Addressing Leakage in Self-Supervised Contextualized Code Retrieval

Johannes Villmow and **Viola Campos** and **Adrian Ulges** and **Ulrich Schwanecke**
RheinMain University of Applied Sciences
Wiesbaden, Germany
firstname.lastname@hs-rm.de

Abstract

We address contextualized code retrieval, the search for code snippets, helpful to fill gaps in a partial input program. Our approach facilitates a large-scale self-supervised contrastive training by splitting source code randomly into contexts and targets. To combat leakage between the two, we suggest a novel approach based on mutual identifier masking, dedentation, and the selection of syntax-aligned targets. Our second contribution is a new dataset for direct evaluation of contextualized code retrieval, based on a dataset of manually aligned subpassages of code clones. Our experiments demonstrate that the proposed approach improves retrieval substantially, and yields new state-of-the-art results for code clone and defect detection.

1 Introduction

AI-supported software development has recently experienced growing interest (Lu et al., 2021), addressing various code understanding tasks such as code auto-completion (Svyatkovskiy et al., 2020), natural language code search (Husain et al., 2019), and code clone detection (Svajlenko and Roy, 2015). Our focus is on a related task called *contextualized code search* (Mukherjee et al., 2020; Dahal et al., 2022): Given an incomplete piece of code and a certain position of interest (e.g., the current cursor position), a retriever searches for code fragments that are relevant for filling in the missing piece. This setting aligns well with programmers' workflow, and differs substantially from the three tasks mentioned above as follows: (1) In contrast to natural language code search, contextualized code search can exploit local code context. (2) While code generated by autocompletion systems such as GitHub's CodEx (Chen et al., 2021) is prone to subtle programming errors, contextualized search leaves the developer in charge, and the origin of a solution remains transparent. (3) In contrast to clone detection, contextualized code search is not targeted at semantically similar code but code that complements the query.

A key challenge with contextualized code search is that supervised labels for relevant code passages are missing. Therefore, we bootstrap a *self-supervised learning process* by drawing inspiration from Cloze Tasks in natural language processing (Lee et al., 2019): Given a large-scale dataset containing pieces of code in 16 programming languages, we erase random blocks. We refer to these blocks as *targets*, and to their surrounding as *contexts*. Together, these pairs form samples for contrastive learning.

Unfortunately, as Figure 1 shows, this approach suffers from leakage between context and target, as the two share (1) common identifiers, (2) a matching indentation level, and (3) in some languages – if dividing a syntactic primitive such as for-loops – matching brackets. Retrievers might exploit these effects and bypass semantic similarity. To this end, our first contribution is a novel approach towards self-supervised code retrieval, which avoids the above bias through de-leaking steps such as mutual identifier masking and dedentation.

The second challenge we address is evaluation: So far, the focus of evaluating code retrieval systems has been on natural language queries (which can be bootstrapped from docstrings) (Husain et al., 2019). Contextualized code retrieval has been evaluated only indirectly via infilling quality (Lu et al., 2022; Parvez et al., 2021), which reflects the actual retrieval quality poorly. Therefore, our second contribution is a rigorous evaluation of contextualized code retrieval on a manually curated dataset based on aligned code clones. We call this dataset `COCOS` and make it available for future research. On `COCOS`, we demonstrate that retrieval quality benefits substantially from our de-leaking approach. Also, we achieve state-of-the-art results

Proceedings of the 29th International Conference on Computational Linguistics, pages 1006–1013
October 12–17, 2022.

