Evaluating Small-Scale Code Models for Code Clone Detection

Jorge Martinez-Gil

Software Competence Center Hagenberg GmbH
Softwarepark 32a, 4232 Hagenberg, Austria
jorge.martinez-gil@scch.at

Abstract

Code clone detection remains a difficult problem in software engineering, especially when code fragments share functionality but differ substantially in syntax or structure. Transformer-based code models have achieved strong results on clone detection, yet most prior studies emphasize larger models and do not provide a controlled comparison of smaller alternatives across diverse benchmarks. This paper presents a benchmark of six small-scale code models, namely CodeBERT, GraphCodeBERT, CodeT5, UniXCoder, PLBART, and PolyCoder, evaluated on four widely used datasets: BigCloneBench, Karnalim, POJ104, and PoolC. All models are fine-tuned under the same experimental protocol and assessed using accuracy, precision, recall, and F1-score. The results show that models that encode structural information, especially GraphCodeBERT and UniXCoder, achieve the most consistent performance across datasets, whereas sequence-to-sequence and decoder-only models are more sensitive to dataset characteristics. The study also shows that benchmark choice strongly affects the conclusions that can be drawn, since some datasets provide limited discriminative power under the evaluated setting. Overall, the findings indicate that small code models can provide strong clone detection performance at moderate model sizes, making them attractive candidates for practical software engineering settings. To support reproducibility, the implementation and experimental setup are publicly available at: `https://github.com/jorge-martinez-gil/small-code-models`

Keywords: Code Clone Detection, Benchmarking, Transformers, Small Code Models

1. Introduction

Code duplication is a common phenomenon in software systems and has long been associated with increased maintenance effort, reduced readability, and higher risk of inconsistencies during evolution [19]. Detecting code clones enables developers to identify redundant or related fragments and supports tasks such as refactoring, debugging, and quality assurance. While early techniques achieved reliable performance for exact or near-exact matches, identifying deeper forms of similarity remains difficult.

Code clones are typically categorized into four types, ranging from identical copies (Type-1) to functionally equivalent implementations with different structures (Type-4). Traditional approaches, including token-based, tree-based, and metric-based methods, perform well on Types 1 to 3 but often fail to capture Type-4 clones due to their reliance on surface-level representations. This limitation has motivated the adoption of machine learning techniques, particularly deep learning models trained on large corpora of source code.

Transformer-based models have significantly advanced the state of the art in code understanding tasks, including clone detection [9]. Through learned representations that combine lexical, structural, and contextual information, these models can detect similarities that are not easily captured with traditional techniques. However, most existing studies focus on large-scale models with billions of parameters, raising concerns about computational cost, energy consumption, and deployment feasibility in resource-constrained environments.

Smaller code models, typically containing hundreds of millions of parameters, offer a more practical alternative for many real-world applications. They can be fine-tuned and deployed with limited hardware, making them suitable for integration into development tools, continuous integration pipelines, and educational platforms. Despite these advantages, there is limited systematic evidence on how such models perform in clone detection across diverse datasets and conditions.

This paper addresses this gap through a controlled evaluation of six widely used small-scale code models across four benchmark datasets. The study focuses on performance consistency, cross-dataset behavior, and the relationship between model design and clone detection effectiveness. The following research questions guide the investigation:

- **RQ1:** What is the performance of small code models in clone detection across widely used benchmark datasets?
- **RQ2:** Which small code models exhibit the most stable performance across datasets?
- **RQ3:** What model characteristics are associated with stronger clone detection performance in this setting?

This paper provides the first controlled, cross-dataset comparison of small code models under identical fine-tuning conditions. The main contributions of this paper are as follows:

- A controlled benchmark of six small-scale code models for clone detection under a unified fine-tuning and evaluation protocol.
- An empirical comparison across four datasets that differ in size, language, clone definition, and difficulty.
- A cross-dataset analysis of performance stability, showing which model families remain reliable under changing benchmark conditions.

- A reproducible experimental pipeline that can support follow-up studies on model selection, efficiency, and robustness in clone detection.

The remainder of this paper is organized as follows. Section 2 reviews related work on clone detection and code models. Section 3 describes the datasets, models, and experimental setup. Section 4 presents the empirical results. Section 5 discusses the findings. Section 6 outlines threats to validity. Section 7 concludes the paper and identifies directions for future work.

