An Exploratory Literature Study on Sharing and Energy Use of Language Models for Source Code

Max Hort
Simula Research Laboratory
Oslo, Norway
maxh@simula.no

Anastasiia Grishina
Simula Research Laboratory &
University of Oslo
Oslo, Norway
anastasiia@simula.no

Leon Moonen
Simula Research Laboratory &
BI Norwegian Business School
Oslo, Norway
leon.moonen@computer.org

***Abstract*—CONTEXT: Large language models trained on source code can support a variety of software development tasks, such as code recommendation and program repair. Large amounts of data for training such models benefit the models' performance. However, the size of the data and models results in long training times and high energy consumption. While publishing source code allows for replicability, users need to repeat the expensive training process if models are not shared.**

GOALS: The main goal of the study is to investigate if publications that trained language models for software engineering (SE) tasks share source code and trained artifacts. The second goal is to analyze the transparency on training energy usage.

METHODS: We perform a snowballing-based literature search to find publications on language models for source code, and analyze their reusability from a sustainability standpoint.

RESULTS: From a total of 494 unique publications, we identified 293 relevant publications that use language models to address code-related tasks. Among them, 27% (79 out of 293) make artifacts available for reuse. This can be in the form of tools or IDE plugins designed for specific tasks or task-agnostic models that can be fine-tuned for a variety of downstream tasks. Moreover, we collect insights on the hardware used for model training, as well as training time, which together determine the energy consumption of the development process.

CONCLUSION: We find that there are deficiencies in the sharing of information and artifacts for current studies on source code models for software engineering tasks, with 40% of the surveyed papers not sharing source code or trained artifacts. We recommend the sharing of source code as well as trained artifacts, to enable sustainable reproducibility. Moreover, comprehensive information on training times and hardware configurations should be shared for transparency on a model's carbon footprint.

***Index Terms*—sustainability, reuse, replication, energy, DL4SE.**

I. INTRODUCTION

The FAIR data principles are designed to support and enhance the reusability of digital research objects following four guiding principles: to be findable, accessible, interoperable, and reusable [1]. While the initial focus of FAIR was on scientific data, the principles have been transferred to research software [2]. Publishing source code supports the replicability of software but may incur repeated training costs, if a software product is data-driven. Training costs can be especially high for tools that are trained on large amounts of data, such as Machine Learning (ML) models, which have achieved state-of-the-art performance in various disciplines (e.g., text and image understanding, video content prediction [3, 4]). In particular, Deep Learning (DL) often achieves performance improvements by increasing the amount of training data and the size of the model, leading to long training times and substantial energy consumption [5], with an increase in computational costs for state-of-the-art models by a factor of 300 000 between 2012 and 2018 [6, 7]. This trend not only raises barriers for researchers with limited computational resources [8], it is also harmful to the environment [5, 6].

One class of DL models that benefit from training on large amounts of data are Large Language Models (LLMs). LLMs have been able to learn semantic information via training on texts from the Internet and achieve high performance on Natural Language Processing (NLP) tasks [5, 9]. Similarly, by training language models on a large corpus of source code (e.g., as provided by GitHub[1]), one can learn semantic information of source code [10] and apply the models on SE tasks, such as code generation, bug prediction and fixing, to alleviate developers from tedious work [11]. This research area is referred to as DL4SE, and the models are referred to as *Source Code Models* (SCMs).

Training an SCM can take more than 100 days and incur high costs from hardware and energy requirements [12, 13]. From an energy usage point of view, only sharing the source code to train the model is wasteful, because replication or reuse requires repeating the expensive and energy-consuming training process. Instead, trained models should be considered *digital artifacts* that must be shared to lower the bar for building on existing work [14]. For instance, fine-tuning an existing task-agnostic model requires only a fraction of the computational costs of training such a model from scratch [12].

Despite the benefits of sharing the trained models and source code, a large number of studies in DL, including many in DL4SE, do not make code or models publicly available. Liu et al. [15] surveyed deep learning studies in SE conferences and journals. They found that 74.2% of the studies did not share the source code and data for replication. Failing to share the data or trained artifacts contradicts the software sustainability-quality characteristics [16]. Software sustainability is defined from economic, environmental, social and technical dimen-

[1] www.github.com

Accepted for publication in the 17th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2023).

sions in that software should generate economic value, enable equal and sustained access to social resources, minimize harm to the environment, and ensure technical improvements and maintainability [17]. In this study, we focus on the technical and environmental aspects of sustainability in software, namely reusability and efficiency [16, 18]. To investigate the reusability and resource efficiency of source code models, we perform an exploratory literature search of existing DL4SE publications. For each publication, we investigate whether code and trained models are available, and what the training and energy requirements are. In other words, we focus on the following two research questions:

RQ1: How many DL4SE publications share source code and/or trained models or related trained artifacts?

RQ2: How much energy was used to train these models?

Contributions: The contributions of this paper include:

- ⋆ We conduct an exploratory study on the sustainability and reusability of (large) language models for source code. We analyze to what extent publications make trained artifacts available, so that software developers and researchers can reuse and profit from large models trained with high energy consumption without incurring such training costs themselves.
- ⋆ We investigate the information provided in 79 publications with shared artifacts;
- ⋆ We estimate the energy needed for training models from 30 publications that provided sufficient information;
- ⋆ We summarize the lessons learned while studying the academic literature with this focus on sustainability;
- ⋆ We provide recommendations to help researchers make their models more sustainable and support clearly communicating the relevant aspects in their publications.

II. Related Work

A. Sustainable Software Engineering

Sustainable software engineering addresses sustainability in two regards: (1) creating software that empowers sustainable applications and (2) creating software in a sustainable resource-efficient way. The former is referred to as Information Technology for Green (IT for Green) [19] or sustainability *BY* software [16]. The latter is called Green IT [20] or sustainability *IN* software [16]. In this study, we focus on sustainability *IN* software and use the term *sustainable software* or *sustainable software engineering* to define reusable shared software that is built with resource usage considerations in mind. Development of sustainable software can be supported by integrating sustainability goals in the development process [20]. One way to improve the sustainability of software is to optimize its performance by refactoring the source code, which can have positive impacts on the accompanying energy consumption [21]. For example, Verdecchia et al. [22] showed that refactoring code smells in Java applications can reduce energy consumption by almost 50%.

In addition to observing the energy consumed by applying software and potential positive effects by providing sustainable solutions, the energy consumed during the development process is of relevance as well, as pointed out by the GREENSOFT model [23]. The GREENSOFT model presents a life cycle for software products. Accordingly, a green software product should be sustainable during the course of the life cycle, including the software engineering process and the tasks developers address during implementation and maintenance. To alleviate their workload, they can use tools to automate and support software engineering tasks. In this regard, Martinez et al. [24] addressed the field of green software research by measuring energy consumption induced by development and maintenance activities, in particular Automated Program Repair (APR). APR is used to fix software bugs, which usually incur a high monetary cost to resolve, without requiring manual intervention of developers. While APR tools tend to report their performance in terms of number of bugs they are able to fix, Martinez et al. [24] considered their energy consumption as an additional quality measure. To evaluate the trade-off between accuracy and energy consumption of APR tools, they computed the energy cost for each point of accuracy (i.e., energy consumption divided by accuracy).

B. Energy Consumption of Machine Learning Models

The energy consumption of training and developing ML models is becoming a growing concern [25], with models requiring large amounts of computational resources to train, causing financial costs and CO_2 emissions [7, 26]. Recently, implementation challenges and leaderboards have been introduced to incentivize the development of energy efficient models [27, 28]. Another proposition is to measure the performance of ML models not only with regard to accuracy, but also to consider energy consumption and trade-offs between the two metrics.

To account for the sustainability–accuracy trade-off, Gutiérrez et al. [29] analyzed the impact of changing solvers for ML models. Having applied the models to credit card fraud data, they found configurations that required 2.9x more energy while improving accuracy by only 0.016. This illustrates that developers can make trade-offs between energy consumption and ML quality measures, such as precision and recall. In the same line of research, Georgiou et al. [7] compared the energy consumption of two frameworks (TensorFlow, PyTorch) for the development of DL by implementing and comparing the performance of six machine learning models. Energy consumption varied significantly in both the training and inference stages, with TensorFlow requiring less energy for training and PyTorch less energy for inference. However, the framework documentation did not provide information on hardware specifications to allow developers to select models and frameworks with regard to energy requirements.