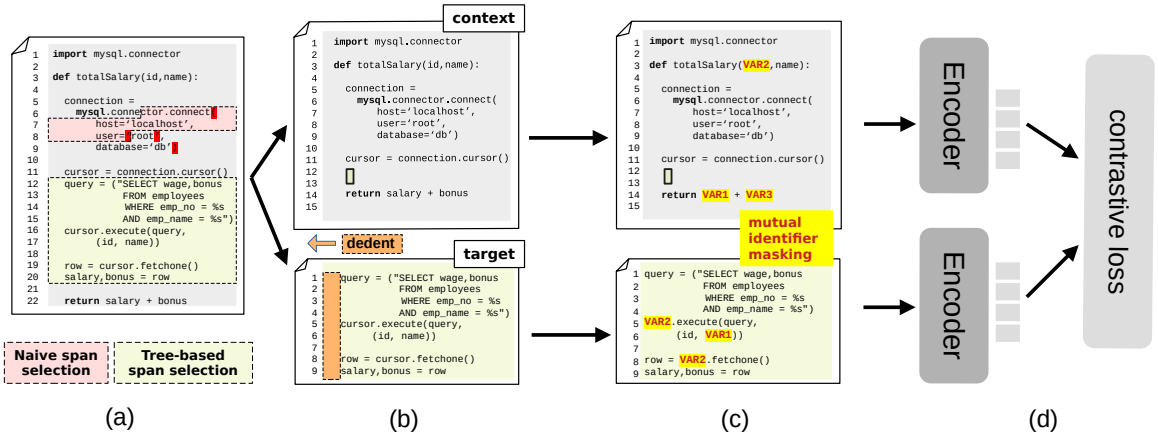

Figure 1: Our approach bootstraps code pairs for contrastive learning by removing target passages (green) from random code contexts (gray). To address leakage between the two – which can be due to matching identifiers, indentation, and brackets – we (a) select the target using the code's syntactic structure, (b) dedent the target (orange arrow), and (c) mutually mask identifiers. Finally, we apply contrastive learning on the resulting code pairs (d).

on the related tasks *code clone* and *defect detection* on CodeXGLUE (Lu et al., 2021).[1]

2 Approach

Given a piece of code as a token sequence $X{=}x_1, \ldots, x_m$, our goal is to boostrap a context-target pair for contrastive learning. The target is a subsequence $Y{=}x_i, \ldots, x_{i+L}$ with $1{\leq}i{\leq}i{+}L{\leq}m$. By replacing this subsequence with a special mask token, we obtain the context $X'{=}x_1, \ldots, x_{i-1}, x_{\text{MASK}}, x_{i+L+1}, \ldots, x_m$. To X' and Y we prepend a programming-language-specific CLS token.

To address the above leakages, we suggest three steps called tree-based span selection (TS), mutual identifier masking (IM) and dedenting (DE).

Tree-based span selection (TS) To select the target Y, we utilize X's concrete syntax tree[2], whose leaves consist of all code tokens. We define the target Y by masking a random *subtree*, which ensures Y to be a syntactically complete piece and avoids leakage due to brackets. Specifically, we first sample the target's length L from a normal distribution with $\mu{=}150$ and $\sigma{=}90$. We then select a node n covering at most L leaves/tokens and iteratively expand the selection, either to n's parent, or by adding n's direct siblings, until reaching the desired size L. Adding siblings allows for multiline targets spanning several statements.

Mutual Identifier Masking (IM) Next, we randomly replace identifiers[3] in X' and Y with special tokens (VAR1, VAR2, ...), to minimize leakage between identifiers. To preserve as much lexical information as possible, we mask only *mutual* identifiers present in both context and target. We hide 90% of those mutual identifiers randomly *either* in the context *or* in target code. For 5% of context-target pairs, we omit identifier masking altogether.

Dedenting (DE) Finally, in 90% of the training samples, we determine the indentation level of the target Y and dedent it, so that it has indentation level zero and the retriever cannot bypass semantic similarity by focusing on targets at the same indentation level.

2.1 Contrastive Training

We encode context code X' and target Y with the same transformer encoder and obtain sequence embeddings $\mathbf{q}, \mathbf{k} \in \mathbb{R}^d$, using the encoding of the CLS token. Following Wang et al. (2021b), we pretrain the transformer with alternating generation tasks identifier masking and span prediction[4] and use the pre-trained encoder.

The retriever is then trained by optimizing the following contrastive InfoNCE loss (van den Oord et al., 2018) with in-batch negative samples, where

[1] We release dataset, code and checkpoints to our experiments under github.com/villmow/coling-cocos

[2] We use the tree-sitter library for parsing.

[3] What is considered an identifier is defined in the grammar of a tree-sitter parser and varies between programming languages, i.e. we do not differentiate between variables, method names or method calls.

[4] Contrary to Wang et al. (2021b) we omit identifier detection and instead use our tree-based span selection to generate large and small spans.

f is the cosine similarity, K the amount of sequences in our batch, and τ=0.1 the temperature.

$$\mathcal{L}_\Theta = -log \frac{\exp(f(\mathbf{q}, \mathbf{k}^+)/\tau))}{\sum_{i=0}^{K-2} \exp(f(\mathbf{q}, \mathbf{k}_i^-)/\tau))} \quad (1)$$

To obtain harder negative samples – which have been found crucial for good retriever training (Ren et al., 2021) – we form batches only with samples from the same programming language.

3 Dataset

In this section, we first describe the large-scale data which our retriever is trained on. Second, we outline COCOS, a new bechmark we propose for contextualized code retrieval.

Pre-training Dataset Our self-supervised code retrieval model is pre-trained on 33M files in 16 programming languages (see Appendix A). As code files tend to be large, we truncate them using tree-based span selection (cmp. Section 2): Starting from a whole file, we randomly select sufficiently large spans of code (length between 150 and 800 tokens). We remove those segments from the original file and feed the shortened file as well as all individual segments as inputs X into the learning process described in Section 2. A special identifier (similar to code folding in an IDE) marks those positions in the original file where segments have been removed.

COCOS Evaluating contextualized code retrieval models is hard because little or no suitable evaluation data is available to indicate which sub-blocks in the code implement the same functionality. To address this gap, we have created a new dataset based on BigCloneBench (Svajlenko and Roy, 2015), a Java code clone dataset that provides pairs of semantically similar functions. Given a function in BigCloneBench, we manually select a sub-passage modeling a particular target functionality (e.g. extracting a zip file). We then label which lines in the function's clones match this functionality (see Listings 1 - 3 in the appendix). Based on these targets and their surrounding contexts, we evaluate how well a model retrieves targets implementing the same functionality in code clones. We manually gather 606 context-target pairs implementing 31 randomly selected functionalities. Finally, we add $10k$ non-relevant disctractor snippets by randomly sampling top-level statements from method bodies in CodeSearchNet (Husain et al., 2019). We call the dataset COCOS (**Co**ntextualized **Co**de **S**earch).

Model Features	MAP	NDCG	P@1	P@3	P@10
BM25 standard	12.36	43.8	27.89	24.92	17.13
BM25 camel	27.95	57.11	39.44	37.07	33.17
None	15.65	49.85	45.87	37.95	24.77
TS	26.47	59.64	58.09	50.77	36.96
TS, IM	33.78	66.03	69.80	60.95	45.33
TS, DE	36.32	65.94	59.41	54.57	44.39
TS, IM, DE	**50.87**	**76.28**	**73.60**	**70.30**	**59.70**

Table 1
Zeroshot code retrieval results for different de-leaking steps as described in Section 2: Tree-based span selection (TS); mutual identifier masking (IM); dedenting (DE). We report non-neural results for BM25 (Jones et al., 2000) using the Elasticsearch standard tokenizer (standard) and a tokenizer that splits on camel case (camel).

4 Evaluation

We report results for zero-shot code retrieval on COCOS and for two similar code understanding tasks from CodeXGlue (Lu et al., 2021), namely code clone detection and code defect detection. For all experiments, we report test results of the model with the highest mean reciprocal rank (MRR) on $30K$ held-out validation samples of the pre-training dataset.

4.1 Zero-shot Code Retrieval

We evaluate our models in a zero-shot setting, i.e. no fine-tuning on COCOS was applied. For each context all possible targets and the $10k$ distractor snippets are ranked, excluding the original target. To assess the proposed approaches, we compare variants of our model trained with different de-leaking steps and Okapi BM25 (Jones et al., 2000) as non-neural baseline. BM25 is evaluated using the standard Elasticsearch tokenization and a tokenizer splitting on camel case which is more suitable for source code. Table 1 reports our ablation studies showing mean average precision (MAP), normalized discounted cumulative gain (NDCG) and precision at k. We found the baseline trained without de-leaking to retrieve only samples with similar identifiers but to fail to consistently retrieve all relevant targets. Using all de-leaking steps significantly outperforms all ablations. Figure 2 also illustrates for a random selection of samples that our approach forms better clusters for both contexts and targets in embedding space.

