# CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code

**Shuyan Zhou**[†*] **Uri Alon**[†*] **Sumit Agarwal**[†] **Graham Neubig**[†‡]
[†]Language Technologies Institute, Carnegie Mellon University
[‡]Inspired Cognition
{shuyanzh,ualon,sumita,gneubig}@cs.cmu.edu

## Abstract

Since the rise of neural models of code that can generate long expressions and statements rather than a single next-token, one of the major problems has been reliably evaluating their generated output. In this paper, we propose CodeBERTScore: an automatic evaluation metric for code generation, which builds on BERTScore (Zhang et al., 2020). Instead of measuring *exact* token matching as BLEU, CodeBERTScore computes a soft similarity score between each token in the generated code and in the reference code, using the contextual encodings of large pretrained models such as CodeBERT (Feng et al., 2020). Further, instead of encoding only the generated tokens as in BERTScore, CodeBERTScore also encodes the programmatic context surrounding the generated code.

We perform an extensive evaluation of CodeBERTScore across four programming languages. We find that CodeBERTScore achieves a higher correlation with human preference and with functional correctness than all existing metrics. That is, generated code that receives a higher score by CodeBERTScore is more likely to be preferred by humans, as well as to function correctly when executed. Finally, while CodeBERTScore can be used with a multilingual CodeBERT as its base model, we release five language-specific pretrained models to use with our publicly available code[1]. Our language-specific models have been downloaded more than **500,000** times from the Huggingface Hub.

## 1 Introduction

Models of code have seen increasing accuracy in the past decade (Hindle et al., 2016; Raychev et al., 2014; Alon et al., 2020; Allamanis et al., 2018) for a variety of tasks such code completion (Fried et al., 2022), code fixing (Allamanis et al., 2021; Brody et al., 2020; Berabi et al., 2021), natural-language-to-code generation (Yin & Neubig, 2017; Zhou et al., 2023) and more (Alon et al., 2019a;b). More recently, the rise of large language models (LLMs) of natural language (Devlin et al., 2019; Brown et al., 2020) has unlocked the application of such models to code (Wang et al., 2021; Austin et al., 2021; Chen et al., 2021). LLMs of code have reached such a high accuracy that they are now useful for the broad programming audience and actually save developers' time when implemented in tools such as GitHub's Copilot. This sharp rise in LLMs' usability was achieved thanks to their ability to accurately generate *long* completions, which span multiple tokens and even lines, rather than only a single next-token as in early models (Allamanis & Sutton, 2013; Hellendoorn & Devanbu, 2017). Nevertheless, evaluating and comparing different models has remained a challenging problem (Xu et al., 2022) that requires an accurate and reliable evaluation metric for the quality of the models' generated outputs, and existing metrics are sub-optimal.

**Existing Evaluation Approaches** The most common evaluation metrics are token-matching methods such as BLEU (Papineni et al., 2002), adopted from natural language processing. These metrics are based on counting overlapping n-grams in the generated code and the reference code.

* Equal contribution.
[1]Our anonymized code is available at https://github.com/neulab/code-bert-score

**Reference:**

```
int f(Object target) {
  int i = 0;
  for (Object elem: this.elements) {
    if (elem.equals(target)) {
      return i;
    }
    i++;
  }
  return -1;
}
```

(a) The ground truth reference – find *the index* of `target` in `this.elements`.

**Non-equivalent candidate:**

```
boolean f(Object target) {
  for (Object elem: this.elements) {
    if (elem.equals(target)) {
      return true;
    }
  }

  return false;
}
```

(b) **Preferred by BLEU & CrystalBLEU** – find *whether or not* `target` is in `this.elements`.

**Equivalent candidate:**

```
int f(Object target) {
  for (int i=0; i<this.elements.size(); i++) {
    Object elem = this.elements.get(i);
    if (elem.equals(target)) {
      return i;
    }
  }
  return -1;
}
```

(c) **Preferred by CodeBERTScore** – find *the index* of `target` in `this.elements`.

Figure 1: An intuitive example for the usefulness of CodeBERTScore in measuring generated code: Figure 1(a) shows a reference code snippet in Java. This reference can be the ground truth code given as part of a test set. Figure 1(b) and Figure 1(c) show two generated predictions. Among these two candidates and given the reference, both BLEU (Papineni et al., 2002) and CrystalBLEU (Eghbali & Pradel, 2022) prefer (score higher) the snippet in Figure 1(b), which *is not* functionally equivalent to the reference, while our proposed CodeBERTScore prefers the code in Figure 1(c), which *is* functionally equivalent to the code in Figure 1(a).