2. Related Work

Code clone detection has been extensively studied in software engineering due to its impact on maintainability, program comprehension, and defect propagation [18]. Existing approaches can be broadly divided into traditional techniques and learning-based methods, with recent work increasingly focusing on transformer-based models for code understanding.

2.1. Traditional Clone Detection Techniques

Early approaches to clone detection rely on lexical, syntactic, or metric-based representations of source code [16, 17, 13, 14]. Token-based methods compare sequences of lexical units to identify duplicated or near-duplicated fragments. Tree-based techniques use abstract syntax trees (ASTs) to capture structural similarity, while metric-based approaches represent code through numerical features derived from complexity or structure.

These techniques are effective for detecting Type-1, Type-2, and, to some extent, Type-3 clones. However, they often fail to capture Type-4 clones, where code fragments implement similar functionality with different structures or programming patterns. This limitation has motivated the development of approaches that incorporate semantic information [12, 17].

2.2. Learning-Based Approaches to Clone Detection

Machine learning methods, particularly deep learning models, have been introduced to address the limitations of traditional techniques. Early work such as code2vec [2] demonstrated that distributed representations can capture structural and behavioral properties of code. Subsequent models extended this idea using neural architectures capable of learning from large-scale code corpora.

Transformer-based models have become dominant in this area due to their ability to model long-range dependencies and contextual relationships. Models such as CodeBERT [4], GraphCodeBERT [7], UniXCoder [6], PLBART [1], and CodeT5 [22] have been successfully applied to clone detection and related tasks. These models differ in how they incorporate structural information, with some relying purely on token sequences and others integrating data flow or graph-based representations.

Despite their strong performance, learning-based approaches still face challenges in detecting semantically equivalent code when surface-level similarity is limited. In addition, interpretability and robustness remain open issues, particularly when models rely on implicit representations learned during pretraining [15].

2.3. Small Code Models and Practical Constraints

Recent advances in code modeling have led to the development of increasingly large models, often containing billions of parameters. While these models achieve strong results, their computational requirements limit their applicability in many practical scenarios, including real-time systems, continuous integration pipelines, and resource-constrained environments.

Smaller models, typically with hundreds of millions of parameters, offer a more feasible alternative for deployment. They can be fine-tuned with moderate hardware requirements and integrated into development workflows with lower latency. Prior work has explored their use in various code-related tasks, including completion, summarization, and generation [24]. However, systematic evaluations of their performance in clone detection remain limited.

2.4. Benchmarking Studies and Research Gap

Several studies have evaluated code models on clone detection benchmarks, often focusing on individual datasets or specific model architectures. These evaluations are typically conducted under different experimental setups, making direct comparison difficult. In addition, most benchmarking efforts emphasize performance metrics without examining cross-dataset behavior or consistency.

To date, there is limited work that provides a unified evaluation of multiple small code models across diverse clone detection datasets using a consistent methodology. This gap makes it difficult to assess how model design choices affect performance under varying conditions and whether smaller models can provide reliable results across different types of clone detection tasks.

This paper addresses this limitation through a controlled benchmark of six small-scale code models across four datasets, focusing on performance consistency, cross-dataset variability, and observable differences associated with model design.

2.5. Code Models

Transformer-based models such as BERT [3] introduced pretraining strategies adapted to source code and contributed to improvements in code-related tasks. Recent work [25] has analyzed the internal representations learned from code, showing that transformer architectures are particularly effective. Table 1 lists six models commonly used in this area.

These models process source code as token sequences and encode structural and semantic aspects with different design choices.

Model	Architecture	Parameters	Significance
CodeBERT [4]	BERT-based	125M	Transformer-based model pre-trained on source code and natural language, effective for code similarity and clone detection.
GraphCodeBERT [7]	BERT+graphs	125M	Extension of CodeBERT that incorporates data flow information and improves structural understanding in code-related tasks.
CodeT5 [22]	Seq2Seq	220M	T5-based model adapted for code-related tasks, including clone detection and code summarization.
UniXCoder [6]	Encoder-Decoder	125M	Multi-modal model that uses both token and structure embeddings, improving performance on several code intelligence tasks.
PLBART [1]	Seq2Seq	140M	Pre-trained model suitable for generation and classification tasks, including clone detection.
PolyCoder (base) [23]	Decoder-only	160M	Model designed for code generation and understanding, trained on multiple programming languages, which makes it relevant for clone detection.