Verdecchia et al. [25] modified the underlying datasets for training DL models to reduce energy consumption during training. Results showed that reducing dataset size, either in the number of features or number of data points, improves the energy efficiency by up to 92%, while having a negligible effect on accuracy reduction for selected algorithms. Garcia-Martin et al. [30] investigated the impact of parameter tuning on energy consumption and accuracy for the Very Fast

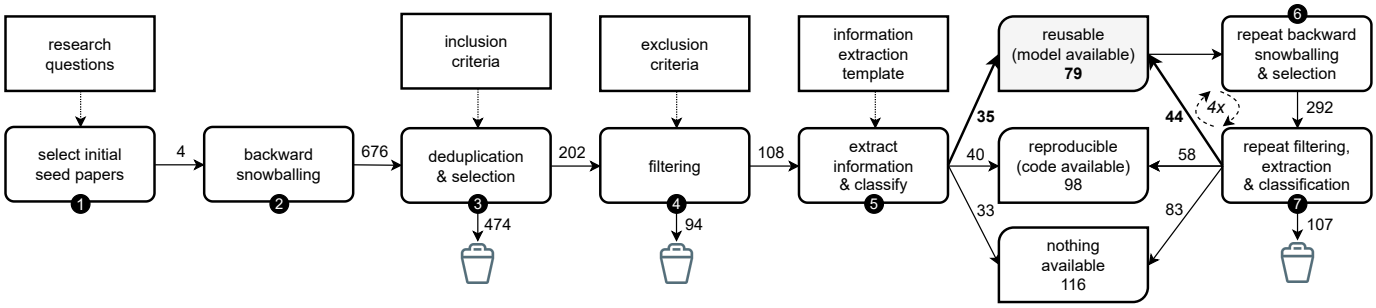

Fig. 1. Overview of the search procedure.

Decision Tree algorithm. In some cases, small reductions in accuracy (< 0.1) can reduce energy consumption by more than 70%. For an overview of publications addressing Green AI (AI systems developed with sustainability and costs considered), we refer to the systematic review by Verdecchia et al. [31].

C. Energy Consumption of Large Language Models

To support responsible NLP, Zhou et al. [12] proposed the platform HULK for benchmarking pre-trained language models in terms of time and cost. Processing time and costs are measured according to cloud services' hardware specifications and resource consumption. The cost of NLP models is evaluated at three stages: pre-training, fine-tuning, and inference. Pre-training is the most expensive stage in the development of language models: it can take several days and can cost up to 75,000$. However, once pre-trained, a model can be fine-tuned for several tasks, which requires less computational resources.

Strubell et al. [5] provided insights on the financial and environmental costs of training large language models for NLP tasks. In particular, they estimate the training cost in USD and carbon emissions for four open-source models. For example, training large language models with neural architecture search can cause CO_2 emissions 17 times as high as the average per-capita consumption in America. Given the high cost of NLP models, Strubell et al. formulated three actionable recommendations: (1) authors should report training times to allow for a cost-benefit analysis rather to solely focus on accuracy; (2) researchers need equal access to computational resources; (3) efficient hardware and algorithms should be prioritized.

III. LITERATURE SEARCH

To find relevant literature, we adopt and adapt a snowballing search procedure [32–34]. We make small adjustments to the search procedure described by Wohlin et al. [33], as we aim to build on four recent surveys in the domain of deep learning models for software engineering tasks. The surveys examine different research questions than ours but consider the same domain, which makes them good starting points. Moreover, we apply the inclusion and exclusion criteria after each snowballing step to control the scope of the search, as it can quickly become too wide and cover all of SE. Figure 1 presents an overview of the search procedure and the number of publications collected. The search and subsequent information extraction were conducted by the first two authors; the third author helped mitigate classification discrepancies where needed. In step ❶, we select the four survey papers that seed the study. We include both published work and arXiv preprints to ensure timeliness. The four seed surveys are:

- Chen and Monperrus [35]: A literature study on embeddings learned from source code. Embeddings have been trained on different levels of granularity (e.g., binary code, tokens, functions). A list of 21 publicly available embeddings is provided.
- Sharma et al. [36]: A survey of ML techniques for analysing source code. A total of 364 studies published from 2002–2021, divided over 12 SE tasks. For each task, data collection, feature extraction and model training stages are outlined. They listed 61 tools for analyzing source code and applying ML techniques.
- Watson et al. [37]: A literature review of deep learning approaches in SE research. A total of 128 deep learning publications spanning 23 SE tasks have been reviewed.
- Niu et al. [38]: A survey on pre-trained models on source code applied to SE tasks. They presented a total of 20 pre-trained code models that have been applied to 18 tasks.

These four initial studies contain references to a total of 676 publications (step ❷). After deduplication, we consider a total of 202 unique publications for further investigation based on their title (step ❸). We deem a paper of interest for further analysis if the title matches the following inclusion criteria:

IC-1: the publication addresses an SE task, and
IC-2: the publication applies a deep learning technique.

To filter relevant publications, we read all 202 publications, published from 2012-2022, and flag publications for exclusion if they did not train language models for source code, i.e., the exclusion criterion for step ❹ is:

EC-1: the publication does not train a source code model.

This step leaves us with 108 publications for further analysis.

In step ❺, we extract information from the 108 publications to determine how they share artifacts. First, we investigated if source code is available. For this purpose, we analyzed the respective publications for links or references to external sources (e.g., a GitHub repository for source code, or Zenodo for datasets and tools). Among the 108 publications, 33 publi-

TABLE I
DESCRIPTION OF SOFTWARE ENGINEERING TASKS. TYPE ILLUSTRATES THE INPUT AND OUTPUT OF THE RESPECTIVE TASKS (NL = NATURAL LANGUAGE, PL = PROGRAMMING LANGUAGE, V = EXTRACTED OR PREDICTED VALUE).

Task	Type	Description
Inference	PL → V	Predict code properties (e.g., variable names, data types, code authors).
Code summarization	PL→ NL	Summarize source code snippets in natural language.
Code search	NL→ PL	Search for code snippets given a description.
Code completion	PL→ PL	Recommend likely tokens for a code sequence.
Default detection	PL→ V	Determine whether a code snippet is faulty.
Documentation generation	PL→ NL	Generate (e.g., comments, commit messages, docstrings)
Code generation	NL→ PL	Generate source code given a description (e.g., log messages, program synthesis).
Comprehension	PL x NL → V PL x NL → NL	Check if a code snippet can answer a question. Answer question about a code snippet.
Clone detection	PL→ V	Clone and similarity determination of two code snippets.
Code translation	PL → PL	Translate source code from one programming language to another.
Code repair	PL → PL	Repair a faulty code snippet.

cations did not provide source code ("unavailable" in Fig. 1).[2] Next, we determined if the shared artifacts do not only provide source code, but include fully functional tools or checkpoints for ML models that are ready to use, without the need to be trained. This was the case for 35 out of the 108 publications ("reusable" in Fig. 1). The remaining 40 publications provided source code but no trained artifacts ("reproducible" in Fig. 1).

The initial survey-based search is followed by repeated backward snowballing in steps ❻ & ❼. During snowballing, we collect additional relevant publications that have been cited by the 35 publications which provided trained artifacts collected prior. Snowballing is performed incrementally on publications that share trained artifacts, until no new publications are found (i.e., we perform multiple iterations, and stop when a fixed point is reached, which happened after four iterations). This yielded 292 additional publications (published from 2002-2023) that fit the inclusion criteria, bringing the total to 494 (202 from step ❸ and 292 from step ❻). After further inspection, 107 of the 292 additional publications did not train a language model and were excluded. Furthermore, 83 of those publications did not provide source code, and 58 of the publications shared source code but not the trained artifacts. Thus, repeated snowballing adds 44 publications that share trained artifacts, bringing the total to 79 publications with shared artifacts, published from 2015 to 2022.

We classify these 79 publications with respect to the 11 SE tasks presented in Table I, which were inspired by Niu et al. [38]. To address these tasks, source code models were trained on 18 different programming languages. The most frequent languages include Java (45 publications), Python (32 publications), and C and/or C++ (18 publications). Figure 2 presents the number of publications for each combination of programming language and SE task (e.g., an approach trained on Java source code for code completion).

We found that there are two types of trained artifacts that were publicly available: (1) trained ML models and tools; (2) source code embeddings. While trained models and tools are aimed at a specific task, source code embeddings are task-agnostic and provide comprehensive code representations for training future models with less effort than generating pre-trained embeddings [12]. Section IV (task-specific tools) and Section V (task-agnostic embeddings) present detailed information for the two types of shared artifacts.

[2] To ensure that no artifacts were overlooked, we performed additional Google searches for publications that did not mention artifacts for replication.

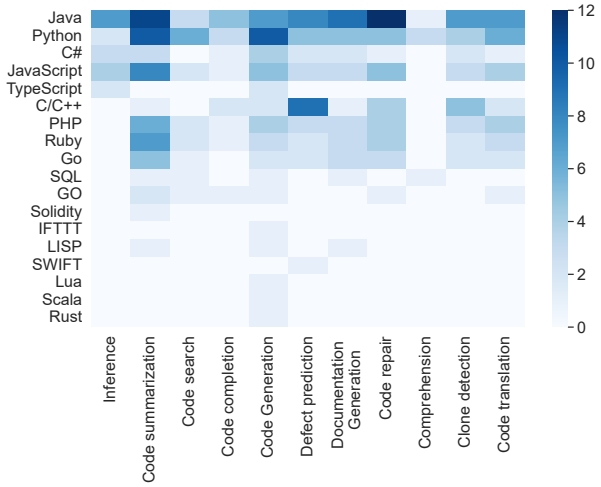

Fig. 2. Number of publications with shared artifacts for combinations of task and programming language.

