UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação
Departamento de Sistemas de Computação

Creating a Parallel C to Rust Corpus for Semantic Similarity Evaluation

Advisor: Alneu de Andrade Lopes
Coadvisor: Leonardo Jesus Almeida
Candidate: Vojtěch Hejlek
Institution: Instituto de Ciências Matemáticas e de Computação – ICMC/USP
Department: Departamento de Sistemas de Computação - SSC

São Carlos
June 2026

Contents

Abstract

Cross-language code similarity detection is increasingly critical for code migration, plagiarism detection, and software maintenance. The C-to-Rust language pair is of particular relevance today, yet no aligned parallel corpus exists to evaluate semantic similarity between the two languages. This paper addresses that gap by constructing a curated dataset of 1,886 function-level C–Rust pairs drawn from competitive programming submissions, filtered for correctness, and annotated with difficulty tiers. Four embedding models are benchmarked on the corpus zero-shot, and it is demonstrated that LoRA finetuning of a 125M-parameter open-source model on as few as 1,386 pairs yields a 4.9× improvement in retrieval performance, achieving a discriminative similarity gap comparable to a state-of-the-art commercial system.

1 Introduction

Cross-language code similarity detection identifies functionally equivalent code written in different programming languages, a capability critical for code migration, plagiarism detection, and software maintenance.

The C-to-Rust language pair is highly relevant today, driven by initiatives like DARPA's TRACTOR program [1] to migrate legacy C codebases to memory-safe Rust. However, evaluating semantic similarity between them is challenging due to stark differences in type systems, syntax, and language paradigms.

Despite this need, no aligned C-to-Rust parallel corpus exists for semantic similarity evaluation. Existing large-scale benchmarks like Project CodeNet [12] and xCodeEval [6] offer massive multilingual repositories but lack systematic, function-level alignment for this specific language pair.

This paper addresses that gap by constructing a general-purpose C-to-Rust parallel corpus of aligned function pairs. By focusing on algorithmic logic, the corpus provides a clean baseline to evaluate whether machine learning models can detect semantic equivalence across fundamentally different language paradigms.

2 Related Work

CodeXGLUE [9] is a widely adopted benchmark covering ten program understanding and generation tasks, including a code-to-code translation dataset of over 11,800 function-level Java–C# pairs mined from bilingual open-source projects. While this dataset demonstrates the viability of function-level parallel alignment between syntactically related languages, its coverage is strictly limited to that single language pair and does not generalize to paradigmatically distant languages like C and Rust. **CoST** [18] extended this approach to seven programming languages (C, C++, C#, Java, JavaScript, PHP, and Python) by extracting fine-grained snippet-level parallel data from GeeksForGeeks using comment-based segmentation. Building on CoST, **XLCoST** [17] provides alignment at both the snippet and program levels, yielding over one million parallel snippets across the same seven languages. However, XLCoST completely lacks coverage for Rust.

3 Dataset Construction

This section describes the full pipeline used to produce the aligned C-to-Rust corpus. The data sources and the filtering criteria applied to ensure functional correctness of each submission are described first (Section 3.1), followed by the preprocessing and scoring procedure used to select a single representative implementation per language per problem (Section 3.2). Semantic diversity is then addressed by analysing problem description similarity and removing near-duplicate entries (Section 3.3). Finally, the difficulty categorization scheme and the structure of the released dataset are presented (Sections 3.4–3.5).

3.1 Data Collection and Curation

The source data are structured around competitive programming *problems*, defined as distinct algorithmic tasks with unique test suites. Initial pools were gathered from Project CodeNet [12] (4,053 problems, 13.9M submissions), xCodeEval [6] (7,514 problems, ∼25M submissions), and 30 algorithmic snippets from TheAlgorithms [15] repository. To ensure semantic alignment, the dataset was restricted exclusively to submissions marked as "Accepted" by the online judges. Because these solutions successfully passed rigorous, hidden validation tests, this criterion guarantees that any paired C and Rust implementations for a given problem exhibit functional equivalence despite their structural differences.