Table 1: Clone detection models, architectures, parameter counts, and significance

2.6. Why Small Code Models Matter in Practice

Small code models are attractive when low latency, moderate hardware requirements, and ease of integration are important. Representative scenarios include IDE assistance, educational tools, lightweight continuous integration pipelines, and software analysis services with limited compute budgets. In these settings, a model with hundreds of millions of parameters may offer a more practical deployment option than a multi-billion-parameter alternative.

Small models can also serve as the first stage in cascaded systems. In such workflows, the smaller model handles routine cases and forwards only uncertain inputs to a larger model. This design can reduce computational cost while preserving strong predictive performance on difficult cases. These practical considerations motivate a closer examination of how much clone detection performance can be obtained from models in the 100M to 220M parameter range.

2.7. Positioning of This Study

This study does not propose a new clone detection model. Its contribution is empirical. The goal is to compare several widely used small code models under the same conditions so that

differences in observed performance can be attributed more directly to model family and dataset characteristics rather than to inconsistent training setups.

The selected models cover encoder-only, encoder with structural signals, sequence-to-sequence, encoder-decoder, and decoder-only designs. This architectural diversity makes it possible to examine whether certain representation choices are associated with stronger and more stable clone detection performance across datasets.

The study intentionally excludes models whose scale would distort the comparison. CodeParrot [21] and CuBERT [8] were excluded because they focus on Python, which would reduce language diversity. Larger models such as StarCoder [10], StarCoder2 [11], InCoder [5], and CodeLlama [20] were excluded because their parameter counts are substantially higher than those of the models considered here.

3. Methodology

This section describes the datasets, model selection, preprocessing steps, training configuration, and evaluation protocol used in this study. The goal is to ensure a controlled and reproducible comparison across models while maintaining consistency in the experimental setup.

3.1. Datasets

Four publicly available datasets were used to evaluate clone detection performance: BigCloneBench (BCB), Karnalim, POJ104, and PoolC. These datasets differ in size, programming language, labeling strategy, and clone characteristics, allowing evaluation under diverse conditions.

Table 2 summarizes the main properties of each dataset, including the number of training and test samples and the programming languages covered.

Dataset	**Training Samples**	**Test Samples**	**Languages**
BigCloneBench	900k	415k	Java
Karnalim	327	140	Java
POJ104	32k	12k	C++
PoolC	480k	120k	Python

Table 2: Clone detection datasets used in this study

For BigCloneBench, a curated subset from CodeXGLUE was used. Karnalim focuses on plagiarism detection scenarios with disguised code reuse. POJ104 groups programs according to problem, allowing indirect construction of clone pairs based on shared functionality. PoolC provides labeled clone and non-clone pairs from open-source projects. The datasets vary not only in size but also in difficulty.

3.2. Clone Pair Construction and Preprocessing

All experiments are conducted as binary classification tasks, where each input consists of a pair of code snippets labeled as clone or non-clone. For datasets that do not provide explicit clone labels, such as POJ104, pairs are constructed according to shared problem identifiers, following common practice in the literature.

Code snippets are tokenized using the tokenizer associated with each model. Input pairs are concatenated with a separator token and truncated to a maximum sequence length of 512 tokens. No additional normalization, including identifier renaming or formatting changes, is applied, so the models operate on raw source code.

3.3. Model Selection

Six pre-trained code models were selected according to parameter size, architectural diversity, and availability:

- CodeBERT (encoder-only)
- GraphCodeBERT (encoder with data flow information)
- CodeT5 (sequence-to-sequence)
- UniXCoder (multi-modal encoder-decoder)
- PLBART (sequence-to-sequence)
- PolyCoder (decoder-only)

All selected models have parameter counts in the range of approximately 100M to 220M, making them suitable for fine-tuning on standard hardware. Larger models such as StarCoder and CodeLlama are excluded due to their substantially higher computational requirements.

3.4. Training Configuration

All models are fine-tuned for binary classification using a consistent training protocol. The following configuration is applied across all experiments:

- Batch size: 8
- Number of epochs: 3
- Weight decay: 0.01
- Random seed: 42

Each model uses its default learning rate as provided in the corresponding implementation. This choice reflects a practical setting in which models are fine-tuned with recommended configurations rather than extensive hyperparameter search.

The optimization objective is binary cross-entropy, and model selection is based on validation F1-score. No early stopping is applied, and the model from the final epoch is used for evaluation.