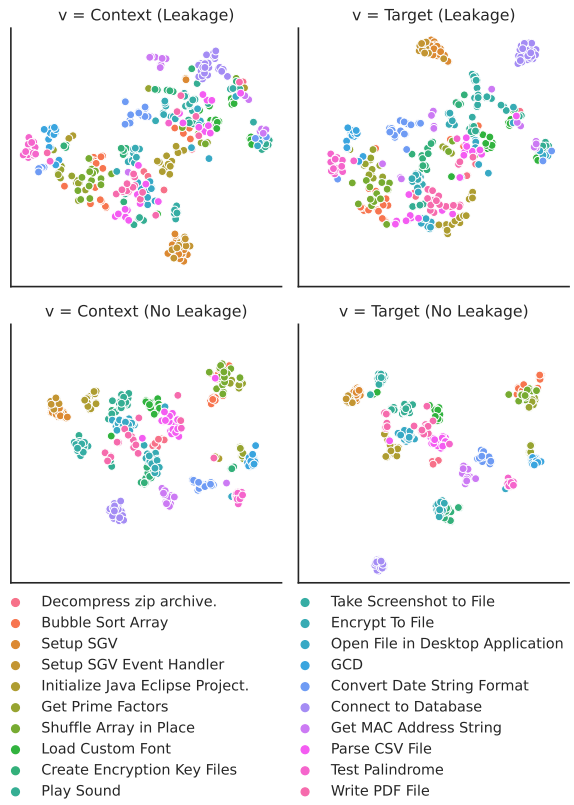

Figure 2: t-SNE comparison between the embeddings of the baseline model with leakage (top) and our model with leakage reduction steps applied (bottom). It can be seen that our approach forms better clusters.

4.2 Clone Detection and Defect Detection

We evaluate our model on clone detection on the `POJ-104` dataset (Mou et al., 2016), which consists of C and C++ programs for 104 problems from an open programming platform (OJ). We follow the evaluation procedure of CodeXGlue and report mean average precision (MAP@R) with R=499.

Finally for defect detection we evaluate on the `Devign` dataset (Zhou et al., 2019), which consists of vulnerable C functions manually collected from open source projects. The task is to predict whether the function is vulnerable. Following CodeXGlue we report accuracy. Baseline results for RoBERTa (Liu et al., 2019), CodeBERT (Feng et al., 2020), code2vec (Alon et al., 2019) and CoTexT (Phan et al., 2021) are reported in Lu et al. (2021), results for PLBART (Ahmad et al., 2021), GraphCodeBERT (Guo et al., 2021), SynCoBERT (Wang et al., 2021a) and CodeT5 (Wang et al., 2021b) are reproduced from Wang et al. (2021a) and Wang et al. (2021b). We find that our model outperforms state-of-the-art on both tasks by a large margin.

Model	Clone	Defect
	MAP@R	Accuracy
RoBERTa (code)	76.67	61.05
CodeBERT	82.67	62.08
code2vec	1.98	62.48
PLBART	-	63.18
GraphCodeBERT	85.16	63.21
SynCoBERT	88.24	64.50
CodeT5	-	65.78
CoTexT	-	66.62
Ours	**91.34**	**69.33**

Table 2
Results on code clone and defect detection (`POJ-104` and `Devign` dataset). We report results from Wang et al. (2021a) and Wang et al. (2021b).

5 Related Work

Code Representation Learning Given the success of pre-trained language models in NLP, recent work has extended pre-training to program syntax. Kanade et al. (2020) and Feng et al. (2020) train a BERT encoder on source code using masked language modeling. Guo et al. (2021) propose GraphCodeBERT to incorporate structural information such as data flow. Besides these encoder models, other work has pre-trained decoders (CodeGPT (Svyatkovskiy et al., 2020), CugLM (Liu et al., 2020)) or encoder-decoders (PLBART (Ahmad et al., 2021), CodeT5 (Wang et al., 2021b)) on pairs of natural language and program code. SynCoBERT (Wang et al., 2021a) is trained on various pre-training tasks on multimodal data, including code, comment and Abstract Syntax Tree (AST) representations. Guo et al. (2022) propose UniXcoder, which takes a similar approach but employs an encoder-decoder architecture instead of a single encoder.

In most of the above work, multiple modalities have been applied, e.g. code and natural language comments. In contrast to contextual code search, this setup does not come with leakage, which is the main concern of our work.