CrystalBLEU (Eghbali & Pradel, 2022) has recently extended BLEU by ignoring the 500 most occurring n-grams, arguing that they are trivially shared between the prediction and the reference. Nonetheless, both BLEU and CrystalBLEU rely on the lexical *exact match* of tokens, which does not account for diversity in implementation, variable names, and code conventions. Figure 1 shows an example: given the reference code in Figure 1(a), both BLEU and CrystalBLEU prefer (rank higher) *the non-equivalent* code in Figure 1(b) over the functionally equivalent code in Figure 1(c).

CodeBLEU (Ren et al., 2020) attempts to lower the requirement for lexical exact match, by relying on data-flow and Abstract Syntax Tree (AST) matching as well; nevertheless, valid generations may have different ASTs and data-flow from the reference code, which may lead to low CodeBLEU score even when the prediction is correct. Further, *partial* predictions may be useful for a programmer, but accepting them may lead to partial code that does not parse, and thus cannot be fully evaluated by CodeBLEU. Execution-based evaluation attempts to address these problems by running tests on the generated code to verify its functional correctness (Chen et al., 2021; Athiwaratkun et al., 2022; Li et al., 2022; Wang et al., 2022; Lai et al., 2022). This provides a direct measure of the functionality of the generated code, while being agnostic to diversity in implementation and style. However, execution-based evaluation requires datasets that are provided with hand-written test cases for each example, which is costly and labor-intensive to create; thus, only few such datasets exist. Further, executing model-generated code is susceptible to security threats,[2] and thus should be run in an isolated sandbox, which makes it technically cumbersome to work with.

**Our Approach** In this work, we introduce CodeBERTScore, an evaluation metric for code generation, leveraging pretrained models such as CodeBERT (Feng et al., 2020), and adopting best practices from natural language generation evaluation (Zhang et al., 2020). First, CodeBERTScore encodes the generated code and the reference code independently with pretrained models, *with* the inclusion of natural language context if available. Then, we compute the dot-product similarity between the encoded representations of each token in the generated code and the reference code. Finally, the best matching token vector pairs are used to compute precision and recall. Code-

[2]https://twitter.com/ludwig_stumpp/status/1619701277419794435?s=46&t=5vlJ05ph9UuGxQc17TXcrA

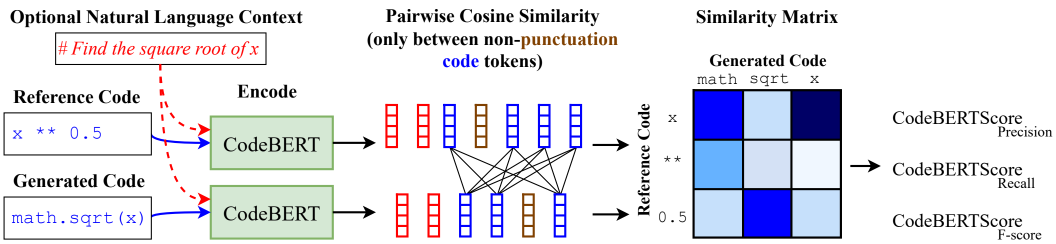



Figure 2: A diagram illustrating CodeBERTScore: We use a language-specific CodeBERT model to encode each of ⟨*natural_language*, *reference_code*⟩ and ⟨*natural_language*, *generated_code*⟩. We then compute the pairwise cosine similarity between every encoded token in the reference and every encoded token in the generated code, ignoring the encoded natural language context tokens and encoded punctuation tokens; finally, we take the $max$ across the rows of the resulting matrix to compute *Precision* (`np.mean(np.max(similarity_matrix, axis=0))`), and across columns to compute *Recall* (`np.mean(np.max(similarity_matrix, axis=1))`).

BERTScore allows comparing code pairs that are lexically different, while taking into account the (1) natural language context, if such provided; the (2) contextual information of each token; and (3) implementation diversity. Our approach is illustrated in Figure 2.

**Example** A concrete example is shown in Figure 1: while BLEU and CodeBLEU prefer *the non-equivalent* code in Figure 1(b) given the reference code in Figure 1(a), CodeBERTScore prefers the code in Figure 1(c), which *is* functionally equivalent to the reference.

**Contributions** In summary, our main contributions are: (a) a new metric for code similarity, adopted from natural language processing (NLP), which leverages the benefits of pretrained models, while not requiring manual labeling or annotation; (b) an extensive evaluation across four programming languages, showing that CodeBERTScore is more correlated with human preference *and* more correlated with execution correctness than all previous approaches including BLEU, CodeBLEU, and CrystalBLEU; (c) we pretrain and release five language-specific CodeBERT models to use with our publicly available code, for Java, Python, C, C++, and JavaScript. As of the time of this submission, our models have been downloaded from the Huggingface Hub more than **500,000** times.