3.5. Evaluation Metrics

Performance is evaluated using four standard metrics:

- Accuracy: proportion of correctly classified pairs
- Precision: proportion of predicted clones that are true clones
- Recall: proportion of actual clones correctly identified
- F1-score: harmonic mean of precision and recall

F1-score is used as the primary metric due to its balance between precision and recall, which is particularly relevant in clone detection scenarios where both false positives and false negatives are undesirable.

3.6. Experimental Protocol and Reproducibility

All models are trained and evaluated under the same data splits and preprocessing conditions to ensure comparability. Experiments are conducted using a single random seed to maintain consistency across runs.

The full implementation, including data preprocessing, training scripts, and evaluation code, is publicly available to support reproducibility and further analysis: `https://github.com/jorge-martinez-gil/small-code-models`

4. Empirical Evaluation

This section reports the empirical results for all datasets and models. The emphasis is not only on absolute metric values, but also on cross-dataset stability, ranking consistency, and the extent to which benchmark choice affects the conclusions. Since the paper focuses on controlled comparison rather than task-specific optimization, all results should be interpreted as comparative evidence under a shared training protocol.

4.1. Overview of Results

Tables 3 to 6 report accuracy, precision, recall, and F1-score for all models across the four datasets. While most models achieve strong performance on several benchmarks, results vary according to dataset characteristics, including size, labeling strategy, and clone complexity.

Overall, models that incorporate structural or multi-modal representations tend to achieve more consistent performance across datasets. In contrast, sequence-to-sequence and decoder-only models exhibit greater variability, suggesting stronger sensitivity to dataset-specific properties.

4.2. BigCloneBench (BCB)

Results on BCB show that PLBART and UniXCoder achieve the highest overall performance, with F1-scores above 0.90. GraphCodeBERT also performs strongly, whereas CodeBERT and PolyCoder lag behind.

The strong performance of PLBART and UniXCoder suggests that models combining sequence modeling with richer representations can effectively capture patterns present in large-scale datasets. UniXCoder achieves the highest recall, indicating strong ability to identify true clone pairs, whereas PLBART maintains a more balanced precision-recall trade-off.

The relatively lower performance of CodeBERT and PolyCoder points to limitations in models that rely primarily on token-level representations. These models appear less effective when structural or contextual information is required.

Model	**Accuracy**	**Precision**	**Recall**	**F1-score**
CodeBERT	0.954	0.798	0.897	0.844
GraphCodeBERT	0.959	0.832	0.888	0.858
CodeT5	0.970	0.868	0.926	0.896
UniXCoder	0.971	0.864	0.941	0.901
PLBART	0.973	0.876	0.936	0.905
PolyCoder	0.951	0.798	0.874	0.834

Table 3: Performance on the BCB dataset

4.3. Karnalim Dataset

The Karnalim dataset reveals clearer differences between models despite its small size. PLBART achieves the highest performance, followed by UniXCoder and GraphCodeBERT, whereas CodeBERT and CodeT5 show lower precision.

The high recall across several models indicates that most clone pairs are correctly identified. However, lower precision suggests over-prediction in some cases, particularly in models without structural modeling.

This dataset includes intentionally modified code designed to disguise similarity, making it challenging despite its size. Models that incorporate structural or contextual information perform better under these conditions.

Model	Accuracy	Precision	Recall	F1-score
CodeBERT	0.667	0.667	1.000	0.800
GraphCodeBERT	0.899	0.882	0.978	0.923
CodeT5	0.725	0.708	1.000	0.829
UniXCoder	0.942	0.938	0.978	0.957
PLBART	0.971	0.978	0.978	0.978
PolyCoder	0.855	0.833	0.978	0.900

Table 4: Performance on the Karnalim dataset

4.4. POJ104 Dataset

Performance on POJ104 differs from other datasets, with PolyCoder achieving the highest F1-score, followed by UniXCoder. Encoder-based models show consistent but slightly lower performance, whereas sequence-to-sequence models perform less effectively.

Since POJ104 is based on problem-level grouping rather than explicit clone labels, models must identify functional similarity across independently written solutions. This increases variability in structure and implementation style.

These results show that model effectiveness depends on how clone relationships are defined, and performance may vary when moving from explicit clone datasets to problem-based similarity settings.