Answer to RQ1: Out of the reviewed 293 publications, 33% shared source code and 27% shared trained artifacts.

IV. TASK-SPECIFIC CODE MODELS

This section presents approaches with shared artifacts that are designed to address specific tasks. In total, we collected 52 task-specific publications, which are summarized in Table II. Publications are presented with regards to the task they address and their respective programming language is shown, as well as hardware configuration and training time, if provided.

Among the 52 publications, two publications shared artifacts for more than one task. Hoang et al. [57] proposed CC2Vec, an approach for representing code changes. For each of the three tasks (log message generation, bug fixing patch identification, and just-in-time defect prediction), they trained and shared a separate model. Huang et al. [63] first introduced a new dataset

TABLE II

TRAINING DETAILS FOR TASK-SPECIFIC LANGUAGE MODELS. FOR EACH MODEL, WE LIST HARDWARE DETAILS, TRAINING TIME IN $hours$ AND ESTIMATED ENERGY CONSUMPTION IN kWh, IF THE INFORMATION IS AVAILABLE. IF ONLY PARTIALLY TRAINED ARTIFACTS ARE AVAILABLE, WE APPEND THE APPROACH REFERENCE WITH †.

Task	Approach	Year	Language	Hardware	Hours	kWh
Clone detection	DeepSim [39]†	2018	Java	desktop PC: Intel i7 CPU at 4.0GHz, 4 cores; 1 NVIDIA RTX 1080 GPU	4	1.1
	FCDetector [40]†	2020	C++	server: 2.4GHz CPU, 8 cores; 1 NVIDIA GTX 1080 GPU	2	1.1
	func2vec [41]†	2018	C	AWS, Intel Xeon E5-2686 v4 (Broadwell) Processors and DDR4 Memory.	2	
	TBCCD [42]†	2019	Java, C			
Code completion	C3PO [43]	2020	C#	1 NVIDIA Tesla V100 GPU	9	4.6
	AnyCodeGen [44]	2020	Java, C++	NVIDIA Tesla V100 GPU		
	Retrieve-and-edit [45]†	2018	Python			
	Codit [46]	2022	Java			
Code generation	Intellicode compose [47]	2020	Python, C#, JavaScript, TypeScript	80 NVIDIA Tesla v100 GPUs, 32 GB	492.5	19,750.0
	Pythia [48]	2019	Python	NVIDIA Tesla V100 GPUs	30,000	15,330.0
	Bayou [49]	2018	Java	AWS, p2.xlarge: 1 NVIDIA K80 GPU, 12 GB	10	4
	Xu et al. [50]	2020	Python			
	NL2code [51]	2017	Python, IFTTT			
Code repair	CURE [52]	2021	Java	server: 1 NVIDIA TITAN V GPU, 3 NVIDIA TITAN Xp GPUs	528	448.8
	RLAssist [53]	2019	C	Intel Xeon E5-2630 v4 CPU at 2.20GHz, 32 GB	504	81.6
	TFix [54]	2021	JavaScript	8 NVIDIA RTX 2080 Ti GPUs	96	321.6
	Sequencer [55]	2019	Java	1 NVIDIA K80 GPU	1	0.6
	DeepRepair [56]	2019	Java			
	CC2Vec [57]	2020	Java, C			
	DL4PatchCorrectness [58]	2020	Java			
	RewardRepair [59]	2022	Java			
	VRepair [60]	2022	C			
	Tufano et al. [61]	2019	Java			
Code search	MP-CAT [62]	2020	Python		67	
	CoCLR [63]	2021	Python	1 NVIDIA Tesla V100 GPU, 16 GB		
	CoaCor [64]	2019	SQL			
	NCS [65]	2020	Python			
Code summarization	CAST [66]	2021	Java	server: 4 NVIDIA Tesla V100 GPUs	33	67.1
	LeClair et al. [67]	2020	Java	Xeon E1430v4 CPUs, 110 GB; 1 NVIDIA Titan RTX GPU, 1 Quadro P5000 GPU		
	Attn-to-FC [68]	2020	Python	Intel Core i7 CPU at 2.2GHz, 64GB 1600 MHz DDR; 1 NVIDIA Titan X GPU, 2GB		
	MMTrans [69]	2021	Solidity	server: 4 NVIDIA RTX 2080 Ti GPUs, 11 GB		
Code translation	TransCoder [70]	2020	Java, C++, Python	32 NVIDIA V100 GPUs		
Comprehension	CoCLR [63]	2021	Python	1 NVIDIA Tesla V100 GPU, 16 GB		
	CodeQA [71]	2021	Java, Python	3 NVIDIA 1080 Ti GPUs		
	Staqc [72]†	2018	Python, SQL			
Defect prediction	IVDetect [73]	2021	C/C++		239	
	VulBERTa [74]	2022	C/C++	pre-train: GCP VMs: 48 vCPUs, 240 GB; 2 NVIDIA Tesla A100 GPUs, 40 GB fine-tuning: 48 cores Intel Xeon Silver CPU, 292 GB; 2 NVIDIA TITAN Xp GPUs	106	91.0
	DeepWukong [75]	2021	C/C++	Intel Xeon E5-1620 CPU at 3.50GHz; 1 NVIDIA RTX 1080 Ti GPU	23	12.7
	Li et al. [76]	2019	Java		11	
	Lin et al. [77]†	2018	C	server: 2 Intel Xeon E5-2690 v3 CPUs at 2.60GHz, 96 GB	6	3.1
	FUNDED [78]†	2021	Java, C, C++, PHP, SWIFT	server: Intel Xeon CPU, 2.4GHz, 14 cores; 1 NVIDIA 2080Ti GPU	2	0.9
	Deepbugs [79]†	2018	JavaScript	Intel Xeon E5-2650 CPU, 48 cores, 64 GB; 1 NVIDIA Tesla P100 GPU	1	1.4
	REVEAL [80]	2020	C/C++	16 Intel Xeon CPUs at 2.60GHz, 252 GB; 1 NVIDIA RTX 1080 Ti GPU		
	CC2Vec [57]	2020	Java, C			
Documentation generation	CUP [81]	2020	Java	Intel Xeon CPU at 2.7GHz, 40 cores	290	168.5
	CC2Vec [57]	2020	Java, C			
	DEEP JIT [82]†	2020	Java			
	DeepCommenter [83]	2020	Java			
Inference	PigeonJS [84]†	2018	JavaScript, Java, Python, C#		100	
	Type4Py [85]	2022	Python	AMD Ryzen Threadripper 1920X with 24 threads at 3.5GHz, 64 GB 2 NVIDIA RTX 2080 TI GPUs	100	2.0
	JS NICE [86]	2015	JavaScript	4 Xeon CPUs at 2.13GHz, 32 cores	10	
	NL2Type [87]	2019	JavaScript	Intel Xeon E5-2650 CPU, 48 cores, 64 GB; 1 NVIDIA Tesla P100 GPU, 16 GB	4	5.7
	DeepTyper [88]	2018	TypeScript	Intel i7-8700 CPU, 6 cores, 32 GB; 1 NVIDIA RTX 1080 Ti GPU		
	Lambdanet [89]	2020	TypeScript			
	NATURALIZE [90]†	2015	Java			

called CoSQA, consisting of 20,604 human-annotated labels for natural language and source code pairs. Additionally, they proposed a model, CoCLR, trained on two tasks: code search and question answering. Their GitHub repository provides model checkpoints for both of these tasks.

The most frequently addressed tasks are concerned with faulty programs: code repair and defect prediction. Ten publications proposed approaches for code repair and nine publications addressed defect prediction. The task with the fewest available artifacts is code translation. Only Lachaux et al. [70] shared their TransCoder models for translating between three programming languages (Java, C++, Python). To allow for the translation of each pair of languages, they shared two models: 1) translate C++ $\rightarrow$ Java, Java $\rightarrow$ C++, Java $\rightarrow$ Python; 2) C++ $\rightarrow$ Python, Python $\rightarrow$ C++, Python $\rightarrow$ Java.

The most popular programming languages, among 11 unique languages considered by the 52 publications, are Java (23 out of 52 publications), C/C++ (14 out of 52 publications), and Python (14 out of 52 publications). In detail, 42 publications considered one programming language, while ten publications were applied to more than one language: six publications considered two programming languages, one

TABLE III
TRAINING DETAILS FOR TASK-AGNOSTIC LANGUAGE MODELS. FOR EACH MODEL, WE LIST HARDWARE DETAILS, TRAINING TIME IN $hours$ AND ESTIMATED ENERGY CONSUMPTION IN kWh, IF THE INFORMATION IS AVAILABLE.

