. Bielik and Vechev (2020) conduct adversarial attacks on source code models with examples of transformed ASTs. They instantiate the attack to type prediction on JavaScript and TypeScript. Jain *et al.* (2021) apply compiler transforms to generates many variants of programs in DeepTyper (Hellendoorn *et al.*, 2018), with equivalent functionality with 11 rules. Li *et al.* (2022d) incorporate srcML (Collard *et al.*, 2013) meta-grammar embeddings to augment the syntactic features of examples in three datasets, *DeepTyper*, *Typilus Data* and *CodeSearchNet*.

C.3 Code Question Answering (CQA)

CQA can be formulated as a task where the source code models are required to generate a textual answer based on a given code snippet and a question. Huang *et al.* (2021) incorporate two rule-based DA methods on code and text to create examples for contrastive learning. Li *et al.* (2022b) explore the efficacy of adversarial training on the continuous embedding space with rule-based DA on *CodeQA*, a free-form CQA dataset. Park *et al.* (2023) evaluate KeyDAC, a framework using query writing and variable renaming as DA, on *WebQueryTest* of CodeXGLUE. Different from *CodeQA*, *WebQueryTest* is a CQA benchmark only containing Yes/No questions.

C.4 Code Classification

The task performs the categorization of programs regarding their functionality or readability. Wang *et al.* (2022) propose a novel AST hierarchy representation for contrastive learning with the graph neural network. Specifically, they augment the node embeddings in AST paths on *OJ*, a dataset containing 104 classes of programs. Zhang *et al.* (2022) incorporate simplex interpolation, an example-interpolation DA approach on IR, to create intermediate embeddings on *POJ-104* from CodeXGLUE. Dong *et al.* (2023b) also explore the example-interpolation DA to fuse the embeddings of code snippets. They evaluate the method on two datasets, *JAVA250* and *Python800*.

D More Challenges and Opportunities

Working with domain-specific data. Our paper focuses on surveying DA techniques for common downstream tasks involving processing source code. However, we are aware that there are a few works on other task-specific data in the field of source code. For instance, API recommendation and API sequence generation can be considered a part of source code tasks (Huang *et al.*, 2018; Gu *et al.*, 2016). DA methods covered by our survey can not be directly generalized to these tasks, as most of them only target program-level augmentation but not API-level. We observe a gap of DA techniques between these two different layers (Treude and Robillard, 2016; Xu *et al.*, 2020; Wang *et al.*, 2021b), which provides opportunities for future works to explore. Additionally, the source code modeling has not fully justified DA for out-of-distribution generalization. Previous studies (Hajipour *et al.*, 2022; Hu *et al.*, 2022) assume the domain as the programs with different complexity, syntax, and semantics. We argue that this definition is not natural enough. Similar to the subdomains in NLP, like biomedical and financial texts, the application subdomains of source code can be diverse. For example, the programs to solve data science problems can significantly differ from those for web design. We encourage SE and ML communities to study the benefits of DA when applied to various application subdomains of source code.

Mitigating social bias. As source code models have advanced software development, they may be used to develop human-centric applications such as human resources and education, where biased programs may result in unjustified and unethical decisions for underrepresented people (Zhuo *et al.*, 2023a). While social bias in NLP has been well studied and can be mitigated with DA (Feng *et al.*, 2021), the social bias in source code has not been brought to attention. For example, Liu *et al.* (2023c) find that LLMs have severe biases in various demographics such as gender, sexuality, and occupation when performing code generation based on the natural language queries. To make these models more responsible in source code, we urge more research on mitigating bias. As prior works in NLP suggested, DA may be an effective technique to make source code models more responsible.

Few-shot learning. In few-shot scenarios, models are required to achieve performance that rivals that of traditional machine learning models, yet the amount of training data is extremely limited. DA methods provide a direct solution to the problem. However, limited works in few-shot scenarios have adopted DA methods (Nashid *et al.*, 2023). Mainstream pre-trained source code models obtain rich semantic knowledge through language modeling. Such knowledge even covers, to some extent, the semantic information introduced by traditional paraphrasing-based DA methods. In other words, the improvement space that traditional DA methods bring to pre-trained source code models has been greatly compressed. Therefore, it is an interesting question how to provide models with fast generalization and problem-solving capability by generating high-quality augmented data in few-shot scenarios.

Multimodal applications. It is important to note that the emphasis on function-level code snippets does not accurately represent the intricacies and complexities of real-world programming situations. In such scenarios, developers often work with multiple files and folders simultaneously.s have also been developed. Wang *et al.* (2021b) and Liu *et al.* (2023a) explore the chart derendering with an emphasis on source code and corresponding APIs. Surís *et al.* (2023) propose a framework to generate Python programs to solve complex visual tasks including images and videos. Although such multimodal applications are more and more popular, no study has yet been conducted on applying DA methods to them. A potential challenge for the multimodal source code task technique is to effectively bridge between the embedding representations for each modality in source code models, which has been investigated in vision-language multimodal tasks (Ray *et al.*, 2019; Tang *et al.*, 2020; Hao *et al.*, 2023).