Model	Accuracy	Precision	Recall	F1-score
CodeBERT	0.863	0.856	0.867	0.861
GraphCodeBERT	0.864	0.857	0.867	0.862
CodeT5	0.600	0.567	0.794	0.662
UniXCoder	0.883	0.883	0.879	0.881
PLBART	0.733	0.676	0.875	0.763
PolyCoder	0.900	0.895	0.905	0.900

Table 5: Performance on the POJ104 dataset

4.5. PoolC Dataset

On the PoolC dataset, UniXCoder achieves the highest performance, followed by GraphCodeBERT and CodeT5. CodeBERT and PolyCoder show slightly lower results.

The dataset includes diverse clone types and coding styles. The balanced precision and recall achieved by UniXCoder indicate effective handling of this diversity, whereas CodeT5 shows higher recall but lower precision, suggesting more false positives.

Model	Accuracy	Precision	Recall	F1-score
CodeBERT	0.936	0.934	0.938	0.936
GraphCodeBERT	0.942	0.934	0.949	0.941
CodeT5	0.943	0.914	0.977	0.944
UniXCoder	0.949	0.960	0.936	0.948
PLBART	0.937	0.929	0.946	0.937
PolyCoder	0.924	0.912	0.936	0.924

Table 6: Performance on the PoolC dataset

4.6. Cross-Dataset Comparison

Table 7 presents model rankings according to F1-score across datasets.

Model	BCB	Karnalim	POJ104	PoolC
CodeBERT	5	6	4	4
GraphCodeBERT	4	3	3	3
CodeT5	3	5	6	2
UniXCoder	2	2	2	1
PLBART	1	1	5	4
PolyCoder	6	4	1	6

Table 7: F1-score ranking (1 = best)

Table 8 reports macro-averaged F1-scores and standard deviation.

Model	Mean F1-score	Std. Dev.
GraphCodeBERT	0.925	0.053
UniXCoder	0.918	0.041
PLBART	0.904	0.083
CodeBERT	0.895	0.072
PolyCoder	0.892	0.030
CodeT5	0.855	0.137

Table 8: Macro-average F1-score across datasets

Table 9 reports instance-weighted average F1-scores.

4.7. Answer to the Research Questions

RQ1: Small code models achieve strong clone detection performance on several benchmarks, with F1-scores above 0.90 on BCB, Karnalim, and PoolC for the strongest models. However, performance varies markedly across datasets, indicating that benchmark characteristics strongly influence the observed ranking.

RQ2: UniXCoder and GraphCodeBERT exhibit the strongest cross-dataset stability. UniXCoder attains the best instance-weighted average F1-score, whereas GraphCodeBERT achieves the strongest macro-level average.

Model	Weighted F1-score	Std. Dev.
UniXCoder	0.937	0.042
GraphCodeBERT	0.917	0.053
PLBART	0.917	0.084
PolyCoder	0.912	0.053
CodeBERT	0.888	0.071
CodeT5	0.866	0.126

Table 9: Instance-weighted average F1-score across datasets

RQ3: Models that incorporate structural or multi-modal signals tend to remain more reliable across datasets than models that rely primarily on token sequences. This pattern is consistent with the view that structural information is beneficial for clone detection, especially when surface similarity is insufficient.

5. Discussion

The results show that small code models can perform strongly on clone detection, but they also show that conclusions depend heavily on the benchmark used. This is the central message of the study. A model that appears dominant on one dataset may lose that advantage on another, especially when clone pairs are defined differently or when the dataset contains disguised or semantically similar implementations.

5.1. Performance Across Datasets

No model dominates across all datasets. UniXCoder and GraphCodeBERT are the most consistent overall, but each model family has strengths that depend on the benchmark. PLBART performs very well on BCB and Karnalim, whereas PolyCoder reaches its strongest result on POJ104. These shifts indicate that clone detection performance is shaped not only by architecture, but also by dataset construction, language, and the operational definition of similarity.

5.2. Implications of Structural Information

Models that encode structural or multi-modal signals, especially GraphCodeBERT and UniXCoder, tend to achieve more stable results across datasets. This suggests that information beyond token sequences is useful when clone detection requires sensitivity to program organization rather than shallow lexical overlap.

At the same time, the advantage is not universal. POJ104 favors PolyCoder, which suggests that decoder-style models may capture useful regularities when similarity is framed at the problem level rather than through explicit clone labels. The main lesson is therefore not that one architecture is always superior, but that architecture interacts with dataset design.