## 2 EVALUATING GENERATED CODE

### 2.1 PROBLEM FORMULATION

Given a context $x \in \mathcal{X}$ (*e.g.,* a natural language comment, or the surrounding code context), a code generation model $\mathcal{M} : \mathcal{X} \rightarrow \mathcal{Y}$ produces a code snippet $\hat{y} \in \mathcal{Y}$ by conditioning on the intent specified by $x$. The quality of the generation is evaluated by comparing $\hat{y} \in \mathcal{Y}$ with the reference implementation $y^* \in \mathcal{Y}$, using a metric function $f : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}$, essentially computing $f(\hat{y}, y^*)$.

A larger value of $f(\hat{y}, y^*)$ indicates that the generated code is more accurate with respect to the reference code, and the way $f$ ranks different candidates is more important than the absolute value of $f(\hat{y}, y^*)$. That is, ideally, if a prediction $\hat{y}_1$ is more functionally equivalent to $y^*$ and more preferable by human programmers over a prediction $\hat{y}_2$, we wish that a good metric would rank $\hat{y}_1$ higher than $\hat{y}_2$. That is, we look for an $f$ function such that $f(\hat{y}_1, y^*) > f(\hat{y}_2, y^*)$.

### 2.2 CODEBERTSCORE

BERTScore Zhang et al. (2020) was proposed as a method for evaluating mainly machine translation outputs. The idea in BERTScore is to encode the candidate and the reference separately using a BERT-based model, and provide a score according to the similarity between individual encoded tokens. Our approach generally follows BERTScore, with the following main differences: (a) We encode the context (e.g., natural language input) along with each of the generated and reference code snippets, but without using the encoded context in the final similarity computation, essentially computing $f(\hat{y}, y^*, x)$ rather than $f(\hat{y}, y^*)$; (b) we found that masking punctuation tokens (such as parentheses, brackets, dots) from the final computation also helps improve the correlation with human preference and functional correctness; (c) given CodeBERTScore's precision and recall, instead of computing the $F_1$ score, we also compute $F_3$ to weigh recall higher than precision, following METEOR (Banerjee & Lavie, 2005).

We use a pretrained model $\mathcal{B}$ to encode the reference and candidate. In our experiments, $\mathcal{B}$ is a CodeBERT model that is further pretrained using the masked language modeling objective (Devlin et al., 2019) on language-specific corpora, but $\mathcal{B}$ can be any transformer-based model.

**Token Representation** We concatenate the context $x$ with each of the reference and the candidate, resulting in $x \cdot y^*$ and $x \cdot \hat{y}$. We use the tokenizer $\mathcal{T}_{\mathcal{B}}$ provided with the model $\mathcal{B}$ to get the sequences of tokens $\mathcal{T}_{\mathcal{B}}(x \cdot y^*) = \langle x_1, ..., x_k, y_1^*, ..., y_m^* \rangle$ and $\mathcal{T}_{\mathcal{B}}(x \cdot \hat{y}) = \langle x_1, ..., x_k, \hat{y}_1, ..., \hat{y}_n \rangle$. We perform a standard forward pass with the model $\mathcal{B}$ for each of the tokenized sequences, resulting in sequences of vectors $\langle \boldsymbol{x}_1, ..., \boldsymbol{x}_k, \boldsymbol{y}_1^*, ..., \boldsymbol{y}_m^* \rangle$ and $\langle \boldsymbol{x}_1, ..., \boldsymbol{x}_k, \hat{\boldsymbol{y}}_1, ..., \hat{\boldsymbol{y}}_n \rangle$. Finally, we mask out the encoded context tokens $\boldsymbol{x}_1, ..., \boldsymbol{x}_k$ as well as masking out all non-alphanumeric tokens (parentheses, brackets, dots, commas, whitespaces, etc.) except for arithmetic operators, from each of the encoded reference and encoded candidate. This results in encoded reference tokens $\boldsymbol{y}^* = \langle \boldsymbol{y}_1^*, ..., \boldsymbol{y}_m^* \rangle$, encoded candidate tokens $\hat{\boldsymbol{y}} = \langle \hat{\boldsymbol{y}}_1, ..., \hat{\boldsymbol{y}}_n \rangle$, and their corresponding masks $\boldsymbol{m}^*$ and $\hat{\boldsymbol{m}}$. We denote $\boldsymbol{y}[\boldsymbol{m}]$ as the remaining encoded tokens in $\boldsymbol{y}$ after selecting only the alphanumeric token vectors according to the mask $\boldsymbol{m}$.

**Similarity Computation** We compute the cosine similarity between the encoded reference and encoded candidate tokens, following Zhang et al. (2020): $sim(y_i^*, \hat{y}_j) = \frac{{\boldsymbol{y}_i^*}^\top \cdot \hat{\boldsymbol{y}}_j}{\|\boldsymbol{y}_i^*\| \cdot \|\hat{\boldsymbol{y}}_j\|}$. Although this compares the individual tokens $y_i^*$ and $\hat{y}_j$, their vector representations $\boldsymbol{y}_i^*$ and $\hat{\boldsymbol{y}}_j$ contain information about their context, and thus about their semantic role in the code.

**CodeBERTScore** We use the similarity scores to compute precision, recall, and $F_1$, by taking the maximum across the rows and columns of the similarity matrix, and then averaging. Following Banerjee & Lavie (2005), we also compute $F_3$ by giving more weight to recall:

$$
\begin{aligned}
\text{CodeBERTScore}_{\text{P}} &= \frac{1}{|\hat{y}[\hat{\boldsymbol{m}}]|} \sum_{\hat{y}_j \in \hat{y}[\hat{\boldsymbol{m}}]} \max_{y_i^* \in y^*[\boldsymbol{m}^*]} sim(y_i^*, \hat{y}_j) \\
\text{CodeBERTScore}_{\text{R}} &= \frac{1}{|y^*[\boldsymbol{m}]|} \sum_{y_i^* \in y^*[\boldsymbol{m}^*]} \max_{\hat{y}_j \in \hat{y}[\hat{\boldsymbol{m}}]} sim(y_i^*, \hat{y}_j) \\
\text{CodeBERTScore}_{\text{F}_1} &= \frac{2 \cdot \text{CodeBERTScore}_{\text{P}} \cdot \text{CodeBERTScore}_{\text{R}}}{\text{CodeBERTScore}_{\text{P}} + \text{CodeBERTScore}_{\text{R}}} \\
\text{CodeBERTScore}_{\text{F}_3} &= \frac{10 \cdot \text{CodeBERTScore}_{\text{P}} \cdot \text{CodeBERTScore}_{\text{R}}}{9 \cdot \text{CodeBERTScore}_{\text{P}} + \text{CodeBERTScore}_{\text{R}}}
\end{aligned}
$$

**Token Weighting** Following Zhang et al. (2020), we compute the inverse document frequency (idf) according to the test set, and weigh each token according to its negative log frequency, instead of computing a standard average.

## 3 Experimental Setup

We evaluate CodeBERTScore across multiple datasets and programming languages. We first show that CodeBERTScore is more correlated with *human preference* than previous metrics, using human-rated solutions for the CoNaLa dataset (Yin et al., 2018b; Evtikhiev et al., 2022). We then show that CodeBERTScore is more correlated with *functional correctness*, using the HumanEval dataset (Chen et al., 2021). Finally, we show that CodeBERTScore achieves a higher *distinguishability* than CrystalBLEU (Eghbali & Pradel, 2022), which proposed this meta-metric.

### 3.1 Training Language-specific CodeBERT models

**Training** We took CodeBERT (Feng et al., 2020) as our base model and continued its pretraining (Gururangan et al., 2020) with the masked language modeling (MLM) objective (Devlin et al., 2019) on Python, Java, C++, C, and JavaScript corpora. We trained a separate model for each programming language, for 1,000,000 steps for each language, using a batch size of 32, an initial learning rate of $5e^{-5}$, decayed linearly to $3e^{-5}$. Our implementation is based on the widely used HuggingFace `transformers` library (Wolf et al., 2019) and BERTScore, and it can be used with any transformer-based model available in the HuggingFace hub.

**Dataset** We trained each model on the language-specific subset of the CodeParrot (Tunstall et al., 2022) dataset[3], which consists of overall 115M code files from GitHub, and further filtered by keeping only files having average line length lower than 100, more than 25% alphanumeric characters, and non-auto-generated files. Even after 1,000,000 training steps, none of the models have completed even a single epoch, meaning that every training example was seen only once at most.

### 3.2 Comparing Different Metrics

**Human Preference Experiments** We evaluate on CoNaLa (Yin et al., 2018a), a natural language to Python code generation benchmark collected from StackOverflow. We use the human annotation released by Evtikhiev et al. (2022) to measure the correlation between each metric and human preference. For each example, Evtikhiev et al. (2022) asked experienced software developers to grade the generated code snippets from five different models. The grade scales from zero to four, with zero meaning that the generated code is irrelevant and unhelpful, and four meaning that the generated code solves the problem accurately. Overall, there are 2860 annotated code snippets (5 generations $\times$ 472 examples) where each snippet is graded by 4.5 annotators on average.

**Functional Correctness Experiments** We evaluate functional correctness using the HumanEval (Chen et al., 2021) benchmark. Each example in HumanEval contains a natural language goal, hand-written input-output test cases, and a human-written reference solution. On average, each example has 7.7 test cases and there are 164 examples in total. While the original HumanEval is in Python, Cassano et al. (2022) translated HumanEval to 18 programming languages, and provided the predictions of `code-davinci-002` and their corresponding functional correctness. We used Java, C++, Python and JavaScript for these experiments, which are some of the most popular programming languages.[4] Notably, Cassano et al. (2022) did not translate the reference solutions to the other languages, so, we collected these from HumanEval-X (Zeng et al., 2022).The reference score of every example is either 1 ("correct", if it passes all test cases) or 0 ("incorrect", otherwise).

**Distinguishability** We also compare different metrics using the *distinguishability* meta-metric proposed in the CrystalBLEU paper (Eghbali & Pradel, 2022). We use their dataset, which was collected from the ShareCode[5] online coding platform, and contains solutions to programming problems. In each of Java and C++, we randomly selected 1000 pairs of examples that belong to the same problem (intra-class), and 1000 examples belonging to different problems (inter-class). We then computed distinguishability for BLEU, CodeBLEU, CrystalBLEU, and CodeBERTScore for these examples, following Eghbali & Pradel (2022).

**Correlation Metrics** Following best practices in natural language evalution, we used Kendall-Tau ($\tau$), Pearson ($r_p$) and Spearman ($r_s$) to measure the correlation between each metric's scores and the references. The detailed equations can be found in Appendix A.

**Hyperparameters** We tuned only the following hyperparameters for CodeBERTScore: whether to use $F_1$ or $F_3$, and which layer of the underlying model should we extract the encoded tokens from. We perform three-fold cross validation and report average results across the three folds. We eventually used $F_1$ in the human preference experiments and $F_3$ in the functional correctness

[3]https://huggingface.co/datasets/codeparrot/github-code-clean

[4]https://octoverse.github.com/2022/top-programming-languages

[5]https://sharecode.io

| Metric | $\tau$ | $r_p$ | $r_s$ |
|---|---|---|---|
| BLEU | .374 | .604 | .543 |
| CodeBLEU | .350 | .539 | .495 |
| ROUGE-1 | .397 | .604 | .570 |
| ROUGE-2 | .429 | .629 | .588 |
| ROUGE-L | .420 | .619 | .574 |
| METEOR | .366 | .581 | .540 |
| chrF | .470 | .635 | .623 |
| CrystalBLEU | .411 | .598 | .576 |
| CodeBERTScore | **.517** | **.674** | **.662** |

Table 1: Kendall-Tau ($\tau$), Pearson ($r_p$) and Spearman ($r_s$) correlations with human preference. The reported correlation is averaged across three runs, with standard deviation shown in Table 5 due to space limitations.

| Metric | Java | C++ |
|---|---|---|
| BLEU | 2.36 | 2.51 |
| CodeBLEU | 1.44 | 1.42 |
| CrystalBLEU | 5.96 | 6.94 |
| CodeBERTScore | **9.56** | **9.13** |

Table 2: Distinguishability with different metrics as the similarity function. CodeBERTScore achieves a higher distinguishability than CrystalBLEU, which proposed this meta-metric, on the same datasets.

| | Java | | C++ | | Python | | JavaScript | | Average | |
|---|---|---|---|---|---|---|---|---|---|---|
| Metric | $\tau$ | $r_s$ | $\tau$ | $r_s$ | $\tau$ | $r_s$ | $\tau$ | $r_s$ | $\tau$ | $r_s$ |
| BLEU | .481 | .361 | .112 | .301 | .393 | .352 | .248 | .343 | .308 | .339 |
| CodeBLEU | .496 | .324 | .175 | .201 | .366 | .326 | .261 | .299 | .325 | .287 |
| ROUGE-1 | .516 | .318 | .262 | .260 | .368 | .334 | .279 | .280 | .356 | .298 |
| ROUGE-2 | .525 | .315 | .270 | .273 | .365 | .322 | .261 | .292 | .355 | .301 |
| ROUGE-L | .508 | .344 | .258 | .288 | .338 | .350 | .271 | .293 | .344 | .319 |
| METEOR | **.558** | **.383** | .301 | .321 | .418 | .402 | **.324** | **.415** | .400 | .380 |
| chrF | .532 | .319 | .319 | .321 | .394 | .379 | .302 | .374 | .387 | .348 |
| CrystalBLEU | .471 | .273 | .046 | .095 | .391 | .309 | .118 | .059 | .257 | .184 |
| CodeBERTScore | **.553** | .369 | **.327** | **.393** | **.422** | **.415** | **.319** | .402 | **.405** | **.395** |

Table 3: The Kendall-Tau ($\tau$) and Spearman ($r_s$) correlations of each metric with the functional correctness on HumanEval in multiple languages. The correlation coefficients are reported as the average across three runs. Standard deviation is shown in Table 6 due to space limitations.

experiments. As for the layer to extract the token vectors from, we used layer 7 for CoNaLa, 7 for HumanEval-Java, 10 for HumanEval-C++, 11 for HumanEval-JS and 9 for HumanEval-Python.

## 4 RESULTS

**Human Preference** Table 1 shows the correlation between different metrics with human preference. CodeBERTScore achieves the highest correlation with human preference, across all correlation metrics. Evtikhiev et al. (2022) suggested that chrF and ROUGE-L are the most suitable for evaluating code generation models in CoNaLa. Nonetheless, CodeBERTScore outperforms these metrics by a significant margin. For example, CodeBERTScore achieves Kendall-Tau correlation of 0.517 compared to 0.470 of chrF and 0.420 of ROUGE-L. These results show that in general, generated code that is preferred by CodeBERTScore also tends to be preferred by human programmers.

**Functional Correctness** Table 3 shows the results for functional correctness: CodeBERTScore achieves the highest or comparable Kendall-Tau and Spearman correlation with functional correctness across *all four* languages. The METEOR baseline achieves a comparable correlation with CodeBERTScore in Java and JavaScript, and its correlation is surprisingly better than other baseline metrics. However in C++ and Python, CodeBERTScore is strictly better. Overall on average across languages, CodeBERTScore is more correlated with functional correctness than all baselines.

**Distinguishability** Table 2 shows that CodeBERTScore achieves a higher *distinguishability* than CrystalBLEU, which proposed this meta-metric, across both Java and C++. However, we also found

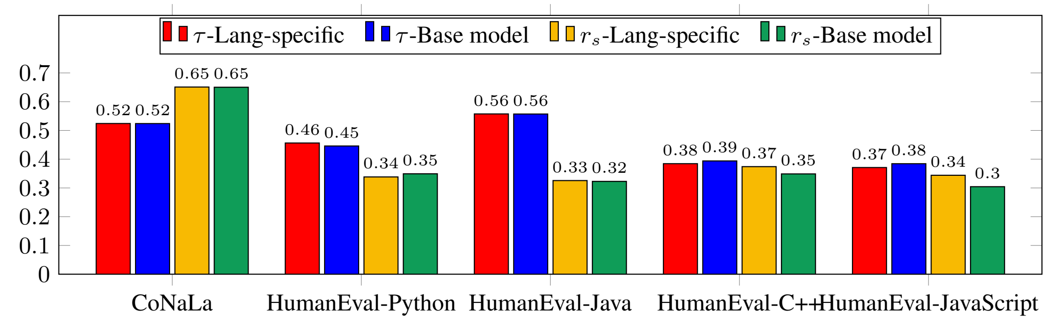



Figure 3: The Kendall-Tau and Spearman on the development set of different datasets with the language-specific pretrained model (Lang) and with the base CodeBERT (Base).

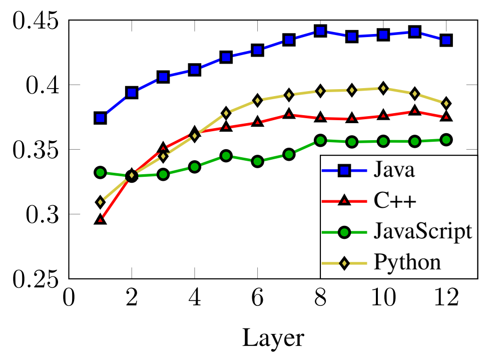



Figure 4: The average of Kendall-Tau and Spearman on the development set of HumanEval when using the embeddings from different layers.

Ref: `shutil.rmtree(folder)`

| Candidate | Ours | chrF |
|---|---|---|
| `os.rmdir(folder)` | 1 | 1 |
| `os.rmdir(f)` | 2 | 3 |
| `(folder)` | 3 | 2 |

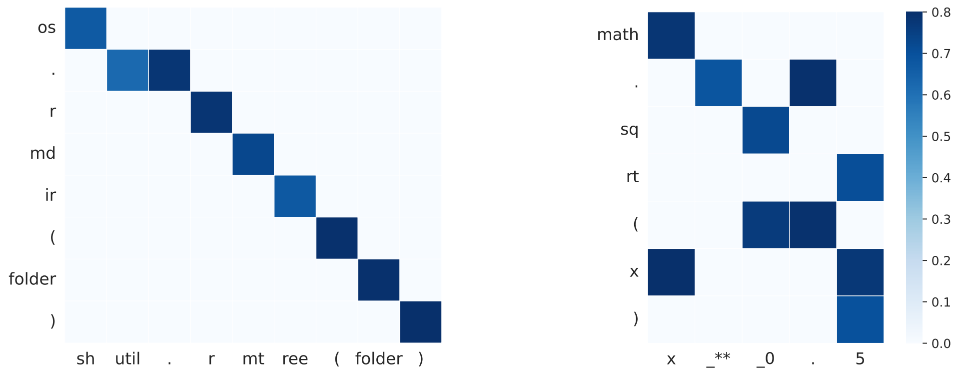
Figure 5: The similarity rankings of three code snippets given the reference code `shutil.rmtree(folder)`. While CodeBERTScore ("Ours") correctly ranks `os.rmdir(f)` over the the non-equivalent `(folder)`, chrF prefers just `(folder)` over `os.rmdir(f)`.

that distinguishability can be easily manipulated since it compares *absolute* scores between different metrics. For example, while CrystalBLEU achieves a distinguishability score of 5.96, we can craft a variant of CodeBERTScore that achieves a distinguishability score of 10,000 (more than 1600 times higher!) by simple exponentiation of CodeBERTScore's output score. We thus argue that distinguishability is not a reliable meta-metric and is no substitute for execution-based- or human-rating. Additional details and results are shown in Appendix B.

## 5 Ablation Study

We conducted a series of additional experiments, to understand the importance of different design decisions, and to gain insights on applying CodeBERTScore to new datasets and scenarios.

**Can we use CodeBERTScore in a new language without a language-specific CodeBERT?** In all experiments in Section 4, we used the language-specific model which we continued to pretrain on every language separately. But what if we wish to use CodeBERTScore in a language that we don't have a language-specific model for? We compared the language-specific models to the base CodeBERT model in Figure 3. Generally, the base CodeBERT achieves a close performance to a language-specific model. Surprisingly, even though C++ is not officially supported by CodeBERT due to its limited presence in the pretraining corpus of CodeBERT, its performance is on par with that of a model after continued pretraining on C++. However, in most HumanEval experiments and correlation metrics, using the language-specific model *is* beneficial. These results show that language-specific models are often preferred if such models are available, but the base CodeBERT can still provide close performance even without language-specific pretraining.




(a) (b)

Figure 6: Heatmaps of the similarity scores between two code pieces that achieve the same goal. Figure 6(a) shows the similarity scores between `os.remove(folder)` and `shutil.rmtree(folder)`. Figure 6(b) shows the similarity scores between `math.sqrt(x)` and `x ** 0.5`. In practice, we do not take all-punctuation tokens into the final score.

**Which transformer layer should we use?** We further investigate the impact of using embeddings from different layers of the model in CodeBERTScore. The results are shown in Figure 4: generally, the deeper the layer – the higher the average correlation between CodeBERTScore and functional correctness, across all programming languages. However sometimes, for example in Python, performance reaches a maximum at layer 10, and decreases afterwards. This suggests that higher layers encode the semantic information of each token more accurately, but the final layers may be more task-specific. These observations are consistent with (Tenney et al., 2019), who found that lower layers in BERT tend to process shallow information, while higher layers encode deeper semantic meaning in natural language.

**Does encoding natural language context help?** One major different between CodeBERTScore and BERTScore is that CodeBERTScore leverages the *context* for the generated code, such as the natural language (NL) intent that was given as input for generation. We find that NL context increases the correlation, for example, the Kendall-Tau of CodeBERTScore from 0.50 to 0.52. As programs often include rich context such as comments, these results suggest the potential of CodeBERTScore in encoding additional useful contextual information and further improve its accuracy.

## 6 ANALYSIS

**Soft matching of tokens** The heatmaps in Figure 6 show the similarity scores between tokens. For example, both `shutil.rmtree` and `os.rmdir` in Figure 6(a) delete a `folder`, and CodeBERTScore aligns them correctly, even though the two spans do not share many common tokens. In Figure 6(b), both code snippets calculate a square root, where one uses `math.sqrt(x)` and the other uses `x ** 0.5`. An exact surface-form-matching metric such as chrF would assign a low similarity score to this code pair, as they only share the token `x`. However, CodeBERTScore assigns non-zero scores to each token with meaningful alignments, such as matching `sqrt` with `_0.5`.

**Robustness to adversarial perturbations** We conducted a qualitative evaluation of CodeBERTScore under various perturbations. An example is shown in Figure 5, which shows the CodeBERTScore and chrF rankings of three code snippets based on the similarity to the reference `shutil.rmtree(folder)`. CodeBERTScore gives a higher ranking to the code snippet that employs the appropriate API (`os.rmdir`), compared to trivial `(folder)` that has the same variable name but without any function call. Contrarily, chrF assigns a higher ranking to `(folder)` which has a longer common sequence of characters, although semantically inequivalent.

## 7 Conclusion

In this paper, we present CodeBERTScore, a simple evaluation metric for code generation, which builds on BERTScore (Zhang et al., 2020), using pretrained language models of code, and leveraging the natural language context of the generated code. We perform an extensive evaluation across four programming languages, showing that CodeBERTScore is more correlated with human preference than all prior metrics. Further, we show that generated code that receives a higher score by CodeBERTScore is more likely to function correctly when executed. Finally, we release five programming language-specific pretrained models to use with our publicly available code. These models were downloaded more than 500,000 times from the HuggingFace Hub. Our code is available at https://github.com/neulab/code-bert-score .

## 8 Acknowledgements

We thank Misha Evtikhiev, Egor Bogomolov, and Timofey Bryksin for the discussions, and for the data from their paper (Evtikhiev et al., 2022) We are grateful to Yiwei Qin for the discussions regarding the T5Score paper (Qin et al., 2022); the idea to use functional correctness as a meta-metric was born thanks to the discussion with her. We are also grateful to Aryaz Eghbali and Michael Pradel for the discussions about CrystalBLEU (Eghbali & Pradel, 2022). Finally, we thank the DL4C workshop anonymous reviewers for their useful suggestions and feedback.

## References

Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th working conference on mining software repositories (MSR)*, pp. 207–216. IEEE, 2013.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. Self-supervised bug detection and repair. *Advances in Neural Information Processing Systems*, 34:27865–27876, 2021.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019a. URL https://openreview.net/forum?id=H1gKYo09tX.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019b.

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. Structural language models of code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pp. 245–256. PMLR, 2020. URL http://proceedings.mlr.press/v119/alon20a.html.

Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. Multi-lingual evaluation of code generation models. *ArXiv preprint*, abs/2210.14868, 2022. URL https://arxiv.org/abs/2210.14868.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *ArXiv preprint*, abs/2108.07732, 2021. URL https://arxiv.org/abs/2108.07732.

Satanjeev Banerjee and Alon Lavie. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pp. 65–72, Ann Arbor, Michigan, 2005. Association for Computational Linguistics. URL https://aclanthology.org/W05-0909.

Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin T. Vechev. Tfix: Learning to fix coding errors with a text-to-text transformer. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pp. 780–791. PMLR, 2021. URL http://proceedings.mlr.press/v139/berabi21a.html.

Shaked Brody, Uri Alon, and Eran Yahav. A structural model for contextual code changes. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. A scalable and extensible approach to benchmarking nl2code for 18 programming languages. *ArXiv preprint*, abs/2208.08227, 2022. URL https://arxiv.org/abs/2208.08227.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *ArXiv preprint*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

Aryaz Eghbali and Michael Pradel. Crystalbleu: precisely and efficiently measuring the similarity of code. In *37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–12, 2022.

Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. Out of the bleu: how should we assess quality of the code generation models? *ArXiv preprint*, abs/2208.03133, 2022. URL https://arxiv.org/abs/2208.03133.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *ArXiv preprint*, abs/2204.05999, 2022. URL https://arxiv.org/abs/2204.05999.

Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A. Smith. Don't stop pretraining: Adapt language models to domains and tasks. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 8342–8360, Online, 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.740. URL https://aclanthology.org/2020.acl-main.740.

Vincent J Hellendoorn and Premkumar Devanbu. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 763–773, 2017.

Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. *ArXiv preprint*, abs/2211.11501, 2022. URL https://arxiv.org/abs/2211.11501.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pp. 311–318, Philadelphia, Pennsylvania, USA, 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL https://aclanthology.org/P02-1040.

Yiwei Qin, Weizhe Yuan, Graham Neubig, and Pengfei Liu. T5score: Discriminative fine-tuning of generative evaluation metrics. *arXiv preprint arXiv:2212.05726*, 2022.

Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 419–428, 2014.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *ArXiv preprint*, abs/2009.10297, 2020. URL https://arxiv.org/abs/2009.10297.

Ian Tenney, Dipanjan Das, and Ellie Pavlick. BERT rediscovers the classical NLP pipeline. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 4593–4601, Florence, Italy, 2019. Association for Computational Linguistics. doi: 10.18653/v1/P19-1452. URL https://aclanthology.org/P19-1452.

Lewis Tunstall, Leandro von Werra, and Thomas Wolf. *Natural Language Processing with Transformers*. " O'Reilly Media, Inc.", 2022.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, Online and Punta Cana, Dominican Republic, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.685. URL https://aclanthology.org/2021.emnlp-main.685.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. Execution-based evaluation for open-domain code generation. *ArXiv preprint*, abs/2212.10481, 2022. URL https://arxiv.org/abs/2212.10481.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *ArXiv preprint*, abs/1910.03771, 2019. URL https://arxiv.org/abs/1910.03771.

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. A systematic evaluation of large language models of code, 2022.

Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 440–450, