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

Detecting code clones is relevant to software maintenance and code refactoring. This challenge still presents unresolved cases, mainly when structural similarity does not reflect functional equivalence, though recent code models show promise. Therefore, this research aims to systematically measure the performance of several newly introduced small code models in classifying code pairs as clones or non-clones. The evaluation is based on five datasets: BigCloneBench, CodeJam, Karnalim, POJ104, and PoolC, as well as six code models: CodeBERT, GraphCodeBERT, Salesforce T5, UniXCoder, PLBART, and Polycoder. Most models performed well across standard metrics, including accuracy, precision, recall, and F1-score. However, a marginal fraction of clones remains challenging to detect, especially when the code looks similar but performs different operations. The source code that illustrates our approach is available at: `https://github.com/jorge-martinez-gil/small-code-models`

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

1. Introduction

The presence of code clones is a frequent issue in software development, affecting maintainability. Code duplication can raise complexity, increase maintenance costs, and introduce mistakes during updates. Therefore, clone detection helps developers find and restructure redundant code snippets, which usually supports better software organization [29].

Earlier approaches to clone detection are usually grouped into three types: token-based, tree-based, and metric-based. Token-based methods compare sequences of tokens to detect similar codes. Tree-based techniques rely on abstract syntax trees to assess structural likeness, and metric-based ones convert code into numerical features for similarity checks. These techniques were generally effective at finding exact copies or structurally related code but often miss clones that perform the same task with different syntax.

Over the last ten years, machine learning (ML) has brought new approaches to clone detection. For example, some deep learning (DL) models trained on extensive code collections have shown

the ability to recognize patterns that go beyond surface-level similarity [13]. These models learn behavioral aspects of code, allowing them to identify matches that traditional techniques often overlook. Their strong performance led them to dominate benchmark rankings soon after their introduction.

Recently, new code models have been introduced that demonstrate improved performance [2]. Given the increasing awareness of large models' environmental and computational costs, it is worth investigating whether smaller, more resource-efficient models can achieve results comparable to the larger ones. For this reason, this study focuses on models considered small, acknowledging that there is no universally accepted definition of a small code model. In line with current developments, we define small-scale models as those at least two orders of magnitude smaller than the largest available code models (medium models would be roughly one order smaller). The number of parameters is typically used as the basis for such comparisons. Therefore, we consider models with just a few hundred million parameters in practice. These smaller models present a trade-off between performance and efficiency, making them suitable for use in environments with limited computational resources.

So far, there has been no thorough comparative and exhaustive evaluation of how well small code models perform in clone detection. Therefore, this work is the first to compare multiple small-scale code models for clone detection across several benchmark datasets using a consistent evaluation setup. The goal is to answer the following research questions:

- **RQ1:** What is the performance of small code models in clone detection on the most adopted clone-related datasets?
- **RQ2:** What small code model demonstrates the highest reliability?
- **RQ3:** Which architectural features in small models contribute most to clone detection performance?

The rest of this paper is structured as follows: Section 2 reviews related work, covering traditional and modern approaches for clone detection. Section 3 presents the methodology, including dataset details, model architecture, and evaluation metrics that we have used. Section 4 shows our experimental results, analyzing the performance of several models over different datasets under equivalent conditions. Section 5 discusses the findings and potential limitations. Section 6 concludes with a summary of contributions and possible directions for future research.

2. Related Work

The problem of the code clones or equivalent code snippets is common in software development [25]. They can complicate maintenance, as changes or bug fixes in one instance often need to be

replicated in others. Clones may also make it difficult to understand the programs and contribute to inconsistencies or defect propagation across large codebases [23].

Code clones are usually grouped into four classes: Type-1 (identical code), Type-2 (minor edits like renaming), Type-3 (structural changes with retained logic), and Type-4 (different structure but same functionality). Traditional methods perform well on Type-1, Type-2, and Type-3, but Type-4 remains challenging due to limited surface similarity. This limitation has increased interest in code models capable of identifying deeper similarities [15]. Smaller models that encode structure and meaning have shown promising results in identifying such clones, mainly when computing resources are restricted. Their ability to work across different programming languages makes them practical for automated code maintenance [36].

2.1. Code-related tasks, clone detection, and their applications

Recent work has extended the use of language models to code-related tasks beyond generation. Pham et al.[26] introduced MAGECODE, a system for detecting machine-generated code using pre-trained models that can distinguish between human- and machine-written code. Hemberg et al.[10] applied language models to program refinement, showing their use in automated code modification. Husein et al.[11] reviewed the performance of language models in code completion, cataloging their current capabilities across languages and tasks. Ramler et al.[28] documented how AI-assisted coding tools are used in professional software development. Finally, Zhang et al.[35] explored multi-intent code comment generation using large models, aiming to support code comprehension and maintainability in day-to-day programming.

Clone detection has a long research tradition as a subfield of code-related tasks. Proposed methods are classified as lexical, syntactic, or semantic, with the semantic category being the most difficult to model. Prior work [20] has investigated unsupervised techniques based on multiple similarity metrics. Traditional methods are effective for exact or near-exact matches, whereas neural models offer better recall by identifying functionally similar code despite structural variation. Early neural approaches like code2vec [3] showed that embeddings can capture structure and behavior. More recent efforts explore hybrid models that use genetic programming [18], and ensemble-based unsupervised approaches [21, 19] continue to show potential for integration with learned models. Semantic similarity remains challenging to quantify objectively [9] and interpretability as well [22]. The reason is the wide variety of applications where it can have an impact.

2.2. Code Models

Transformer-based models such as BERT [4] introduced pretraining strategies adapted to source code, contributing to improvements in code-related tasks. Recent work [37] 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.

Model	Architecture	Parameters	Significance
CodeBERT [5]	BERT-based	125M	Transformer-based model pre-trained on source code and natural language, effective for code similarity and clone detection.
GraphCodeBERT [8]	BERT+graphs	125M	Extension of CodeBERT by incorporating data flow information, improving structural understanding in code-related tasks.
Salesforce T5 (base) [33]	Seq2Seq	220M	A T5-based model fine-tuned for code-related NLP tasks, including clone detection and code summarization.
UniXCoder [7]	Encoder-Decoder	undisclosed*	A multi-modal model that uses both token and structure embeddings, enhancing performance on various code intelligence tasks.
PLBART [1]	Seq2Seq	140M	Pre-trained model suitable for generation and classification tasks, including clone detection.
PolyCoder (base) [34]	Decoder-only	160M	Model designed for code generation and understanding, trained on multiple programming languages, making it versatile for clone detection.

* Although the authors have not disclosed the number of parameters, its physical size suggests that it is below 200M.

Table 1: Clone Detection Models, Architectures, Parameters, and Their Significance

These models process source code as token sequences and consider structural and semantic aspects. Current efforts aim to make these models more effective when dealing with complex code structures.

2.3. Powerful Large Code Models vs Efficient Small Code Models

Small models are often better when hardware limits are strict or low latency is essential. Typical cases include code editors, browser extensions, mobile development environments, or continuous integration pipelines. These systems require quick responses and minimal resource usage. A smaller model can typically run without a GPU and still return valuable results, making integrating it into everyday developer tools easier.

They can also work as filters in multi-stage setups. The smaller model handles the easy cases and forwards only uncertain or borderline inputs to a larger model. This setup can reduce total

computation time without sacrificing too much accuracy. It also avoids the energy demands of running large models constantly, which matters in shared environments.

2.4. Contribution Over the State-of-the-Art

This work compares several small code models used for clone detection. Many models have been released recently, but direct comparisons under the same setup are rare. This study addresses that by evaluating a diverse group of architectures under consistent conditions. CodeBERT and GraphCodeBERT are encoder-only and are mainly used for classification tasks. CodeT5 and PLBART use a sequence-to-sequence structure suitable for generation and classification. UniXCoder supports both task types without needing changes to its architecture. PolyCoder is decoder-only and belongs to the GPT family.

These models differ in how they represent source code. CodeBERT uses plain text tokens. GraphCodeBERT and UniXCoder include structural elements, which help when surface similarity is insufficient. PLBART and PolyCoder are geared toward code generation, with PolyCoder also trained in multiple programming languages. Among the models considered, Salesforce T5 has the highest parameter count.

Please note that in the context of this study, we have explicitly discarded CodeParrot [32] and CuBERT [12] due to their focus on Python-only code, which limits generalization across multiple programming languages. Additionally, we have excluded some very popular models such as StarCoder [16], Starcoder2 [17], Incoder [6], and CodeLlama [30] because their most basic versions range between 3B and 7B parameters, which cannot guarantee a balanced comparison with the models with significantly lower parameter counts we are considering here.

3. Methodology

There has been limited evaluation of small code models in clone detection so far [39]. To address this gap, we ran experiments using five well-known datasets: BigCloneBench [31], CodeJam [38], Karnalim [14], PoolC [24], and POJ104 [27]. Each dataset includes (implicitly or explicitly) labeled code pairs, making measuring model performance across different types of clones possible. The six models selected for this study were CodeBERT, GraphCodeBERT, Salesforce T5, UniXCoder, PLBART, and PolyCoder. They were chosen based on their parameter count and availability. All models were tested under the same setup using accuracy, precision, recall, and F1-score.

3.1. Technical Features

Table 2 lists the datasets used to measure clone detection performance, including sample counts and the programming languages they cover. The datasets differ in source, size, and purpose, allowing for evaluation across various conditions.

Dataset	Training Samples	Test Samples	Languages
BigCloneBench	900k	415k	Java
CodeJam	726k	14k	Java
Karnalim	327	140	Java
POJ104	32k	12k	C++
PoolC	480k	120k	Python

Table 2: Clone Detection Datasets: Samples and Programming Languages

3.2. The BigCloneBench Dataset

BigCloneBench (BCB) is a large dataset built from open-source projects and labeled with clone types ranging from exact copies to structurally different but functionally similar code. Labels are generated using a mix of automated tools and manual checks. It is widely used to test clone detection methods, including older techniques and ML models. Due to its size, it is also suitable for testing how well models scale.

3.3. The Google Code Jam Dataset

The Google Code Jam (GCJ) dataset includes code submissions from the Google Code Jam contest, in which many participants solved the same problems. This led to multiple implementations of similar algorithms, differing in structure, style, and efficiency. Such variation allows models to be evaluated on functional similarity rather than exact matches, making the task harder and more realistic.

3.4. The Karnalim Dataset

The Karnalim dataset is used for source code plagiarism detection in academic settings. It contains labeled pairs of student submissions, some modified copies of others. Many samples include changes such as renaming variables or altering code structure to avoid easy detection. This makes it suitable for testing models that need to go beyond simple pattern matching. While smaller than datasets like BigCloneBench, it is well-suited for evaluating techniques to detect disguised reuse in student work.

3.5. The Peking University Online Judge Dataset

The POJ104 dataset includes student code submissions from the Peking University Online Judge. Each sample is tied to a specific programming problem, which allows grouping based on intended functionality. The dataset covers various styles and coding strategies, making it useful for testing models that aim to detect functional similarity. While it does not contain direct clone labels, the structure supports indirectly evaluating clone detection methods.

3.6. Poolc Benchmark Dataset

The PoolC dataset contains labeled code pairs marked as clones or non-clones. It includes samples from open-source projects covering a range of programming styles. Clone types include near duplicates, minor edits, and structurally different but functionally similar code. PoolC is split into training, validation, and test sets, making it suitable for ML experiments. Its structure allows consistent comparisons across models, though varying styles and unclear boundaries between clone types add difficulty.

4. Empirical Evaluation

This section presents an empirical evaluation of the small code models, focusing on their training setup, evaluation metrics, and experimental results. The goal is to understand the model's strengths and limitations across different datasets and identify areas for improvement.

4.1. Model and Training Setup

All the models have been fine-tuned for binary classification (clones or non-clones). The idea is that the models process concatenated code snippets and use attention mechanisms to determine similarity. The training parameters that we have used in all cases are:

- Batch size: 8
- Epochs: 3
- Weight decay: 0.01
- Seed: 42
- Optimization for: F1-score

Please note that we allow each model's learning rate to be the one by default.

4.2. Evaluation Metrics

Performance is measured using four metrics: accuracy, which refers to the proportion of correctly classified code snippets; precision, which indicates the proportion of detected clones that are actual clones; recall, which captures the proportion of actual clones that are correctly detected; and F1-score, defined as the harmonic mean of precision and recall.

4.3. BCB Dataset

Table 3 presents the performance evaluation of our small-scale code models on the BCB benchmark dataset.

Benchmark	**Accuracy**	**Precision**	**Recall**	**F1-score**
CodeBERT	0.954	0.798	0.897	0.844
GraphCodeBERT	0.959	0.832	0.888	0.858
Salesforce T5	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 benchmark dataset

- PLBART achieves the best performance, with the highest accuracy (0.973), precision (0.876), and a strong F1-score (0.905). Therefore, PLBART is the most balanced model for detecting code clones in BCB.
- UniXCoder also performs exceptionally well, achieving an accuracy of 0.971 and the highest recall (0.941). Therefore, UniXCoder is highly effective at identifying true clones but has slightly lower precision (0.864) than PLBART.
- Salesforce T5 achieves a strong F1-score of 0.896, with good precision (0.868) and recall (0.926). This model performs well but does not surpass PLBART or UniXCoder in performance.
- GraphCodeBERT and CodeBERT perform moderately, with GraphCodeBERT showing better precision (0.832) than CodeBERT (0.798). However, their recall and F1 scores are significantly lower than the top-performing models.
- PolyCoder has the weakest performance, with the lowest accuracy (0.951) and F1-score (0.834).

4.4. GCJ Dataset

Table 4 shows the performance of the small-scale code models under study on the GCJ benchmark dataset.

- All models achieve perfect performance on the GCJ dataset, with accuracy, precision, recall, and F1-score equal to 1.000.

Benchmark	Accuracy	Precision	Recall	F1-score
CodeBERT	1.000	1.000	1.000	1.000
GraphCodeBERT	1.000	1.000	1.000	1.000
Salesforce T5	1.000	1.000	1.000	1.000
UniXCoder	1.000	1.000	1.000	1.000
PLBART	1.000	1.000	1.000	1.000
PolyCoder	1.000	1.000	1.000	1.000

Table 4: Performance on the GCJ benchmark dataset

- Such good results may indicate a problem (data bias or limited sample variation), but after further investigation, we have found that such results can be achieved with sufficient training. When training with less data, these results are not achieved.
- These results may reflect unknown dataset-specific characteristics, making it easier for these models to generalize.

4.5. Karnalim Dataset

Table 5 reports the performance of our small-scale code models on the Karnalim benchmark dataset.

Benchmark	Accuracy	Precision	Recall	F1-score
CodeBERT	0.667	0.667	1.000	0.800
GraphCodeBERT	0.899	0.882	0.978	0.923
Salesforce T5	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 5: Performance on the Karnalim benchmark dataset

- PLBART achieves the best performance, with the highest accuracy (0.971) and F1-score (0.978).
- UniXCoder also performs strongly, reaching 0.942 accuracy and 0.957 F1-score, with a slight drop in precision compared to PLBART.
- GraphCodeBERT follows, with solid scores across all metrics, particularly an F1-score of 0.923 and an accuracy of 0.899.
- PolyCoder performs reasonably well, with high recall (0.978) but a lower precision (0.833).
- Salesforce T5 and CodeBERT lag behind, achieving perfect recall (1.000) but low precision (0.708 and 0.667), leading to a noticeable drop in accuracy and F1-score.

4.6. POJ104 Dataset

Table 6 reports how our small-scale code models under study can perform on the POJ104 dataset.

Benchmark	**Accuracy**	**Precision**	**Recall**	**F1-score**
CodeBERT	0.863	0.856	0.867	0.861
GraphCodeBERT	0.864	0.857	0.867	0.862
Salesforce T5	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 6: Performance on the POJ104 benchmark dataset

- PolyCoder achieves the best performance, with accuracy and F1-score of 0.900, as well as balanced precision (0.895) and recall (0.905).
- UniXCoder also performs strongly, reaching 0.883 accuracy and 0.881 F1-score, maintaining a close balance between precision and recall.
- GraphCodeBERT and CodeBERT show solid and consistent results, with nearly identical metrics and F1-scores of 0.862 and 0.861, respectively.
- PLBART shows acceptable recall (0.875) but has noticeably lower precision (0.676), resulting in a moderate F1-score of 0.763.
- Salesforce T5 ranks lowest, with the weakest accuracy (0.600) and lowest precision (0.567), indicating a high rate of false positives despite a decent recall (0.794).

4.7. PoolC Dataset

Table 7 presents the performance evaluation of our small-scale code models on the PoolC benchmark dataset.

Benchmark	**Accuracy**	**Precision**	**Recall**	**F1-score**
CodeBERT	0.936	0.934	0.938	0.936
GraphCodeBERT	0.942	0.934	0.949	0.941
Salesforce T5	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 7: Performance on the PoolC benchmark dataset

- UniXCoder achieves the best performance, with the highest F1-score (0.948), precision (0.960), and strong accuracy (0.949).
- Salesforce T5 shows the highest recall (0.977) and strong F1-score (0.944), though its lower precision (0.914) indicates a higher rate of false positives.
- GraphCodeBERT also performs well, with a good balance of recall (0.949), precision (0.934), and F1-score (0.941).
- CodeBERT and PLBART are slightly behind, reaching F1-scores of 0.936–0.937, with minor differences in recall and precision.
- PolyCoder scores lowest, with accuracy and F1-score of 0.924 due to its lower precision (0.912).

4.8. Summary

Figure 1 shows the performance distribution of the six small-scale code models across the five benchmarks. The plot allows a quick comparison of their relative strengths. UniXCoder maintains a stable performance profile and reaches high scores across the board. PLBART performs well on most datasets, though it drops on POJ104. Salesforce T5 shows higher variance, with a lower score on POJ104 too. CodeBERT and PolyCoder achieve reasonable results but are slightly behind in consistency.

Table 8 shows the F1-score rankings of six code models across five benchmark datasets. A lower rank indicates better performance, with one being the best. Each entry reflects the position of a model on a specific dataset based on its F1 score. PLBART ranks in three datasets and stays within the top five. UniXCoder also performs well, never ranking lower than second place across all datasets, while other models show greater variation in their positions.

Model	**BCB**	**GCJ**	**Karnalim**	**POJ104**	**PoolC**
CodeBERT	5	**1**	6	4	4
GraphCodeBERT	4	**1**	3	3	3
Salesforce T5	3	**1**	5	6	2
UniXCoder	2	**1**	2	2	**1**
PLBART	**1**	**1**	**1**	5	4
PolyCoder	6	**1**	4	**1**	6

Table 8: F1-score ranking (1 = best) of each model across all benchmark datasets

Table 9 shows the mean and standard deviation of the F1-score across datasets. UniXCoder performs well and is the most stable. GraphCodeBERT has the highest mean but with slightly more fluctuation. PLBART and Salesforce T5 vary more across datasets, while PolyCoder is more uniform but has lower averages.

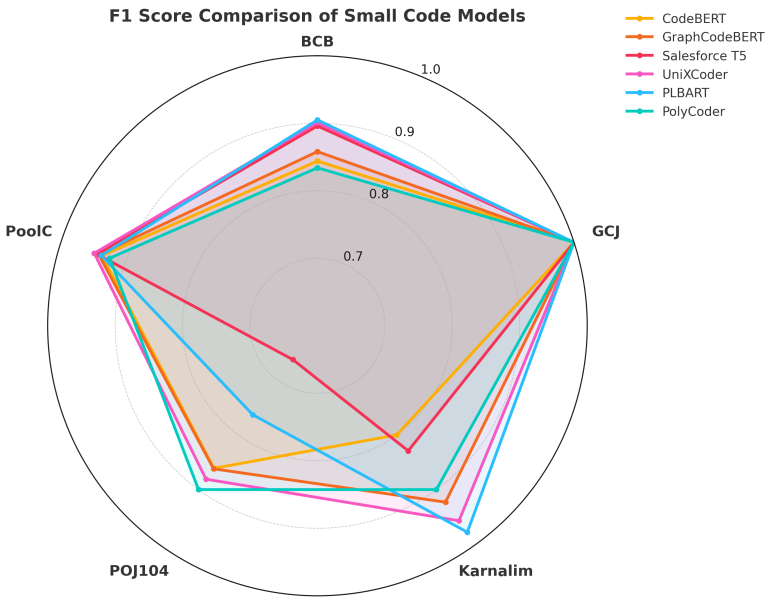

Figure 1: F1 scores of small code models measured on five standard clone detection datasets. Each axis represents one dataset, and each line tracks the performance of one model across all datasets.

Model	Mean F1-score	Standard Deviation
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
Salesforce T5	0.855	0.137

Table 9: Mean and standard deviation of F1-score across all benchmark datasets

Table 10 shows each model's F1-score mean and standard deviation, assuming the validation instances solved from each benchmark dataset. This is done to decrease the importance of small datasets such as Karnalim when establishing the ranking. UniXCoder leads with the highest average (0.937) and the lowest standard deviation (0.042). GraphCodeBERT and PLBART follow, averaging 0.917, though PLBART shows more variation. PolyCoder performs slightly below, while CodeBERT and Salesforce T5 rank lower.

4.9. Answer to the Research Questions

As a conclusion of our experiments, we can now answer the research questions previously formulated.

Model	E.-W. Mean F1-score	E.-W. 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
Salesforce T5	0.866	0.126

Table 10: Mean and standard deviation of F1-score across all benchmark datasets (equal-weighted, instance count)

RQ1: What is the performance of small code models in clone detection on the most adopted clone-related datasets?
AQ1: GraphCodeBERT reaches the highest average F1-score (0.925) and ranks consistently high in most datasets. PLBART and UniXCoder follow closely. However, if we consider the total number of instances validated, UniXCoder reaches the highest average F1-score (0.937). GraphCodeBERT and PLBART follow closely, both averaging 0.917.

RQ2: What small code model demonstrates the highest reliability?
AQ2: UniXCoder has a high mean F1-score (0.918) and the lowest standard deviation (0.041), which means stable performance across benchmarks. This is also valid even if all the instances validated are counted equally.

RQ3: Which architectural features in small models contribute most to clone detection performance?
AQ3: Graph-based representations (GraphCodeBERT) and multi-modal encoding (UniXCoder) perform better when compared to models that rely on general-purpose sequence-to-sequence designs. Models like Salesforce T5 and PLBART show variable results across datasets, and CodeBERT and PolyCoder generally score lower. This means specialized code-aware pretraining benefits clone detection more than general sequence modeling.

Furthermore, Table 11 summarizes the practical strengths of each small code model and suggests usage scenarios where their design or performance profile is particularly well-suited.

5. Discussion

Results allow us to compare model behavior across datasets and identify where models tend to perform well and where they do not. Each dataset tests different qualities, making it possible to observe differences in accuracy or consistency. Some models perform well across most cases, while others are more sensitive to specific dataset characteristics.

Use Case	Recommended Model(s)	Rationale
High recall (catch most clones)	PLBART, Salesforce T5	Near-perfect recall on multiple datasets; suitable when missing a clone is costly.
Low false positives (high precision)	UniXCoder, GraphCodeBERT	Strong precision and F1 scores; ideal when clone over-identification must be avoided.
Balanced performance (general use)	UniXCoder, PLBART	High F1 scores across datasets with good precision-recall trade-offs.
Resource-constrained environments (no GPU)	CodeBERT, PolyCoder	Fewer parameters; acceptable results without high compute requirements.
Cross-language or mixed-language projects	UniXCoder, PolyCoder	Multi-language support; performs well across language-diverse datasets.
Educational plagiarism detection	GraphCodeBERT, PLBART	Ideal when structural changes are present; high scores on the Karnalim dataset.
Real-time tools (low latency prediction)	CodeBERT	Encoder-only model; lightweight with fast inference time.
Multi-stage pipeline (filter + rerank)	CodeBERT (filter), PLBART (rerank)	Efficient two-stage setup: CodeBERT filters obvious cases, PLBART handles borderline examples.

Table 11: Recommended small-scale code models by use case

5.1. Observed Strengths

Our observations reveal that the models we have considered can maintain high accuracy across different datasets. Furthermore, the models generalize well to unseen examples. While most models perform well in most cases, there are still some problems with specific datasets that exhibit different types of clones or have less training data.

The evaluation across multiple datasets reveals that UniXCoder and GraphCodeBERT achieve strong performance, often ranking among the top models. UniXCoder leads on the PoolC and POJ104 datasets and remains highly competitive on BCB and Karnalim. It combines high recall with solid precision, effectively controlling false positives. GraphCodeBERT also shows strong generalization capabilities, particularly excelling in Karnalim.

In contrast, the performance of PLBART, Salesforce T5, and PolyCoder varies across datasets. While PLBART shines on BCB with top F1 scores, it underperforms in POJ104 and GCJ. Salesforce T5, although achieving top scores on GCJ, shows inconsistent behavior, especially in POJ104 and Karnalim, where its accuracy drops sharply. PolyCoder performs moderately, with lower precision and recall. These inconsistencies show the importance of evaluating models on diverse datasets, as performance can vary widely depending on data characteristics.

5.2. Observed Limitations

The models tested in this study show strong performance on several datasets, but there are still areas where they fall short. In particular, datasets with code snippets that appear similar in structure but differ in behavior often lead to wrong predictions. This means that the models favor surface patterns over deeper meaning.

Additionally, some models, including PLBART and Salesforce T5, show high recall but lower precision on specific datasets, indicating a tendency to classify too many pairs as clones. Small changes in the syntax, like alternative loop styles, can still mislead some models, particularly those that do not use structural information.

5.3. Threats to Validity

Our results must be interpreted carefully, as several factors may limit the validity of direct comparisons between models. While efforts were made to maintain consistency, some decisions, like using fixed hyperparameters or fixed random seeds, may favor some models over the rest. The following points outline potential limitations that could affect the fairness of the reported results.

- Several small-scale code models included in the benchmark were not originally trained for classification tasks. As a result, their internal representations and optimization objectives might not be optimal for clone detection, which may affect their performance.
- All models were trained using the same set of hyperparameters. While this favors consistency, it does not account for the possibility that different models may require different configurations to achieve optimal performance. This uniform setup could disadvantage some models and lead to unfair comparisons.
- We do not know if any small code model has been pre-trained with test data from any benchmarks considered. If so, the comparison would not be fair either.
- The same random seed was used across all training runs to facilitate reproducibility. However, this may also constrain the variability in training outcomes, which could be particularly relevant for models with high sensitivity to initialization.
- The optimization objective was set to maximize the F1 score. While this is a common choice, it implicitly balances precision and recall equally, which may not align with the priorities of all clone detection use cases.

6. Conclusions

This research has evaluated a selection of small pre-trained code models for automated clone detection across diverse benchmark datasets. When trained appropriately, the results demonstrate that transformer-based models can deliver high clone detection performance, even under resource-constrained conditions. For example, models such as GraphCodeBERT and UniXCoder have consistently achieved high accuracy and F1 scores across datasets, confirming that compact models can be both practical and efficient in identifying code similarity.

The analysis also revealed some challenges in detecting semantically equivalent clones that differ in syntactic structure. While most models performed well in identifying syntactically similar clones, their performance degraded in datasets involving deeper logical variations, pointing out some limitations of representation learning strategies. These findings indicate the need for novel ways to better capture code intent, functionality, and control flow beyond surface-level patterns.

Future work must explore novel ways to measure how performance changes with code model size and training time, which is relevant for deployment in environments with limited hardware. Moreover, testing with adversarial code will also help assess performance under less controlled conditions.

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] Almatrafi, A. A., Eassa, F. A., & Sharaf, S. A. (2025). Code clone detection techniques based on large language models. *IEEE Access*, . doi:10.1109/ACCESS.2025.3549780.

[3] 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.

[4] 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.

[5] 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.

[6] 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, .

[7] 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.

[8] 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.

[9] Haque, S., Eberhart, Z., Bansal, A., & McMillan, C. (2022). Semantic similarity metrics for evaluating source code summarization. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension* (pp. 36–47). doi:10.1145/3524610.3527909.

[10] Hemberg, E., Moskal, S., & O'Reilly, U. (2024). Evolving code with a large language model. *Genet. Program. Evolvable Mach.*, *25*, 21. doi:10.1007/S10710-024-09494-2.

[11] Husein, R. A., Aburajouh, H., & Catal, C. (2025). Large language models for code completion: A systematic literature review. *Comput. Stand. Interfaces*, *92*, 103917. doi:10.1016/J.CSI.2024.103917.

[12] 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.

[13] 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.

[14] Karnalim, O., Budi, S., Toba, H., & Joy, M. (2019). Source code plagiarism detection in academia with information retrieval: Dataset and the observation. *Informatics in Education*, *18*, 321–344. doi:10.15388/INFEDU.2019.15.

[15] Kaur, M., & Rattan, D. (2023). A systematic literature review on the use of machine learning in code clone research. *Comput. Sci. Rev.*, *47*, 100528. doi:10.1016/J.COSREV.2022.100528.

[16] 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*.

[17] 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*, .

[18] Martinez-Gil, J. (2023). A comparative study of ensemble techniques based on genetic programming: A case study in semantic similarity assessment. *Int. J. Softw. Eng. Knowl. Eng.*, *33*, 289–312. doi:10.1142/S0218194022500772.

[19] Martinez-Gil, J. (2024). Improving source code similarity detection through graphcodebert and integration of additional features. *arXiv preprint arXiv:2408.08903*, .

[20] Martinez-Gil, J. (2024). Source code clone detection using unsupervised similarity measures. In P. Bludau, R. Ramler, D. Winkler, & J. Bergsmann (Eds.), *Software Quality as a Foundation for Security - 16th International Conference on Software Quality, SWQD 2024, Vienna, Austria, April 23-25, 2024, Proceedings* (pp. 21–37). Springer volume 505 of *Lecture Notes in Business Information Processing*. doi:10.1007/978-3-031-56281-5_2.

[21] Martinez-Gil, J. (2025). Advanced detection of source code clones via an ensemble of unsupervised similarity measures. In J. Fischbach, R. Ramler, D. Winkler, & J. Bergsmann (Eds.), *Balancing Software Innovation and Regulatory Compliance - 17th International Conference on Software Quality, SWQD 2025, Munich, Germany, May 20-22, 2025,*

Proceedings (pp. 72–90). Springer volume 544 of *Lecture Notes in Business Information Processing*. URL: https://doi.org/10.1007/978-3-031-89277-6_5. doi:10.1007/978-3-031-89277-6_5.

[22] Martinez-Gil, J. (2025). Augmenting the interpretability of graphcodebert for code similarity tasks. *Int. J. Softw. Eng. Knowl. Eng.*, *35*, 657–678. URL: https://doi.org/10.1142/S0218194025500160. doi:10.1142/S0218194025500160.

[23] Martinez-Gil, J., & Yin, S. (2024). Evaluation of code similarity search strategies in large-scale codebases. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems LVII* (pp. 99–113). Springer. doi:10.1007/978-3-662-70140-9_4.

[24] Mou, L., Li, G., Zhang, L., Wang, T., & Jin, Z. (2016). Convolutional neural networks over tree structures for programming language processing. In D. Schuurmans, & M. P. Wellman (Eds.), *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA* (pp. 1287–1293). AAAI Press. doi:10.1609/AAAI.V30I1.10139.

[25] 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.

[26] Pham, H., Ha, H., Tong, V., Hoang, D., Tran, D., & Le, T. N. (2024). MAGECODE: machine-generated code detection method using large language models. *IEEE Access*, *12*, 190186–190202. doi:10.1109/ACCESS.2024.3509987.

[27] PoolC (2022). 1-fold-clone-detection-600k-5fold. https://huggingface.co/datasets/PoolC/1-fold-clone-detection-600k-5fold.

[28] Ramler, R., Moser, M., Fischer, L., Nissl, M., & Heinzl, R. (2024). Industrial experience report on ai-assisted coding in professional software development. In *LLM4CODEICSE* (pp. 1–7). doi:10.1145/3643795.3648377.

[29] 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.

[30] 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*, .

[31] Svajlenko, J., & Roy, C. K. (2021). Bigclonebench. *Code Clone Analysis: Research, Tools, and Practices*, (pp. 93–105). doi:10.1007/978-981-16-1927-4_7.

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

[33] 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.

[34] 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.

[35] Zhang, X., Chen, Z., Cao, Y., Chen, L., & Zhou, Y. (2024). Multi-intent inline code comment generation via large language model. *Int. J. Softw. Eng. Knowl. Eng.*, *34*, 845–868. doi:10.1142/S0218194024500050.

[36] 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*, .

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

[38] Zhao, G., & Huang, J. (2018). Deepsim: deep learning code functional similarity. In G. T. Leavens, A. Garcia, & C. S. Pasareanu (Eds.), *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018* (pp. 141–151). ACM. doi:10.1145/3236024.3236068.

[39] Zhao, Y., Luo, Z., Tian, Y., Lin, H., Yan, W., Li, A., & Ma, J. (2025). Codejudge-eval: Can large language models be good judges in code understanding? In O. Rambow, L. Wanner, M. Apidianaki, H. Al-Khalifa, B. D. Eugenio, & S. Schockaert (Eds.), *Proceedings of the 31st International Conference on Computational Linguistics, COLING 2025, Abu Dhabi, UAE, January 19-24, 2025* (pp. 73–95). Association for Computational Linguistics. URL: https://aclanthology.org/2025.coling-main.7/.