5.3. Implications for Practice

From a software engineering standpoint, the results support the use of small code models when practitioners need strong predictive performance without moving to very large models. UniXCoder and GraphCodeBERT appear to offer the best overall trade-off in the present benchmark, especially when consistency across tasks matters.

Model choice should still depend on application priorities. High-recall configurations are attractive when missing a clone is costly, whereas higher-precision behavior is preferable when false alarms would burden developers. The benchmark therefore supports model selection decisions, not only leaderboard comparison.

5.4. Error Analysis

A manual inspection of representative misclassifications suggests three recurring failure modes. First, several false positives occur when two code fragments share strong lexical or structural overlap but differ in computational intent. Second, false negatives arise when semantically related solutions use substantially different control-flow patterns, library calls, or decomposition strategies. Third, disguised clones with localized edits, renamed identifiers, or reordered statements remain difficult for models that rely more heavily on token-level evidence.

Table 10 summarizes representative examples of these failure modes. Although this analysis is qualitative, it aligns with the broader pattern observed in the benchmark: surface similarity remains easier to capture than deeper semantic equivalence.

Failure Type	**Typical Pattern**	**Most Affected Models**
Lexically similar but semantically different	Similar variable names, loop shapes, or method structure, but different computation goals	CodeBERT, PolyCoder
Semantically similar with low token overlap	Same task solved with different APIs, decomposition style, or control-flow organization	CodeBERT, CodeT5
Disguised clone with localized edits	Identifier renaming, statement reordering, and lightweight obfuscation	CodeT5, PolyCoder
Problem-level similarity with stylistic variation	Same programming task solved through different algorithmic idioms	PLBART, CodeT5

Table 10: Summary of recurring error patterns observed during manual inspection.

6. Threats to Validity

Several factors may affect the validity of the results presented in this study.

- All models were trained using a shared configuration, including batch size, number of epochs, and optimization settings. This improves comparability, but it may not reflect the best configuration for each architecture. Some models may benefit from different hyperparameter choices, which could alter their relative ranking.
- The evaluation relies on accuracy, precision, recall, and F1-score. These are standard metrics for binary classification, but they do not fully capture the difficulty of semantic clone detection, especially when functional equivalence is expressed through substantially different code structures.
- The datasets differ in language, size, labeling strategy, and clone definition. POJ104 requires pair construction from shared problem identifiers rather than explicit clone labels. These characteristics affect how strongly each dataset supports comparative evaluation.
- The study uses benchmark datasets rather than industrial codebases. Real-world systems may contain broader language variation, larger code contexts, noisier labels, and more heterogeneous development practices. The reported results should therefore be interpreted as benchmark evidence rather than direct deployment evidence.
- All experiments were conducted with a single random seed. This improves repeatability of the reported runs, but it does not quantify performance variance across independent fine-tuning runs. Multi-seed evaluation would provide a stronger basis for comparing closely ranked models.

Despite these limitations, the study provides a controlled comparison that helps clarify how small code models behave across clone detection datasets with different characteristics.

7. Conclusions

This paper presented a controlled benchmark of six small-scale code models for code clone detection across commonly used datasets. The results show that small models can achieve strong performance, with UniXCoder and GraphCodeBERT emerging as the most consistent choices across benchmarks.

The study also shows that benchmark design matters greatly. Some datasets offer limited discriminative value, whereas others reveal meaningful differences between architectures. This finding supports the use of multi-dataset evaluation when drawing conclusions about clone detection performance.

The results indicate that models in the 100M to 220M range can provide a strong balance between predictive performance and practical deployability. This makes them relevant candidates for software engineering tools that cannot rely on very large models.

Future work should extend the benchmark with multi-seed evaluation, statistical testing, efficiency measurements, and harder settings for semantic clone detection. These additions would strengthen the evidence base for model selection in practical clone detection workflows.

Acknowledgments

The research reported in this paper has been funded by the Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation, and Technology (BMK), the Federal Ministry for Digital and Economic Affairs (BMDW), and the State of Upper Austria in the frame of SCCH, a center in the COMET - Competence Centers for Excellent Technologies Programme.

References

[1] Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K. (2021). Unified pre-training for program understanding and generation, . (pp. 2655–2668). doi:10.18653/V1/2021.NAACL-MAIN.211.

[2] Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019). code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, *3*, 1–29. doi:10.1145/3290353.

[3] Devlin, J., Chang, M., Lee, K., & Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. In J. Burstein, C. Doran, & T. Solorio (Eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)* (pp. 4171–4186). Association for Computational Linguistics. doi:10.18653/V1/N19-1423.

[4] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages. In T. Cohn, Y. He, & Y. Liu (Eds.), *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020* (pp. 1536–1547). Association for Computational Linguistics volume EMNLP 2020 of *Findings of ACL*. doi:10.18653/V1/2020.FINDINGS-EMNLP.139.

[5] Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, S., Zettlemoyer, L., & Lewis, M. (2023). Incoder: A generative model for code infilling and synthesis, .

[6] Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., & Yin, J. (2022). Unixcoder: Unified cross-modal pre-training for code representation, . (pp. 7212–7225). doi:10.18653/V1/2022.ACL-LONG.499.

[7] Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C. B., Drain, D., Sundaresan, N., Yin, J., Jiang, D., & Zhou, M. (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.

[8] Kanade, A., Maniatis, P., Balakrishnan, G., & Shi, K. (2020). Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event* (pp. 5110–5121). PMLR volume 119 of *Proceedings of Machine Learning Research*. URL: http://proceedings.mlr.press/v119/kanade20a.html.

[9] Karmakar, A., & Robbes, R. (2021). What do pre-trained code models know about code? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 1332–1336). IEEE. doi:10.1109/ASE51524.2021.9678927.

[10] Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J. et al. (2023). Starcoder: may the source be with you! *Trans. Mach. Learn. Res., 2023*.

[11] Lozhkov, A., Li, R., Allal, L. B., Cassano, F., Lamy-Poirier, J., Tazi, N., Tang, A., Pykhtar, D., Liu, J., Wei, Y. et al. (2024). Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, .

[12] Martinez-Gil, J. (2014). An overview of textual semantic similarity measures based on web intelligence. *Artificial Intelligence Review*, *42*, 935–943.

[13] Martinez-Gil, J. (2022). A comprehensive review of stacking methods for semantic similarity measurement. *Machine Learning with Applications*, *10*, 100423.

[14] Martinez-Gil, J. (2024). Source code clone detection using unsupervised similarity measures. In *International Conference on Software Quality* (pp. 21–37). Springer.

[15] Martinez-Gil, J. (2025). Augmenting the interpretability of graphcodebert for code similarity tasks. *Int. J. Softw. Eng. Knowl. Eng.*, *35*, 657–678.

[16] Martinez-Gil, J., & Aldana-Montes, J. F. (2013). Semantic similarity measurement using historical google search patterns. *Information Systems Frontiers*, *15*, 399–410.

[17] Martinez-Gil, J., Paoletti, A. L., Rácz, G., Sali, A., & Schewe, K.-D. (2018). Accurate and efficient profile matching in knowledge bases. *Data & Knowledge Engineering*, *117*, 195–215.

[18] Nasrabadi, M. Z., Parsa, S., Ramezani, M., Roy, C., & Ekhtiarzadeh, M. (2023). A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *J. Syst. Softw.*, *204*, 111796. doi:10.1016/J.JSS.2023.111796.

[19] Rattan, D., Bhatia, R., & Singh, M. (2013). Software clone detection: A systematic review. *Information and Software Technology*, *55*, 1165–1199. doi:10.1016/J.INFSOF.2013.01.008.

[20] Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T. et al. (2023). Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, .

[21] Tunstall, L., Von Werra, L., & Wolf, T. (2022). *Natural language processing with transformers*. O'Reilly Media, Inc.

[22] Wang, Y., Wang, W., Joty, S. R., & Hoi, S. C. H. (2021). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, . (pp. 8696–8708). doi:10.18653/V1/2021.EMNLP-MAIN.685.

[23] Xu, F. F., Alon, U., Neubig, G., & Hellendoorn, V. J. (2022). A systematic evaluation of large language models of code, . (pp. 1–10). doi:10.1145/3520312.3534862.

[24] Zhang, X., Li, Z., Zhang, Y., Long, D., Xie, P., Zhang, M., & Zhang, M. (2023). Language models are universal embedders. *arXiv preprint arXiv:2310.08232*, .

[25] Zhang, Z., & Saber, T. (2025). Machine learning approaches to code similarity measurement: A systematic review. *IEEE Access*, . doi:10.1109/ACCESS.2025.3553392.