Filtering for problems with at least one accepted submission in both target languages yielded 1,958 qualifying problems from Project CodeNet (from 271,137 C and 77,489 Rust submissions) and 3,600 from xCodeEval (from 148,286 C and 29,920 Rust submissions).

3.2 Data Extraction

3.2.1 Preprocessing Pipeline

To guarantee that all code in the corpus is fully functional, all submissions that failed to compile under modern environments were excluded. C code was validated using GCC under the C17 standard [5], and Rust code was verified using Rust version 1.94.0 [14]. This strict filtration eliminates historically accepted online judge solutions that fail to compile locally due to compiler version drift, the deprecation of nonstandard language extensions, or implicit reliance on platform-specific headers.

Following compilation validation, a multistep normalization pipeline isolated the core semantic logic of each submission. First, unused functions, dead code, and redundant macros were removed to prevent unnecessary linter and compiler warnings. For Rust submissions, explicit `#[allow(...)]` attributes were systematically stripped to expose the true nature of the code layout. Finally, all source code comments were removed to eliminate natural language bias, and the remaining code was standardized using native formatting tools, namely `clang-format` [7] for C and `rustfmt` [14] for Rust, to eliminate stylistic and indentation variances across submissions.

As a final preprocessing step, the corpus was filtered to ensure that the core logic of each solution lived inside a single function block. Disregarding the standard `main` wrapper itself, submissions containing either a single function or at most two functions were targeted, where the helper routine was subsequently inlined into the main execution block. Minor constructs such as input parsing macros or basic utility definitions like `MAX` and `MIN` were treated as simple syntactic sugar rather than independent logic and did not disqualify a submission. This consolidation process isolated a single, unambiguous semantic unit for each program, bringing the aligned dataset to 2,013 functional units.

3.2.2 Code Evaluation

Because the corpus requires exactly one C and one Rust submission per problem, a structured evaluation procedure was designed to rank all preprocessed submissions and select the highest-quality representative for each language. Each submission is independently scored along two orthogonal dimensions, namely a *warning penalty* reflecting compiler and linter diagnostics and a *keyword intensity* measuring idiomatic language usage.

3.2.2.1 Compiler and Linter Warnings Penalty

C submissions are analyzed with GCC in syntax check mode under full warning flags (`-Wall -Wextra -Wpedantic -O2`), and are additionally inspected with `clang-tidy` [7] using all available checks. Rust submissions are analyzed with `cargo clippy` [14] configured with `clippy::all` and `clippy::pedantic` lints. For C, a custom static pass also tracks `goto` statements as a dedicated diagnostic.

Importantly, not every diagnostic emitted by these tools contributes to the penalty. Only a curated subset of warning identifiers, selected for their direct relevance to code quality and safety, are assigned weights, while all remaining stylistic or noncritical diagnostics are silently ignored.

The selected warnings are organized into distinct severity categories, each carrying a category multiplier that scales its contribution to the final penalty. For C, the categories and their multipliers span *critical safety* ($\times 10$), *data integrity* ($\times 5$), *program logic* ($\times 3$), *language standards compliance* ($\times 2$), *control flow integrity* ($\times 1.5$), and *maintenance and style* ($\times 0.2$). Representative warnings in the highest severity category include `clang-analyzer-core.NullDereference` and `clang-analyzer-unix.Malloc` from the linter, alongside `-Warray-bounds` and `-Wuse-after-free` from the compiler. Lower severity categories capture minor issues such as misleading indentation or unused parameters.

For Rust, the categories and their multipliers span *memory safety* ($\times 10$), *undefined behavior* ($\times 8$), *logic errors* ($\times 5$), *type safety* ($\times 3$), *control flow warnings* ($\times 2$), and *style and idiom* ($\times 0.5$). The most critical Clippy lints tracked include `clippy::uninit_vec` and `clippy::not_unsafe_ptr_arg_deref`, while compiler-level warnings such as `invalid_reference_casting` and `mutable_transmutes` fall into the memory safety and undefined behavior categories, respectively.

Within each category, each triggered warning identifier carries its own raw weight proportional to its individual severity. The penalty for a given category is computed as the sum of raw weights multiplied by occurrence counts, scaled by the category multiplier. The total linter penalty P_l is the sum of these scaled category scores across all linter categories, and the total compiler penalty P_c is computed analogously from compiler diagnostics.

3.2.2.2 Keyword Intensity

To capture idiomatic language usage, each submission is tokenized and checked against a language-specific keyword weight table. For C, positively weighted tokens include memory management primitives such as `malloc`, `calloc`, and `realloc`, as well as `struct`, `sizeof`, and pointer operators. Negatively weighted tokens include unsafe or outdated patterns such as `gets` and `goto`, and standard conversion constants like `atoi`.

For Rust, positively weighted tokens include idiomatic safe constructs such as `match`, `mut`, smart pointer types (`Box`, `Rc`, `Arc`), and the error propagation `?` operator. Negatively weighted tokens include `unsafe` blocks, `panic!`, `unwrap`, and `expect`, which flag an unhandled departure from Rust's core robust error handling idioms.

Positive and negative raw scores are each normalized by the total token count of the submission and expressed as percentages, yielding a positive intensity metric I^+ and a negative intensity metric I^-.

3.2.2.3 Combined Code Idiomacy Score

To select the single best submission per language per problem, both dimensions are combined into a standardized *Code Idiomacy Score* (CIS). Let P_l and P_c denote the linter and compiler penalties, μ_l and μ_c represent their dataset-wide averages for the respective language, LoC represent the submission's total line count, I^+ represent the positive keyword intensity, and I^- represent the negative keyword intensity. The score is formally defined as:

$$\text{CIS} = \frac{\left(\frac{P_l}{\mu_l} + \frac{P_c}{\mu_c}\right)}{\ln(LoC + 1)} \cdot \frac{1 + I^-}{1 + I^+}$$

A lower CIS is optimal, indicating a more idiomatic, structurally clean, and warning-free submission. Dividing each penalty by its dataset-wide baseline average (μ_l and μ_c) scales the two diagnostic signals into a dimensionless space, rendering them directly comparable regardless of their differing absolute scales.

The baseline log-scale term $\ln(LoC+1)$ normalizes the combined diagnostic penalty by length, ensuring that longer solutions are not mathematically penalized simply for containing a larger volume of safe code. Because $LoC \geq 1$ for any functional execution script, the denominator remains positive and mathematically well defined.

Finally, the keyword adjustment factor $\frac{1+I^-}{1+I^+}$ scales the final score based on structural idiomatic design choices. When a submission is stylistically neutral ($I^- = I^+ = 0$), the scaling factor reduces to 1 and has no numerical effect; a prominent negative keyword intensity raises the final score to penalize unsafe architectural workarounds, while a dominant positive keyword intensity decreases the score to reward high idiomatic language fluency.

3.3 Problem Uniqueness and Semantic Diversity

Because the source pools were aggregated from multiple distinct competitive programming platforms, many tasks inevitably share overlapping or near-identical natural language problem descriptions. To ensure the final dataset is semantically diverse and free of redundancies, the text of all 2,013 raw problem descriptions was analyzed. Dense textual vectors generated by OpenAI's `text-embedding-3-large` model [11] were used to compute the global pairwise cosine similarity distribution across these descriptions. The initial results indicated a generally diverse dataset, yielding a mean pairwise text similarity of 0.333 (median $= 0.327, \sigma = 0.103$).

To eliminate descriptions that are overlapping, filtering was applied using a two-step approach based on similarity scores. First, pairs with a cosine similarity at or above 0.90 were classified as duplicates, resolved automatically by retaining only the harder algorithmic variant. Second, pairs with similarity scores between 0.80 and 0.90 were manually inspected to distinguish between shared narrative metaphors and true algorithmic duplicates. This process successfully removed 127 redundant entries, ensuring that 93.3% of the finalized problem statements sit safely below a 0.80 similarity threshold.

3.4 Categorization

To evaluate model performance across varying levels of cross-language alignment difficulty, the 1,886 problems in the cleaned dataset were partitioned into three difficulty tiers. Because the grading systems of the original source platforms are incompatible, an origin-agnostic method was required.

Each pair was scored by the zero-shot SFR-Embedding-Code-400M_R [8] model, measuring the cosine similarity it assigns to the C and Rust embeddings. Quantile-based thresholds at the 25th and 75th percentiles of the similarity distribution partition the corpus into three balanced tiers.

Category	SFR Similarity	Count
Easy	≥ 0.868	472 (25%)
Medium	0.781–0.868	943 (50%)
Hard	< 0.781	471 (25%)

Table 1: Difficulty tiers defined by SFR-Embedding-Code-400M_R zero-shot cosine similarity.

The tiers measure how hard each pair is for a generic pretrained model to align, rather than any intrinsic property of the source problems or their origin platforms.

3.5 Final Dataset Structure

The finalized corpus is distributed as a single JSONL file containing 1,886 entries. Each line is a self-contained JSON object representing one aligned problem pair.

Field	Description
`problem_id`	Zero-padded sequential identifier (`0001`–`1886`)
`problem_description`	Full problem statement as provided by the source platform
`c_code`	C implementation
`rust_code`	Rust implementation
`difficulty`	Difficulty tier: `easy`, `medium`, or `hard`

Table 2: Schema of the dataset.

All function names have been normalized to `solution` in both languages, removing source-specific naming conventions and ensuring that function identifiers do not constitute a retrieval signal. The `difficulty` field is derived directly from the categorization procedure described in Section 3.4 and is included as a convenience for category-based evaluation.

4 Evaluation

This section evaluates model performance on the corpus. Zero-shot baselines are established first, covering how well each model performs without any finetuning (Section 4.2), preceded by an introduction of the models used throughout the evaluation (Section 4.1). Finetuning experiments using UniXcoder with LoRA adaptation are then presented, including an ablation over training set size and a per-difficulty-category breakdown (Section 4.3). The section concludes with a comparison against voyage-code-3, which contextualises the finetuned model's performance against a purpose-built commercial system (Section 4.4).

4.1 Models Used

Four pretrained embedding models are used across the validation and evaluation experiments. All models produce fixed-length vector representations of code that can be compared via cosine similarity.

CodeBERT [2] (Microsoft, 125M parameters) is a bimodal transformer pretrained on natural language and code pairs across six programming languages using masked language modelling and replaced token detection. It has a context window of 512 tokens. It serves as a saturation baseline evaluated zero-shot.

UniXcoder [3] (Microsoft, 125M parameters) is a direct successor to CodeBERT, sharing the same RoBERTa-based architecture and parameter scale. It extends the CodeBERT pretraining approach by additionally incorporating flattened abstract syntax tree structures, allowing it to capture code syntax more explicitly. Its context window of 1,024 tokens is double that of CodeBERT. It is used both as the zero-shot baseline and as the base model for finetuning.

SFR-Embedding-Code-400M_R [8] (Salesforce, 434M parameters) is trained using a multitask contrastive learning framework optimised for multilingual code-to-code retrieval. It supports a context window of 8,192 tokens. It serves both as the difficulty categorisation model (Section 3.4) and as a zero-shot validation baseline.

voyage-code-3 [16] (Voyage AI, architecture and parameter count undisclosed) is a state-of-the-art commercial embedding API purpose-built for cross-language code retrieval at repository scale. It is evaluated zero-shot as an external upper-bound reference, representing what a well-resourced, production-grade system trained on far larger data achieves on the same task.

4.2 Zero-Shot Baselines

The performance of three models on the corpus was measured without any finetuning, namely CodeBERT [2], SFR-Embedding-Code-400M_R [8], and UniXcoder [3]. For every positive pair in the full 2,013-problem dataset, the cosine similarity between the C and Rust embeddings produced by each model was computed. Table 3 reports these results.

Model	Mean	Std	Min	Max
CodeBERT	0.991	0.004	0.961	0.997
UniXcoder	0.731	0.094	0.308	0.935
SFR-Code-400M	0.823	0.062	0.604	0.962

Table 3: Cosine similarity metrics for aligned (positive) C–Rust pairs.

To assess discriminative power, each C embedding was also paired with five randomly selected Rust embeddings from different problems, producing 9,430 mismatched pairs per model.

Model	Pos mean	Neg mean	Gap	Neg > Pos mean
CodeBERT	0.991	0.970	+0.009	41.4%
UniXcoder	0.731	0.539	+0.076	27.7%
SFR-Code-400M	0.823	0.654	+0.176	0.2%

Table 4: Discriminative performance across models. *Neg > Pos mean* indicates the percentage of mismatched pairs that outscored the model's average positive similarity.

CodeBERT's near-uniform positive similarity (mean = 0.991) is a consequence of two compounding factors. First, its cross-language embedding space collapses into a narrow band near 1.0 regardless of input, making discrimination impossible. Second, its 512-token context window causes longer function bodies to be silently truncated, so the model often compares only the opening lines of each implementation rather than the full algorithmic logic. The result is that 41.4% of random mismatched pairs score above the mean positive.

UniXcoder's 1,024-token window reduces truncation for most functions in the corpus, yet it still provides only limited zero-shot discriminative signal, with a gap of +0.076 and 27.7% of mismatches beating the mean positive. This reflects the inherent

difficulty of cross-language alignment without domain-specific training rather than a context length limitation. SFR-Code-400M, with its 8,192-token window, can embed even the longest functions in full and shows the strongest zero-shot discrimination, with a gap of +0.176 and only 0.2% of mismatches exceeding the mean positive.

The results reveal a clear spread in zero-shot cross-language discriminability across model families. SFR-Code-400M separates correct from mismatched pairs most reliably, while UniXcoder and CodeBERT leave the distributions largely overlapping.

4.3 Finetuning Experiments

4.3.1 Experimental Setup

UniXcoder was selected as the base model for finetuning, as its encoder-only architecture and pretraining on multi-language code make it a suitable starting point for cross-language embedding tasks. The cleaned 1,886-problem dataset was split into three non-overlapping subsets, comprising 1,386 training pairs, 200 validation pairs, and 300 test pairs. The validation set is used exclusively for checkpoint selection during training: after each epoch the model is evaluated on it, and the checkpoint with the highest validation MRR@10 (Mean Reciprocal Rank at 10) is retained. The test set is held out entirely and evaluated only once, after the best checkpoint has been selected. The resulting splits are approximately balanced, with easy, medium, and hard pairs each accounting for roughly 25%, 50%, and 25% of every subset, consistent with the full dataset distribution. The model was finetuned using a symmetric InfoNCE contrastive loss with temperature $\tau = 0.07$ [10, 13], treating the positive pair as the target and all other pairs within the batch as in-batch negatives. Each batch of size 8 thus yields 8 positive signals and 56 implicit negatives per step.

Finetuning was performed using LoRA (Low-Rank Adaptation) [4], which freezes all 126M base model param eters and injects small trainable rank-decomposition matrices into the query and value projection layers of the attention blocks. With rank $r = 8$ and scaling factor $\alpha = 16$, this adds only 294,912 trainable parameters, just 0.23% of the total model size. This parameter-efficient approach limits peak memory usage and is appropriate for the scale of the available training data.