Approach	Year	Language	Hardware	Time in hours	kWh
BLOOM [13]	2022	Java, PHP, C++, Python, JavaScript, C#, Ruby, Lua, TypeScript, GO, C, Scala, Rust	server: 384 NVIDIA A100 GPUs, 80 GB	1,082,990	433,196
Prophetnet-x [91]	2021	Go, Java, JS, Php, Python, Ruby	NVIDIA Tesla V100 GPUs	30,000	15,330
CodeBERT [92]	2020	Python, Java, JavaScript, PHP, Ruby, Go	server: 16 NVIDIA Tesla V100 GPUs, 32 GB	1,320	10,610
Dobf [93]	2021	Java, Python	32 NVIDIA V100 GPUs	192	3,080
Codet5 [94]	2021	Ruby, JavaScript, Go, Python, Java, Php, C, C#	server/cluster: 16 NVIDIA A100 GPUs, 40 GB	288	1,930
PLBART [95]	2021	Java, Python	8 NVIDIA RTX 2080 Ti GPUs	276	925
Mastropaolo et al. [96]	2021	Java	Google Cloud, Colab: 8 TPUs, 35.5 GB memory	343	766
Graphcodebert [97]	2021	Ruby, JS, Go, Python, Java, PHP	server: 32 NVIDIA Tesla V100 GPUs, 32 GB	83	667
CodeTrans [98]	2021	Python, Java, JavaScript, PHP, Ruby, Go, C#, SQL, LISP	1 TPU v3-8	2,088	582
GREAT [99]	2022	Python	1 Tesla P100 GPU	120	51
Javabert [100]	2021	Java	3 NVIDIA Titan X GPUs, 12 GB	24	30
code2vec [101]	2019	Java	1 NVIDIA Tesla K80 GPU	36	18
OpenVocabCodeNLM [102]	2020	Java, Python, C	GPUs	336	-
GraphCode2Vec [103]	2022	Java	server: 40 CPUs@2.20GHz, 256GB; 1 NVIDIA Tesla V100 GPU	-	-
Spt-code [104]	2022	Java, Python, JavaScript, PHP, GO, Ruby	4 NIVDIA A100s9 GPUs	-	-
StructCoder [8]	2022	Java, Python, PHP, JavaScript	4 RTX 8000 GPUs, 48GB	-	-
Codex [10]	2021	Python	Azure	-	-
Cotext [105]	2021	Python, Java, JavaScript, PHP, Ruby, Go	1 TPU v2-8	-	-
CuBERT [106]	2020	Python	TPUs	-	-
TSSA [107]	2020	Java	1 NVIDIA P100 GPU, 16 GB; 1 K80 GPU, 16GB memory	-	-
CodeGPT [108]	2021	Python, Java	-	-	-
ContraCode [109]	2021	JavaScript	-	-	-
CodeTransformer [110]	2021	Python, JavaScript, Ruby, GO	-	-	-
DAMP [111]	2020	Java, C#	-	-	-
Obfuscated Code2Vec [112]	2020	Java	-	-	-
code2seq [113]	2019	Java, C#	-	-	-
Efstathiou and Spinellis [114]	2019	Java, Python, C++, C#, C, PHP	-	-	-

publication considered three languages, and three publications considered four languages. This results in an average of 1.33 programming languages considered per publication.

In addition to programming languages considered, we collect training details, such as hardware used and training time for each publication. However, those are not always provided. There are 22 out of 52 publications without hardware details (42%) and 26 out of 52 without training time (50%), 33% shared neither information (17 out of 52 publications). The training time of 26 publications with such details ranges from two hours or less [41, 55, 78, 79, 85] to hundreds of hours [47, 53]. While it is common to perform training on GPUs, there are four publications that did not use any GPU for their training procedure, published from 2015–2019 [41, 53, 77, 86]. Commonly, publications used a single GPU for training [39, 40, 43, 44, 49, 55, 63, 68, 75, 78–80, 87, 88], sometimes in combination with CPUs. The highest amount of GPUs have been used by Svyatkovskiy et al. [47]. They utilized 5 Lambda V100 boxes, with 16 V100 GPUs each, resulting in 80 GPUs.

While we focus on the training procedure and the energy associated with creating and sharing an ML model, we note the application of such models can vary highly for different SE tasks. Usually, the reported tested times are lower than the required training time (e.g., more than 100 times quicker than training [40, 75, 76]), but in particular, program repair experiments can require long testing times. For example, Chen et al. [55] applied Sequencer for 130 hours to find patches for 75 bugs. White et al. [56] applied their program repair tool DeepRepair for 2,616 days. Data extraction and preparation steps can also require considerable amounts of time and compute resources, ranging from 5-12 days [73, 78, 81].

The majority of task-specific publications provided access to the full trained models, some of which one needs to request access to [51, 76]. Moreover, there are approaches shared as online tools [44, 49, 86] or IDE extensions [47, 48, 83, 85]. There are also 12 out of 52 publications that did not share the full model, but trained embedding files, which are used by the model. These are marked in Table II with the † symbol.

V. TASK-AGNOSTIC CODE MODELS

This section presents task-agnostic code models which share means of representing source code as embeddings, for a variety of downstream tasks. These models are able to transform code snippets to embeddings, which can be fine-tuned to SE tasks. For example, Lu et al. [108] provided fine-tuning details for the CodeXGLUE benchmark, with information for task-specific training and inference time for each task.[3] The fine-tuning time ranges from 2 GPU hours (defect detection) to 60 hours (text-to-code generation, documentation translation).

In total, we collected 27 task-agnostic models, as shown in Table III. For each publication, we list the model name and the programming languages it was trained on. If available, we list details on hardware configuration and training times. Among the 27 publications, 52% did not provide training time details (14 out of 27) and 26% did not provide their hardware configurations (7 out of 27). For publications without hardware details, training time is not reported as well.

Among the publications that shared training time details, the shortest duration is found for code2vec [101], which was trained for 1.5 days and a single GPU. However, training

[3] https://microsoft.github.io/CodeXGLUE/

large models can usually take weeks, up to 87 days for CodeTrans [98] and 3.5 months for BLOOM [13]. The long training time of BLOOM can be explained by the fact that it was trained on the highest number of programming languages (13 programming languages) in addition to 46 natural languages. Thereby, BLOOM is also the model trained on the highest number of programming languages, as it was trained on 13 out 14 programming languages we observed. BLOOM was not trained on LISP, which was only considered by CodeTrans [98] On average, each task-agnostic model is trained on source code data from 3.6 programming languages. Moreover, 10 out of the 27 publications train on a single programming language, which in 6 out 10 cases is Java.

In comparison to task-specific models, task-agnostic models are trained on more programming languages, 3.6 in comparison to 1.3 programming languages on average, and require a higher computational effort. In addition, publications that provide task-agnostic models for embedding source code are more likely to share hardware configurations than publications with task-specific models. The proportion of publications without training time details is comparable for both types (50% and 52%, for task-specific and task-agnostic models, respectively). Another difference is that task-agnostic models use more sophisticated hardware for training, with each publication using either GPUs or TPUs. Only one publication considered CPUs in addition to GPUs for training [103].

VI. Discussion

To discuss the various facets of RQ2, we consider three aspects: (A) How much energy do task-specific and task-agnostic models consume? (B) To what extent do studies on source code models take sustainability concerns into account? (C) When is sharing a model more efficient than re-training?

A. Energy Usage of Task-specific vs. Task-agnostic Models

First, we perform a comparison of the energy consumed by training task-specific and task-agnostic models. For this purpose, we collect all publications that provide hardware and training time details, such that we can estimate the consumed energy in kilowatt-hours (kWh). In total, 30 publications provide sufficient information.[4]

To estimate energy consumption, we used the Green Algorithms calculator [115].[5] This calculator is designed to estimate the carbon footprint and energy needed to run algorithms based on the number and type of CPU/GPU cores, runtime, available memory, and platform run on (PC, local server, cloud). It is also possible to consider the location for training and running algorithms, because the energy mix in the grid impacts the carbon footprint. In contrast to the *Machine Learning Emissions Calculator* [116], the Green Algorithms calculator provides averaged options when details are missing (e.g., "world" if the country is unknown, "Any" CPU type if the type is not known), which is beneficial for estimating energy consumption if these details are missing. Our estimates report the energy needed in kWh with the default location set to "world", because server locations are seldom reported.

[4]Note that we did not contact authors to provide missing information.

[5] https://www.green-algorithms.org/

TABLE IV
Assumptions about missing hardware specifications made for energy usage estimation with Green Algorithms calculator. We specify assumptions made for GPU, CPU and accelerators if hardware information in a publication is incomplete.