Contextualized Code Search retrieves complementary code, given a code context and sometimes an additional natural language query. Non-neural approaches include FaCoY (Kim et al.,

2018), which extends the query with related code from StackOverFlow, and Siamese (Ragkhitwetsagul and Krinke, 2019), which combines multiple code representations for pure code-to-code search. Aroma (Luan et al., 2019) clusters candidate code and intersects the snippets in each cluster to recommend likely subsequent code for a given snippet. Mukherjee et al. (2020) address the task by decompiling code fragments into a simpler representation called SKETCH (Murali et al., 2018) to learn a statistical model. The neural approach SCOTCH (Dahal et al., 2022) finetunes a CodeBERT model to discover relevant methods for queries combined with surrounding source code. None of the above approaches address the issue of leakage, either because they are non-neural (FaCoY, Siamese, Aroma), or leakage is neglected because the respective approach operates on method level (SCOTCH).

The issue of leakage in code search has only been scarcely studied before: Jain et al. (2021) propose ContraCode, a contrastive neural model that allows to retrieve code clones. To generate samples for contrastive learning, they augment code snippets using compiler-based semantic-preserving code transformations. Lu et al. (2022) propose ReACC, which uses partial code as search query in the context of retrieval-augmented code completion. To combat leakage, they insert dead code and rename variables. Compared to these approaches, our steps towards leakage reduction are much simpler. UniXcoder (Guo et al., 2022) pre-trains code representations using a variety of tasks, including contrastive learning. A positive sample pair is generated by running the same code piece through a transformer under dropout, which is a known trick for natural language (Gao et al., 2021) and can be seen as a simple form of de-leaking. Note that – since all our transformer encoders apply dropout during training – this mechanism applies for all models in our study too.

6 Conclusion

We have proposed a new approach towards unsupervised code retrieval, which reduces leakage between randomly drawn targets and their contexts. We also contribute a dataset COCOS, on which we demonstrate via ablations that leakage reduction is crucial for an efficient training. Our approach also yields competitive representations for related tasks, as demonstrated by new state-of-the-art results on clone and defect detection. An interesting future direction will be to combine our retriever with generators for a combined, unsupervised training.

Acknowledgements

This work has been supported by the Federal State of Hesse / Germany, Program LOEWE 5 (Project "Code Buddy").

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL).

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Samip Dahal, Adyasha Maharana, and Mohit Bansal. 2022. Scotch: A semantic code search engine for IDEs. In *Deep Learning for Code Workshop*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple contrastive learning of sentence embeddings. In *Proceedings of the 2021 Conference*

on Empirical Methods in Natural Language Processing, pages 6894–6910, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. 2021. Contrastive code representation learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

K. Sparck Jones, S. Walker, and S. E. Robertson. 2000. A probabilistic model of information retrieval: Development and comparative experiments. *Inf. Process. Manage.*, 36(6):779–808.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5110–5121. PMLR.

Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. FaCoY: A code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 946–957, New York, NY, USA. Association for Computing Machinery.

Kenton Lee, Ming-Wei Chang, and Kristina Toutanova. 2019. Latent retrieval for weakly supervised open domain question answering. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 6086–6096. Association for Computational Linguistics.

Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task learning based pre-trained language model for code completion. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 473–485, New York, NY, USA. Association for Computing Machinery.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 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, Dublin, Ireland. Association for Computational Linguistics.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. *ArXiv*, abs/2102.04664.

Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code recommendation via structural code search. *Proc. ACM Program. Lang.*, 3(OOPSLA).

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, page 1287–1293. AAAI Press.

Rohan Mukherjee, Chris Jermaine, and Swarat Chaudhuri. 2020. Searching a database of source codes using contextualized code search. *Proc. VLDB Endow.*, 13(10):1765–1778.

Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. 2018. Neural sketch learning for conditional program generation. In *International Conference on Learning Representations*.

Md Rizwan Parvez, Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2719–2734, Punta Cana, Dominican Republic. Association for Computational Linguistics.

Long N. Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James T. Anibal, Alec Peltekian, and Yanfang Ye. 2021. Cotext: Multi-task learning with code-text transformer. *CoRR*, abs/2105.08645.

Chaiyong Ragkhitwetsagul and Jens Krinke. 2019. Siamese: Scalable and incremental code clone search via multiple code representations. *Empirical Softw. Engg.*, 24(4):2236–2284.

Ruiyang Ren, Yingqi Qu, Jing Liu, Wayne Xin Zhao, QiaoQiao She, Hua Wu, Haifeng Wang, and Ji-Rong Wen. 2021. RocketQAv2: A joint training method for dense passage retrieval and passage re-ranking. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 2825–2835, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Jeffrey Svajlenko and Chanchal K. Roy. 2015. Evaluating clone detection tools with bigclonebench. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 131–140.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. *IntelliCode Compose: Code Generation Using Transformer*, page 1433–1443. Association for Computing Machinery, New York, NY, USA.

Aäron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *CoRR*, abs/1807.03748.

Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021a. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021b. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, Red Hook, NY, USA. Curran Associates Inc.

A Pre-training Dataset Details

We crawl 237k active GitHub repositories with more than 10 stars[5] and perform per file deduplication. We keep files in programming languages for which a tree-sitter parser is available (16 languages). The resulting dataset is shown in Table 3 and consists of $\approx 33M$ code files in 16 programming languages. We select 570 repositories for validation.

Language	Training	Valid	Total
Java	7,345,753	8,434	7,354,187
JavaScript	4,471,689	14,134	4,485,823
C++	3,734,357	1,698	3,736,055
Python	3,016,545	4,718	3,021,263
C#	2,843,642	570	2,844,212
TypeScript	2,299,964	2,392	2,302,356
C	2,242,379	781	2,243,160
PHP	2,206,063	4,648	2,210,711
Go	1,759,600	129	1,759,729
Ruby	1,068,668	3,397	1,072,065
Rust	366,891	54	366,945
CSS	349,525	2,579	352,104
Scala	273,822	1,198	275,020
Haskell	114,311	177	114,488
OCaml	55,838	0	55,838
Julia	34,403	29	34,432

Table 3
Number of files in unsupervised pre-training dataset.

B Training Details

On all models and tasks we use the AdamW optimizer and linearly increase the learning rate for 10% of the training steps, along with a polynomial decay for the remaining steps. We train our unsupervised models for 500k steps on a single A6000 GPU, with a peak learning rate of 0.0001 and use a dynamic batch size so that batches contain around 7000 tokens.

For clone and defect detection we fine-tune our model on the respective training set. Following Wang et al. (2021b) we run a brief sweep over learning rate, batch size and number of epochs and report results of the model with highest validation score, using the published evaluation code. We release our code including precise hyperparameter configs under github.com/villmow/coling-cocos.

[5]We consider a repository as active if there has been a pull request between 04/21 and 09/21.

```
public boolean extract(File f, String folder) {
    Enumeration entries;
    ZipFile zipFile;
    try {
        zipFile = new ZipFile(f);
        entries = zipFile.getEntries();
        [MASK]
        zipFile.close();
    } catch (IOException ioe) {
        this.errMsg = ioe.getMessage();
        Malgn.errorLog(
            "{Zip.unzip} " + ioe.getMessage()
        );
        return false;
    }
    return true;
}
```

Listing 1: Incomplete and masked query X' from our COCOS dataset. The [MASK] token denotes the current position of interest (cursor). Code that extracts elements from a zip file needs to be found.

```
while (entries.hasMoreElements()) {
    ZipArchiveEntry entry =
        (ZipArchiveEntry) entries.nextElement();
    if (entry == null) continue;
    String path = folder + "/"
        + entry.getName().replace('\\', '/');
    if (!entry.isDirectory()) {
        File destFile = new File(path);
        String parent = destFile.getParent();
        if (parent != null) {
            File parentFile = new File(parent);
            if (!parentFile.exists()) {
                parentFile.mkdirs();
            }
        }
        copyInputStream(
            zipFile.getInputStream(entry),
            new BufferedOutputStream(
                new FileOutputStream(destFile)
            )
        );
    }
}
```

Listing 2: The masked section Y manually selected from X (Listing 1). It has been re-formatted for better readability. Note that we omit Y from the result list for query X during evaluation.

```
ArchiveEntry ae = zis.getNextEntry();
while(ae != null) {
    //Resolve new file
    File newFile = new File(
        outputdir + File.separator + ae.getName()
    );

    //Create parent directories if not exists
    if(!newFile.getParentFile().exists())
        newFile.getParentFile().mkdirs();

    if(ae.isDirectory()) { //create if not exists
        if(!newFile.exists())
            newFile.mkdir();
    } else { //If file, write file
        FileOutputStream fos = new FileOutputStream(
            newFile);
        int len;
        while((len = zis.read(buffer)) > 0) {
            fos.write(buffer, 0, len);
        }
        fos.close();
    }

    //Proceed to the next entry in the zip file
    ae = zis.getNextEntry();
}
```

Listing 3: Possible solution, that implements the same functionality as target in Listing 2.