4.3.2 Baseline Performance

Prior to any finetuning, the pretrained UniXcoder model was evaluated zero-shot on the 300-problem test set. The evaluation uses three retrieval metrics. **MRR@10** (Mean Reciprocal Rank at 10) embeds each C implementation and ranks all 300 Rust candidates by cosine similarity; the reciprocal of the correct match's rank, capped at 10, is then averaged across all queries. **R@1** and **R@5** are recall metrics measuring the fraction of problems where the correct Rust match appears at rank 1 or within the top 5, respectively.

Split	MRR@10	R@1	R@5	Pos sim	Neg sim
Validation (200)	0.207	0.150	0.235	0.617	0.525
Test (300)	0.150	0.100	0.180	0.615	0.539

Table 5: Zero-shot UniXcoder baseline retrieval performance.

On the test set, MRR@10 = 0.150 means the correct Rust match is ranked approximately seventh on average among 300 candidates, only marginally above random. R@1 = 0.100 means the correct match appears at rank 1 in only 1 in 10 problems. Pos sim and Neg sim report the mean cosine similarity across all correct pairs and all mismatched pairs respectively, and the gap between them is only 0.077. At this level of overlap, the model cannot reliably distinguish a correct C-to-Rust match from a random one.

4.3.3 Effect of Training Data Size

To characterise how retrieval performance scales with available training data, six LoRA models were trained on progressively larger subsets of the training set, specifically 100, 300, 500, 750, 1,000, and the full 1,386 pairs. Each model was trained for 3 epochs to isolate the effect of data quantity from training duration. All models were evaluated on the same 300-problem test pool.

Training pairs	MRR@10	R@1	R@5	Sim gap
Baseline (0)	0.150	0.100	0.180	+0.077
100	0.282	0.207	0.337	+0.122
300	0.493	0.400	0.593	+0.301
500	0.577	0.490	0.670	+0.353
750	0.637	0.553	0.730	+0.404
1000	0.658	0.577	0.743	+0.435
1386 (full)	0.716	0.650	0.793	+0.494

Table 6: UniXcoder LoRA finetuning performance on the 300-problem test set as a function of training set size. All models trained for 3 epochs.

Even 100 training pairs are sufficient to nearly double MRR@10 from 0.150 to 0.282, demonstrating that the pretrained model already understands code structure

and requires only limited signal to learn the cross-language correspondence. The improvement is steep in the low-data regime: 300 pairs cross the MRR@10 = 0.5 threshold, recovering roughly half of the achievable performance from 13% of the training data. Returns diminish significantly above 750 pairs, with the 1,000-to-full-data increment contributing only 0.058 MRR, suggesting that a training set of 1,386 pairs is comfortably above the point of saturation for this architecture and task.

4.3.4 Full Training Run Results

Training the LoRA model for 5 epochs on the full 1,386-pair training set yields MRR@10 = 0.729 on the 300-problem test set. The best checkpoint was found at epoch 5, selected by validation MRR@10 (0.774), with performance improving steadily across all five epochs.

Model	MRR@10	R@1	R@5	Pos sim	Neg sim	Gap
UniXcoder (baseline)	0.150	0.100	0.180	0.615	0.539	+0.077
UniXcoder + LoRA	0.729	0.643	0.827	0.621	0.087	+0.533

Table 7: Baseline versus LoRA-finetuned UniXcoder on the 300-problem test set.

Finetuning increases MRR@10 from 0.150 to 0.729, a 4.9× improvement. The primary mechanism is the collapse of the negative similarity distribution: the mean negative similarity drops from 0.539 to 0.087. Contrastive training actively pushes mismatched pairs toward near-zero rather than simply elevating positive pairs, producing a far wider margin between the two distributions. The positive mean similarity shifts only marginally (0.615 → 0.621), confirming that discrimination improves by reorganizing the embedding space around correct alignments rather than by pulling all positives upward uniformly.

4.3.5 Per-Category Results

To assess whether finetuning generalises across difficulty tiers, both the baseline and LoRA models were evaluated separately on the Easy, Medium, and Hard subsets of the 300 categorised test pairs, embedded together in a single retrieval pool.

Category	Model	MRR@10	R@1	R@5
Easy	Baseline	0.420	0.310	0.549
	LoRA	0.846	0.803	0.901
Medium	Baseline	0.084	0.042	0.085
	LoRA	0.710	0.620	0.817
Hard	Baseline	0.011	0.000	0.000
	LoRA	0.643	0.515	0.818

Table 8: Per-category retrieval performance on a 300-pair test set.

The baseline model completely fails on Hard pairs: R@1 = 0.000 and R@5 = 0.000, meaning not a single Hard pair appears in the top five out of 300 candidates before

finetuning. This confirms that the categorization captures genuine variation in model-relative alignment difficulty. After finetuning, the Hard category shows the largest absolute gain, with MRR@10 rising from 0.011 to 0.643, a 58× improvement. The absolute MRR lift is larger on Hard (+0.632) than on Easy (+0.426), indicating that finetuning rescues pairs the pretrained model cannot resolve at all, rather than merely reinforcing those it already handles adequately.

4.4 Comparison with Commercial Embedding Models

To contextualise the finetuned model's performance, voyage-code-3 [16] was evaluated zero-shot on the same 300-problem test set. voyage-code-3 is a purpose-built commercial code embedding API designed for cross-language retrieval at repository scale. It is evaluated without any exposure to the training data used in this work.

Model	MRR@10	R@1	R@5	Pos sim	Neg sim	Gap
voyage-code-3	0.977	0.960	1.000	0.805	0.296	+0.509
UniXcoder + LoRA	0.729	0.643	0.827	0.621	0.087	+0.533
UniXcoder (baseline)	0.150	0.100	0.180	0.615	0.539	+0.077

Table 9: Comparison of voyage-code-3 against UniXcoder on the 300-problem test set.

voyage-code-3 achieves near-perfect retrieval, reflecting the scale and specialisation of a state-of-the-art commercial system. What is notable, however, is that a 125M-parameter open-source model finetuned on 1,386 pairs produces a similarity gap of +0.533. Both models push mismatched pairs away from correct pairs with comparable force, despite the orders-of-magnitude difference in likely training data and model scale.

The two models achieve this through different geometric strategies. voyage-code-3 pulls correct pairs upward into a high-similarity region (mean pos sim = 0.805), but its negative distribution remains relatively high (mean = 0.296, max = 0.863), meaning some mismatched pairs score near 0.86, above the weakest correct pairs. The finetuned UniXcoder takes the opposite approach: correct pair similarity stays modest (mean = 0.621) while mismatched pairs are actively suppressed (mean = 0.087, min = −0.350).

5 Data and Model Availability

The finalized general-purpose C-to-Rust parallel corpus constructed in this work is publicly available on Hugging Face at https://huggingface.co/datasets/hejlevoj/C-Rust-parallel-corpus. Additionally, the best-performing LoRA-finetuned UniXcoder model checkpoint evaluated in Section 4.3 is hosted and available for reuse at https://huggingface.co/hejlevoj/UniXCoder-C-Rust-semantic-similarity.

6 Conclusion

This work addresses the absence of a dedicated parallel corpus for C-to-Rust semantic similarity evaluation by constructing a dataset of 1,886 aligned, function-level code pairs. The construction pipeline combines correctness guarantees from online judge verdicts with local compilation validation, a principled idiomacy-based selection criterion, and dense-embedding deduplication of problem descriptions. The result is a semantically diverse, structurally clean corpus annotated with difficulty tiers that reflect genuine variation in cross-language alignment challenge.

The evaluation experiments demonstrate that the corpus is both discriminative and practically useful. Zero-shot baselines expose substantial differences in cross-language discriminability across model families, confirming that the task is non-trivial and that model architecture and context window size have measurable impact on performance. Finetuning experiments show that even modest amounts of aligned training data produce dramatic improvements: 100 pairs nearly double MRR@10, 300 pairs recover more than half of total achievable performance, and the full 1,386-pair training set yields a 4.9× overall improvement over the zero-shot baseline. Crucially, finetuning generalises across difficulty tiers, including Hard pairs that the pretrained model resolves at essentially random chance, demonstrating that the difficulty categorization captures genuine variation rather than noise.

Comparison with voyage-code-3 places the finetuned model in broader context. Despite the commercial system's near-perfect retrieval (MRR@10 = 0.977), the finetuned UniXcoder achieves a comparable positive-to-negative similarity gap (+0.533 vs +0.509), through an opposite geometric strategy.

Several directions remain open for future work. The corpus is drawn exclusively from competitive programming platforms, which favour short, algorithmically focused solutions. Extending the corpus with systems programming source material, such as OS kernels, embedded drivers, or network libraries, would better capture the structural divergence between the two languages and produce a more challenging and representative benchmark, one that surfaces the ownership and borrowing idioms central to idiomatic Rust.

Nevertheless, the results demonstrate that a small, carefully curated parallel corpus is sufficient to drive substantial gains in cross-language embedding quality.

References

[1] Defense Advanced Research Projects Agency. Translating all c to rust (tractor). https://www.darpa.mil/program/translating-all-c-to-rust, 2024.

[2] Feng, Z., et al. Codebert: A pre-trained model for programming and natural languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (2020).

[3] Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, L., Zhou, J., Narayanan, S., Mueller, M., Wu, L., Zhou, J., Mohapatra, S., Lin, S., Sun, M., and Zhou, M. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics* (2022).

[4] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)* (2022).

[5] ISO/IEC. Programming languages — C: ISO/IEC 9899:2018. Standard ISO/IEC 9899:2018, International Organization for Standardization, 2018.

[6] Khan, M. A. M., Bari, M. S., Do, X. L., Wang, W., Parvez, M. R., and Joty, S. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. *arXiv preprint arXiv:2303.03004* (2023).

[7] Lattner, C., and Adve, V. Llvm: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization (CGO)* (2004), pp. 75–86.

[8] Liu, Y., Meng, R., Joty, S., Savarese, S., Xiong, C., Zhou, Y., and Yavuz, S. Codexembed: A generalist embedding model family for multilingual and multi-task code retrieval. *arXiv preprint arXiv:2411.12644* (2024).

[9] Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C. B., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S. K., Fu, S., and Liu, S. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[10] Oord, A. v. d., Li, Y., and Vinyals, O. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748* (2018).

[11] OpenAI. New embedding models and api updates. https://openai.com/blog/new-embedding-models-and-api-updates, 2024.

[12] Puri, R., et al. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. In *NeurIPS Datasets and Benchmarks Track* (2021).

[13] Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., et al. Learning transferable visual models from natural language supervision. In *International Conference on Machine Learning (ICML)* (2021).

[14] The Rust Project Developers. The rust programming language — reference. https://doc.rust-lang.org/reference/, 2025. Accessed: June 2026.

[15] TheAlgorithms contributors. Thealgorithms: Open source resource for learning data structures and algorithms. https://github.com/TheAlgorithms, 2020.

[16] Voyage AI. voyage-code-3: More accurate code retrieval with lower dimensional, quantized embeddings, 2024.

[17] Zhu, M., Jain, A., Suresh, K., Ravindran, R., Tipirneni, S., and Reddy, C. K. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474* (2022).

[18] Zhu, M., Suresh, K., and Reddy, C. K. Multilingual code snippets training for program translation. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2022), vol. 36, pp. 11783–11790.