Model (NVIDIA)	**GPU Memory Assumptions**
Any	16 GB, if missing
GTX 1080	8 GB
RTX 2080 Ti	16 GB
Tesla K80	12 GB
Tesla V100	16 GB
Titan P100	16 GB
Titan V	12 GB
Titan Xp	12 GB
Study	**Assumed Hardware Parameters**
CUP [81]	CPU: Intel Xeon E5-2697 v4, 64 GB
RLAssist [53]	CPU: 10 cores; TDP 8.5 W
DeepSim [39]	CPU: Intel Core i7-4790K, 32 GB, TDP 22 W
NL2Type [87]	CPU: TDP 11.875 W
Mastropaolo et al. [96]	TPU type v2
Type4Py [85]	CPU: 12 cores, TDP 15 W

We share energy usage estimations in the last column of Table II and Table III for task-specific and task-agnostic artifacts, respectively. Hardware specifications required by the Green Algorithms calculator are incomplete in the majority of studies considered. Most of the models are trained using a type of accelerator, such as GPU or TPU. Four studies reported cloud provider utilization, while the other studies used different server configurations. To this end, we make assumptions about the missing specifications based on the standard CPU and GPU values stated in product descriptions on web pages of Intel and NVIDIA. In case the calculator does not cover a specific CPU type, we fetch Thermal Design Power (TDP) information from the manufacturers' website, to estimate the power used per core. In addition, for publications that used both CPU and GPU for training their models, we consider both to be active during the entirety of the training time, unless stated differently. We use the specifications reported in Table IV unless stated otherwise by the publications.

Figure 3 illustrates the energy consumed for training for each of the 30 publications. Of these, 12 provided task-agnostic models and 18 task-specific models. Among the task-specific models, 5 only provided partially trained artifacts (e.g., embeddings that are used for later training), and therefore require additional training effort before usage.

Answer to RQ2-A: 30 out of 79 publications share sufficient information to estimate their energy consumption during training. Among these, the training of task-agnostic models used more sophisticated hardware (GPUs and TPUs) and required more energy.

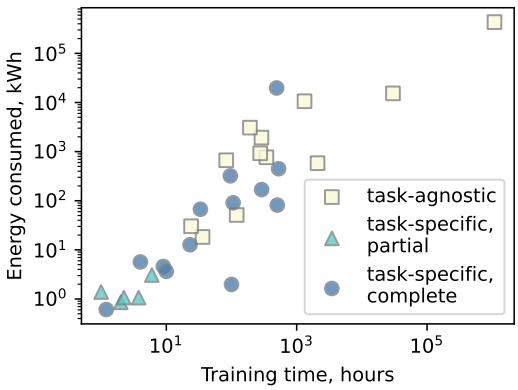

Fig. 3. Energy used for training publicly available models for code. We distinguish partially shared and fully shared task-specific models, and fully shared task-agnostic models.

B. Sustainability Concerns Considered in DL4SE Studies

In Section VI-A, we outlined publications that provided sufficient information to estimate the energy required to replicate their models (i.e., hardware and training time). While this is important to understand how high the energy requirements are, it does not illustrate whether the resource usage is sustainable, or whether sustainability was taken into account. Only in a few cases do authors consider the sustainability of the training process and the carbon footprint caused. Here, we present all three publications that, in addition to providing pre-trained artifacts, mention sustainability concerns when training. All of these three trained and provided large task-agnostic models, two of which required "hundreds of petaflop/s-days of compute" [10] or more than a million GPU hours for training [13].

Chen et al. [10] trained Codex on Azure, which purchases carbon credits and uses renewable energies to reduce the carbon footprint. Using the pre-trained Codex model for repeated inference could exceed training costs.

Wang et al. [94] stated that the experimental design followed the objective of avoiding unnecessary computation, by creating smaller-sized models in comparison to existing ones, such as Codex. Moreover, training has been conducted on the Google Cloud Platform, which purchases carbon credits to offset the 49.25kg CO_2 caused by training CodeT5.

Le Scao et al. [13] considered various sustainability aspects during the creation of BLOOM: equipment manufacturing, model training, model deployment. The 81 tons of CO_2 needed for training BLOOM can be attributed to 14% equipment manufacturing, 30% training, 55% idle energy consumption. Training benefits from France's energy grid, which uses nuclear energy in a large proportion, as a low-carbon energy source. Further details on the carbon footprint of BLOOM are provided in a dedicated study by Luccioni et al. [117].

Answer to RQ2-B: Three publications covered sustainability concerns of the training process in addition to providing trained models. For example, they used cloud providers that purchase carbon credits or calculated CO_2 emissions resulting from training the shared models.

C. When is Sharing Models More Efficient than Re-training?

In this section, we provide an exemplary scenario to compute and compare the energy required for training and storing a task-specific and task-agnostic model. We also show the energy used for downloading shared artifacts, to illustrate the energy-saving capabilities of sharing models trained on code.

In accordance with the energy estimates for the training process in Section VI-A, we used the calculator provided by Lannelongue et al. [115]. To determine the energy consumption of training and sharing language models, we followed Lakim et al. [118] who provided an assessment of the carbon footprint for the Arabic language model *Noor*. Data storage energy consumption estimates are based on the cloud storage energy consumption reported by Posani et al. [119], with a mean operating peak power of 11.3 W/TB. This measure includes a redundancy factor of 2 (i.e., an additional copy is stored) and Power Usage Effectiveness (PUE) of 1.6. Per year, this results in the energy consumption of 99 kWh per TB of data. Following the formula by Baliga et al. [120], Posani et al. [119] estimated the energy consumption of data transfers to be 23.9kJ/GB, with 1kJ being equal to 1/3600 kWh.

In Table V, we illustrate the exemplary energy consumption of sharing a tool (500 MB) and a large task-agnostic model (5 GB) over the span of one year. Note that we only consider the energy consumed by training and data storage. Other aspects, such as the manufacturing of hardware components, are omitted. Therefore, our example presents a reduced estimate of the complete energy consumption of the entire model lifecycle.

One also needs to consider the rebound effect depending on the number of downloads when estimating potential energy savings [121]. If trained models are downloaded because it is easy rather than necessary, then excess downloads can cause higher energy consumption than the initial model training caused. In our example, this is the case after 1,247 downloads for the task-specific model and 20,544 downloads for the task-agnostic model. While 20,544 downloads may sound like a large number, CodeBERT [122] was downloaded 1,982,300

TABLE V
COMPARISON OF ENERGY CONSUMED FOR TRAINING, SHARING AND DOWNLOADING PRE-TRAINED ARTIFACTS FOR EXEMPLARY TASK-SPECIFIC AND TASK-AGNOSTIC MODELS.

		Task-specific	Task-agnostic
Training	Hardware	1 GPU	8 GPUs
	Training time	10 hours	250 hours
	Memory available	64 GB	64 GB
	Platform	Local Server	Local Server
	Location	World	World
	Energy	3.74 kWh	677.95 kWh
Sharing	Model Size	500 MB	5 GB
	Upload	0.003 kWh	0.033 kWh
	Storage (1 year)	0.5 kWh	4.95 kWh
Download	100 downloads	0.33 kWh	3.32 kWh
	1,000 downloads	3.32 kWh	33.19 kWh
	10,000 downloads	33.19 kWh	331.94 kWh
	1,000,000 downloads	3,319.44 kWh	33,194.44 kWh

times from Hugging Face in January 2023.[6]

Answer to RQ2-C: A rebound effect happens when a shared model is downloaded too many times. For example, energy usage for storage and downloading of a 500 MB-size task-specific model is higher than re-training it after ca. 1,130 downloads.

VII. Threats to Validity

This section discusses the threats to validity of this mapping study based on the categories identified by Zhou et al. [123].

Internal Validity: Internal validity refers to threats to the validity of results presented in this work, for example, due to missing relevant publications during the literature search stage [124]. To mitigate this threat, we use a systematic process, starting our literature search with four comprehensive surveys on machine learning approaches for the SE domain. These provide an overview of relevant publications from 2022 and prior. Moreover, we apply four stages of snowballing to find additional references. This allows us to gather previous approaches which shared their artifacts, but there is a chance that we miss more recent works that have not been cited by any publication in our corpus, as we did not perform forward snowballing. While this can slightly alter our results, we are hopeful that recent works are more likely to share artifacts than publications from the past 10 years.

External Validity: External validity addresses the domain to which our findings can be generalized to. While our study focuses on the sustainability of shared artifacts for LLMs on code, our results confirm observations of related studies, such as high energy consumption in training LLMs for NLP tasks [5] and a lack of shared artifacts of DL studies for SE tasks [15]. Therefore, we hope our findings and recommendations are beneficial beyond LLM models for code.

Construct Validity: Construct validity is concerned with the quality of measures chosen to study the construct of interest. In our case, we were first interested in whether and which artifacts are shared (i.e., none, source code, trained models). For this purpose, we considered the absolute amount of publications with respect to the amount of shared artifacts, which coincides with the construct we want to measure.

Afterwards, we estimated the energy requirements in kWh for training language models, for which we used the Green Algorithms calculator [115]. For the validity of estimates, we assume the correctness of the calculator and the information specified in the respective publications (i.e., type of CPU/GPU and training time). If the information provided was not sufficient, we had to make choices for available memory and hardware parameters. All such choices are provided in Table IV, to make our kWh estimates reproducible.

Conclusion Validity: Conclusion validity describes whether the operations performed and obtained results in this study (e.g., literature search, data collection) can be reproduced [123]. Section III outlines our literature search procedure, starting from four existing surveys, followed by iterative snowballing steps. We list our inclusion and exclusion criteria, to allow for a reproducibility. Moreover, we provide a link to the collected publications and extracted information in the *Data Availability* section, such that our search results can be verified. To allow for the reproducibility of observations and results, we provide all the relevant extracted information in Tables II and III. When information is insufficient, we provided all assumptions over hardware specifications in Table IV.

[6] https://huggingface.co/microsoft/codebert-base

VIII. Lessons Learned

1. In general, shared information on the amount of energy consumed by the training and use of SCMs is limited (RQ1; some notable exceptions, such as BLOOM [13, 117], RQ2-B). Specifically, CPU and GPU details are missing or incomplete in the majority of papers, while they are crucial for energy usage estimation. Even with hardware details and training time available, it is hard to make accurate estimates of energy consumption and CO_2 footprint since other missing factors, such as the server location, impact the estimation (RQ2-A).

2. From the data that is shared, we see that SCMs are extremely energy-intensive to train due to computational requirements, in particular, when compared to the energy required for downloading shared artifacts (RQ2-C). It is therefore important that researchers share their artifacts (RQ1), including pre-trained and fine-tuned models, as well as explore ways to reduce their energy consumption, such as training in clouds with low CO_2 emissions (e.g., hydro-powered).

3. In general, we find that the larger the model, the higher the energy consumed for its training (RQ2-A), increasing the importance to share model artifacts to ensure sustainability.

4. Not only the energy consumption of training but also that of long-term storage of pre-trained models and datasets, as well as of their downloads should be considered (RQ2-C).

5. On the positive side, SCMs provide ample opportunities for collaborative and cooperative efforts. Sharing artifacts in the end can lead to higher sustainability than when all users would develop their models independently. More work and data is needed to be able to analyze this trade-off, which is why there is a need for a series of guidelines or a checklist to help people systematically report on the environmental/sustainability impact of their techniques.

IX. Recommendations

1. Define the scope of the research and the intended application of task-agnostic or task-specific SCMs to ensure a good understanding of the intended tasks and reuse potential.

2. Establish a set of clear and transparent metrics for energy consumption and sustainability to ensure systematic, accurate, and reliable reporting.

3. Specify details of the hardware and software configuration used for the training and inference of SCMs, including the exact types of the processors and accelerators, memory and the number of cores for CPU (e.g., Intel i7-8700 CPU, 6 cores, 32GB memory), the model and memory for GPUs (e.g., 1 NVIDIA Titan X GPU, 12GB), as well as storage media and infrastructure (RQ2-A).

4. Provide energy consumption measurements [5, 125] or estimations for both training and inference (RQ2-A). Use existing proven calculators [115, 116] and provide complete details in the paper, not just the final result, so that the computation can be repeated if an improved calculator becomes available.
5. Document the CO_2 footprint associated with energy consumption, considering energy sources and carbon offsetting applied. For cloud infrastructures, this means including the provider and region, because these details vary by location.
6. Assess other environmental impacts of SCMs, including the amount of data and storage required and the impact on the (network) infrastructure (RQ2-C).
7. Provide (and promote) open access to data and models to foster collaboration and reduce duplication of efforts, thereby reducing the energy and resource requirements for SCM development and fine-tuning.

Observe that several of these recommendations overlap with the recommendations for reproducible machine learning [126], which also cover additional aspects.

X. Conclusion

In this exploratory study, we have performed a snowballing study (i.e., four iterations of backwards snowballing) to find publications on language models for SE tasks, from which we gathered 494 publications of interest. After applying our inclusion and exclusion criteria, we are left with 293 studies, which we investigated further with regard to their reusability and sustainability (e.g., are trained artifacts shared?). We showed that there are deficiencies in the existing studies that train language models on source code regarding the transparency of sustainability aspects. Among the 293 publications, only 27% provide trained artifacts to enable the reuse of their models without incurring the same amount of training effort; 40% of the reviewed publications provide neither source code nor trained artifacts.

We collect training information from the surveyed publications, including the hardware configurations and training time. This allows us to estimate how much time and resources can be saved by reusing the artifacts or how many resources are needed to replicate the models. We have estimated the energy consumption for 30 publications that provided sufficient information (i.e., number and type of processors, training time), while only two publications provided details on energy consumption and CO_2 of the model training [13, 94].

We stress the importance of describing hardware configurations and processing times, so that even if energy consumption is not reported, one can estimate the required resources and judge whether one wants to spend effort to replicate ML models. This agrees with Bender et al. [127], who called for the research community to prioritize the environmental and financial cost of deep learning systems, by reporting or evaluating them with regard to resource usage. Optimally, if a publication creates an ML tool or model with the clear intention of its reuse, it can be beneficial to make trained artifacts available. As shown, making small tools available for download and reuse can prevent unnecessary energy consumption as opposed to training tools from scratch.

Future Work: One possible direction for future investigation is an analysis of the literature that cites the energy calculators [115, 116] mentioned earlier to assess if their use indeed leads to better communication of sustainability aspects. This could add further evidence to our recommendations.

Data Availability

To support open science and allow for replication and verification of our work, an overview of the collected publications and the extracted information is made available via Zenodo.[7]

Acknowledgements

The research presented in this paper was financially supported by the Research Council of Norway through the secureIT project (grant #288787). Max Hort is supported through the ERCIM 'Alain Bensoussan' Fellowship Programme.

References

[1] M. D. Wilkinson et al. "The FAIR Guiding Principles for Scientific Data Management and Stewardship." In: *Scientific Data* 3.1 (2016), p. 160018.
[2] A.-L. Lamprecht et al. "Towards FAIR Principles for Research Software." In: *Data Science* 3.1 (2020), pp. 37–59.
[3] C. Sun et al. "VideoBERT: A Joint Model for Video and Language Representation Learning." In: *Int'l Conf. Comp. Vision*. 2019, pp. 7464–7473.
[4] T. B. Brown et al. "Language Models Are Few-Shot Learners." In: *Int'l Conf. Neural Information Processing Sys.* Curran, 2020, pp. 1877–1901.
[5] E. Strubell et al. "Energy and Policy Considerations for Deep Learning in NLP." In: *Meeting of the Association for Computational Linguistics*. 2019, pp. 3645–3650.
[6] R. Schwartz et al. "Green AI." In: *Comm. ACM* 63.12 (2020), pp. 54–63.
[7] S. Georgiou et al. "Green AI: Do Deep Learning Frameworks Have Different Costs?" In: *Int'l Conf. Softw. Eng.* 2022, pp. 1082–1094.
[8] S. Tipirneni et al. *StructCoder: Structure-Aware Transformer for Code Generation*. 2022. arXiv: 2206.05239.
[9] T. Mikolov et al. "Distributed Representations of Words and Phrases and Their Compositionality." In: *Int'l Conf. Neural Information Processing Sys.* Curran, 2013, pp. 3111–3119.
[10] M. Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374.
[11] S. Lu et al. "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation." In: *Neural Information Processing Sys. Track on Datasets and Benchmarks*. 2021.
[12] X. Zhou et al. *HULK: An Energy Efficiency Benchmark Platform for Responsible Natural Language Processing*. 2020. arXiv: 2002.05829.
[13] T. Le Scao et al. *BLOOM: A 176B-Parameter Open-Access Multilingual Language Model*. 2022. arXiv: 2211.05100.
[14] D. S. Katz et al. "Working towards Understanding the Role of FAIR for Machine Learning." In: *Ws. Data and Research Objects Management for Linked Open Science*. 2021, pp. 1–6.
[15] C. Liu et al. "On the Reproducibility and Replicability of Deep Learning in Software Engineering." In: *ACM Trans. Softw. Eng. and Methodology* 31.1 (2021), 15:1–15:46.
[16] N. Condori-Fernandez et al. "Towards a Software Sustainability-Quality Model: Insights from a Multi-Case Study." In: *Int'l Conf. Research Challenges in Information Science*. 2019.
[17] P. Lago et al. "Framing Sustainability as a Property of Software Quality." In: *Comm. ACM* 58.10 (2015), pp. 70–78.
[18] N. Condori-Fernandez et al. *A Software Sustainability-Quality Model*. Vrije Universiteit Amsterdam, 2018.

[7] Replication package on Zenodo: https://doi.org/10.5281/zenodo.8058667.

[19] B. Tomlinson. *Greening through IT: Information Technology for Environmental Sustainability*. 2010.

[20] M. Dick et al. "Enhancing Software Engineering Processes towards Sustainable Software Product Design." In: *Integration of Environmental Information in Europe*. Ed. by K. Greve et al. Shaker Verlag, 2010.

[21] A. Hindle. "Green Mining: A Methodology of Relating Software Change and Configuration to Power Consumption." In: *Emp. Softw. Eng.* 20.2 (2015), pp. 374–409.

[22] R. Verdecchia et al. "Empirical Evaluation of the Energy Impact of Refactoring Code Smells." In: *Int'l Conf. ICT for Sustainability*. Vol. 52. 2018, pp. 365–383.

[23] S. Naumann et al. "The GREENSOFT Model: A Reference Model for Green and Sustainable Software and Its Engineering." In: *Sustainable Computing: Informatics and Systems* 1.4 (2011), pp. 294–304.

[24] M. Martinez et al. *Energy Consumption of Automated Program Repair*. 2022. arXiv: 2211.12104.

[25] R. Verdecchia et al. "Data-Centric Green AI: An Exploratory Empirical Study." In: *Int'l Conf. ICT for Sustainability*. 2022, pp. 1–11.

[26] E. García-Martín et al. "Estimation of Energy Consumption in Machine Learning." In: *J. Parallel and Distributed Computing* 134 (2019), pp. 75–88.

[27] D. Li et al. "Evaluating the Energy Efficiency of Deep Convolutional Neural Networks on CPUs and GPUs." In: *IEEE Int'l Conf.s on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)*. IEEE, 2016, pp. 477–484.

[28] P. Henderson et al. "Towards the Systematic Reporting of the Energy and Carbon Footprints of Machine Learning." In: *J. Machine Learning Research* 21.1 (2022), 248:10039–248:10081.

[29] M. Gutierrez et al. "Analysing the Energy Impact of Different Optimisations for Machine Learning Models." In: *Int'l Conf. ICT for Sustainability*. IEEE, 2022, pp. 46–52.

[30] E. Garcia-Martin et al. "Identification of Energy Hotspots: A Case Study of the Very Fast Decision Tree." In: *Green, Pervasive, and Cloud Computing*. Ed. by M. H. A. Au et al. Vol. 10232. Springer International Publishing, 2017, pp. 267–281.

[31] R. Verdecchia et al. *A Systematic Review of Green AI*. 2023. arXiv: 2301.11047.

[32] S. Jalali et al. "Systematic Literature Studies: Database Searches vs. Backward Snowballing." In: *Int'l Symp. Empirical Softw. Eng. and Measurement*. ACM, 2012, pp. 29–38.

[33] C. Wohlin. "Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering." In: *Int'l Conf. Evaluation and Assessment in Softw. Eng.* ACM, 2014, pp. 1–10.

[34] K. Petersen et al. "Guidelines for Conducting Systematic Mapping Studies in Software Engineering: An Update." In: *Information and Softw. Technology* 64 (2015), pp. 1–18.

[35] Z. Chen et al. *A Literature Study of Embeddings on Source Code*. 2019. arXiv: 1904.03061.

[36] T. Sharma et al. *A Survey on Machine Learning Techniques for Source Code Analysis*. 2021. arXiv: 2110.09610.

[37] C. Watson et al. "A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research." In: *ACM Trans. Softw. Eng. and Methodology* 31.2 (2022), 32:1–32:58.

[38] C. Niu et al. *Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code*. 2022. arXiv: 2205.11739.

[39] G. Zhao et al. "DeepSim: Deep Learning Code Functional Similarity." In: *ACM J. Meeting Eur. Softw. Eng. Conf. and Symp. Found. Softw. Eng.* ACM, 2018, pp. 141–151.

[40] C. Fang et al. "Functional Code Clone Detection with Syntax and Semantics Fusion Learning." In: *ACM SIGSOFT Int'l Symp. Softw. Testing and Analysis*. ACM, 2020, pp. 516–527.

[41] D. DeFreez et al. *Path-Based Function Embedding and Its Application to Specification Mining*. 2018. arXiv: 1802.07779.

[42] H. Yu et al. "Neural Detection of Semantic Code Clones Via Tree-Based Convolution." In: *IEEE/ACM 27th Int'l Conf. Program Comprehension*. IEEE, 2019, pp. 70–80.

[43] S. Brody et al. *A Structural Model for Contextual Code Changes*. 2020. arXiv: 2005.13209.

[44] U. Alon et al. "Structural Language Models of Code." In: *Int'l Conf. Machine Learning*. PMLR, 2020, pp. 245–256.

[45] T. B. Hashimoto et al. "A Retrieve-and-Edit Framework for Predicting Structured Outputs." In: *Int'l Conf. Neural Information Processing Sys.* Curran, 2018, pp. 10073–10083.

[46] S. Chakraborty et al. "CODIT: Code Editing With Tree-Based Neural Models." In: *IEEE Trans. Softw. Eng.* 48.4 (2022), pp. 1385–1399.

[47] A. Svyatkovskiy et al. *IntelliCode Compose: Code Generation Using Transformer*. 2020. arXiv: 2005.08025.

[48] A. Svyatkovskiy et al. "Pythia: AI-assisted Code Completion System." In: *ACM SIGKDD Int'l Conf. Knowledge Discovery & Data Mining*. ACM, 2019, pp. 2727–2735.

[49] V. Murali et al. *Neural Sketch Learning for Conditional Program Generation*. 2018. arXiv: 1703.05698.

[50] F. F. Xu et al. "Incorporating External Knowledge through Pre-training for Natural Language to Code Generation." In: *Annual Meeting of the Association for Computational Linguistics*. ACL, 2020, pp. 6045–6052.

[51] P. Yin et al. *A Syntactic Neural Model for General-Purpose Code Generation*. 2017. arXiv: 1704.01696.

[52] N. Jiang et al. "CURE: Code-Aware Neural Machine Translation for Automatic Program Repair." In: *IEEE/ACM 43rd Int'l Conf. Softw. Eng.* 2021, pp. 1161–1173.

[53] R. Gupta et al. "Deep Reinforcement Learning for Syntactic Error Repair in Student Programs." In: *AAAI Conf. Artificial Intelligence*. Vol. 33. 2019, pp. 930–937.

[54] B. Berabi et al. "TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer." In: *Int'l Conf. Machine Learning*. Vol. 139. PMLR, 2021, pp. 780–791.

[55] Z. Chen et al. "SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair." In: *IEEE Trans. Softw. Eng.* (2019).

[56] M. White et al. "Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities." In: *Int'l Conf. Softw. Analysis, Evolution and Reengineering*. 2019, pp. 479–490.

[57] T. Hoang et al. "CC2Vec: Distributed Representations of Code Changes." In: *Int'l Conf. Softw. Eng.* 2020, pp. 518–529.

[58] H. Tian et al. "Evaluating Representation Learning of Code Changes for Predicting Patch Correctness in Program Repair." In: *IEEE/ACM Int'l Conf. Autom. Softw. Eng.* ACM, 2020, pp. 981–992.

[59] H. Ye et al. "Neural Program Repair with Execution-Based Backpropagation." In: *Int'l Conf. Softw. Eng.* ACM, 2022, pp. 1506–1518.

[60] Z. Chen et al. "Neural Transfer Learning for Repairing Security Vulnerabilities in C Code." In: *IEEE Trans. Softw. Eng.* 49.1 (2023), pp. 147–165.

[61] M. Tufano et al. "On Learning Meaningful Code Changes Via Neural Machine Translation." In: *Int'l Conf. Softw. Eng.* 2019, pp. 25–36.

[62] R. Haldar et al. "A Multi-Perspective Architecture for Semantic Code Search." In: *Annual Meeting of the Association for Computational Linguistics*. ACL, 2020, pp. 8563–8568.

[63] J. Huang et al. *CoSQA: 20,000+ Web Queries for Code Search and Question Answering*. 2021. arXiv: 2105.13239.

[64] Z. Yao et al. "CoaCor: Code Annotation for Code Retrieval with Reinforcement Learning." In: *Int'l World Wide Web Conf.* ACM, 2019, pp. 2203–2214.

[65] G. Heyman et al. *Neural Code Search Revisited: Enhancing Code Snippet Retrieval through Natural Language Intent*. 2020. arXiv: 2008.12193.

[66] E. Shi et al. *CAST: Enhancing Code Summarization with Hierarchical Splitting and Reconstruction of Abstract Syntax Trees*. 2021. arXiv: 2108.12987.

[67] A. LeClair et al. *Improved Code Summarization via a Graph Neural Network*. 2020. arXiv: 2004.02843.

[68] S. Haque et al. "Improved Automatic Summarization of Subroutines via Attention to File Context." In: *Int'l Conf. Mining Softw. Repositories*. ACM, 2020, pp. 300–310.

[69] Z. Yang et al. *A Multi-Modal Transformer-based Code Summarization Approach for Smart Contracts*. 2021. arXiv: 2103.07164.

[70] M.-A. Lachaux et al. *Unsupervised Translation of Programming Languages*. 2020. arXiv: 2006.03511.

[71] C. Liu et al. *CodeQA: A Question Answering Dataset for Source Code Comprehension*. 2021. arXiv: 2109.08365.

[72] Z. Yao et al. "StaQC: A Systematically Mined Question-Code Dataset from Stack Overflow." In: *Int'l World Wide Web Conf.* ACM, 2018, pp. 1693–1703.

[73] Y. Li et al. "Vulnerability Detection with Fine-Grained Interpretations." In: *ACM J. Meeting Eur. Softw. Eng. Conf. and Symp. Found. Softw. Eng.* ACM, 2021, pp. 292–303.

[74] H. Hanif et al. *VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection*. 2022. arXiv: 2205.12424.

[75] X. Cheng et al. "DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network." In: *ACM Trans. Softw. Eng. and Methodology* 30.3 (2021), pp. 1–33.

[76] Y. Li et al. "Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks." In: *Proceedings of the ACM on Progr. Languages* 3.OOPSLA (2019), pp. 1–30.

[77] G. Lin et al. "Cross-Project Transfer Representation Learning for Vulnerable Function Discovery." In: *IEEE Trans. Industrial Informatics* 14.7 (2018), pp. 3289–3297.

[78] H. Wang et al. "Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection." In: *IEEE Trans. Information Forensics and Security* 16 (2021), pp. 1943–1958.

[79] M. Pradel et al. "DeepBugs: A Learning Approach to Name-Based Bug Detection." In: *Proceedings of the ACM on Progr. Languages* 2.OOPSLA (2018), pp. 1–25.

[80] S. Chakraborty et al. *Deep Learning Based Vulnerability Detection: Are We There Yet?* 2020. arXiv: 2009.07235.

[81] Z. Liu et al. "Automating Just-in-Time Comment Updating." In: *Int'l Conf. Autom. Softw. Eng.* ACM, 2020, pp. 585–597.

[82] S. Panthaplackel et al. *Deep Just-In-Time Inconsistency Detection Between Comments and Source Code*. 2020. arXiv: 2010.01625.

[83] B. Li et al. "DeepCommenter: A Deep Code Comment Generation Tool with Hybrid Lexical and Syntactical Information." In: *ACM J. Meeting Eur. Softw. Eng. Conf. and Symp. Found. Softw. Eng.* ACM, 2020, pp. 1571–1575.

[84] U. Alon et al. *A General Path-Based Representation for Predicting Program Properties*. 2018. arXiv: 1803.09544.

[85] A. M. Mir et al. "Type4Py: Practical Deep Similarity Learning-Based Type Inference for Python." In: *Int'l Conf. Softw. Eng.* ACM, 2022, pp. 2241–2252.

[86] V. Raychev et al. "Predicting Program Properties from "Big Code"." In: *Symp. Princ. Prog. Lang.* Vol. 50. ACM, 2015, pp. 111–124.

[87] R. S. Malik et al. "NL2Type: Inferring JavaScript Function Types from Natural Language Information." In: *Int'l Conf. Softw. Eng.* IEEE, 2019, pp. 304–315.

[88] V. J. Hellendoorn et al. "Deep Learning Type Inference." In: *Joint Eur. Softw. Eng. Conf. and Symp. Found. Softw. Eng.* ACM, 2018, pp. 152–162.

[89] J. Wei et al. *LambdaNet: Probabilistic Type Inference Using Graph Neural Networks*. 2020. arXiv: 2005.02161.

[90] M. Allamanis et al. "Suggesting Accurate Method and Class Names." In: *Joint Eur. Softw. Eng. Conf. and Symp. Found. Softw. Eng.* ACM, 2015, pp. 38–49.

[91] W. Qi et al. *ProphetNet-X: Large-Scale Pre-training Models for English, Chinese, Multi-lingual, Dialog, and Code Generation*. 2021. arXiv: 2104.08006.

[92] Z. Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. arXiv: 2002.08155.

[93] B. Roziere et al. *DOBF: A Deobfuscation Pre-Training Objective for Programming Languages*. 2021. arXiv: 2102.07492.

[94] Y. Wang et al. *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation*. 2021. arXiv: 2109.00859.

[95] W. U. Ahmad et al. *Unified Pre-training for Program Understanding and Generation*. 2021. arXiv: 2103.06333.

[96] A. Mastropaolo et al. *Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks*. 2021. arXiv: 2102.02017.

[97] D. Guo et al. *GraphCodeBERT: Pre-training Code Representations with Data Flow*. 2021. arXiv: 2009.08366.

[98] A. Elnaggar et al. *CodeTrans: Towards Cracking the Language of Silicon's Code Through Self-Supervised Deep Learning and High Performance Computing*. 2021. arXiv: 2104.02443.

[99] V. J. Hellendoorn et al. "Global Relational Models of Source Code." In: *Int'l Conf. Learning Representations*. 2022.

[100] N. T. de Sousa et al. *JavaBERT: Training a Transformer-Based Model for the Java Programming Language*. 2021. arXiv: 2110.10404.

[101] U. Alon et al. "Code2vec: Learning Distributed Representations of Code." In: *Princ. Prog. Lang.* ACM, 2019, pp. 1–29.

[102] R.-M. Karampatsis et al. "Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code." In: *ACM/IEEE 42nd Int'l Conf. Softw. Eng.* ACM, 2020, pp. 1073–1085.

[103] W. Ma et al. "GraphCode2Vec: Generic Code Embedding via Lexical and Program Dependence Analyses." In: *Int'l Conf. Mining Softw. Repositories*. ACM, 2022, pp. 524–536.

[104] C. Niu et al. "SPT-code: Sequence-to-Sequence Pre-Training for Learning Source Code Representations." In: *Int'l Conf. Softw. Eng.* ACM, 2022, pp. 2006–2018.

[105] L. Phan et al. *CoTexT: Multi-task Learning with Code-Text Transformer*. 2021. arXiv: 2105.08645.

[106] A. Kanade et al. "Learning and Evaluating Contextual Embedding of Source Code." In: *Int'l Conf. Machine Learning*. PMLR, 2020, pp. 5110–5121.

[107] D. Shrivastava et al. "On-the-Fly Adaptation of Source Code Models." In: *NeurIPS 2020 Ws. Comp.-Assisted Prog.* 2020.

[108] S. Lu et al. *CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation*. 2021. arXiv: 2102.04664.

[109] P. Jain et al. "Contrastive Code Representation Learning." In: *Conf. Empirical Methods in Natural Lang. Processing*. ACL, 2021, pp. 5954–5971.

[110] D. Zügner et al. *Language-Agnostic Representation Learning of Source Code from Structure and Context*. 2021. arXiv: 2103.11318.

[111] N. Yefet et al. "Adversarial Examples for Models of Code." In: *Proceedings of the ACM on Progr. Languages* (2020), pp. 1–30.

[112] R. Compton et al. "Embedding Java Classes with Code2vec: Improvements from Variable Obfuscation." In: *Int'l Conf. Mining Softw. Repositories*. ACM, 2020, pp. 243–253.

[113] U. Alon et al. *Code2seq: Generating Sequences from Structured Representations of Code*. 2019. arXiv: 1808.01400.

[114] V. Efstathiou et al. *Semantic Source Code Models Using Identifier Embeddings*. 2019. arXiv: 1904.06929.

[115] L. Lannelongue et al. "Green Algorithms: Quantifying the Carbon Footprint of Computation." In: *Adv. Science* 8.12 (2021), p. 2100707.

[116] A. Lacoste et al. *Quantifying the Carbon Emissions of Machine Learning*. 2019. arXiv: 1910.09700.

[117] A. S. Luccioni et al. *Estimating the Carbon Footprint of BLOOM, a 176B Parameter Language Model*. 2022. arXiv: 2211.02001.

[118] I. Lakim et al. "A Holistic Assessment of the Carbon Footprint of Noor, a Very Large Arabic Language Model." In: *BigScience Episode #5 – Ws. Challenges & Perspectives in Creating Large Lang. Models*. ACL, 2022, pp. 84–94.

[119] L. Posani et al. *The Carbon Footprint of Distributed Cloud Storage*. 2019. arXiv: 1803.06973.

[120] J. Baliga et al. "Green Cloud Computing: Balancing Energy in Processing, Storage, and Transport." In: *Proceedings of the IEEE* 99.1 (2011), pp. 149–167.

[121] L. M. Hilty et al. "The Five Most Neglected Issues in "Green IT"." In: *CEPIS Upgrade* 12.4 (2011), p. 5.

[122] Z. Feng et al. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages." In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. 2020, pp. 1536–1547.

[123] X. Zhou et al. "A Map of Threats to Validity of Systematic Literature Reviews in Software Engineering." In: *Asia-Pacific Softw. Eng. Conf.* 2016, pp. 153–160.

[124] W. Martin et al. "A Survey of App Store Analysis for Software Engineering." In: *IEEE Trans. Softw. Eng.* 43.9 (2017), pp. 817–847.

[125] G. Kalaitzoglou et al. "A Practical Model for Evaluating the Energy Efficiency of Software Applications." In: *Int'l Conf. ICT for Sustainability*. 2014.

[126] J. Pineau et al. "Improving Reproducibility in Machine Learning Research (a Report from the NeurIPS 2019 Reproducibility Program)." In: *J. Machine Learning Research* 22.1 (2022), 164:7459–164:7478.

[127] E. M. Bender et al. "On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?" In: *Conf. Fairness, Accountability, and Transparency*. ACM, 2021, pp. 610–623.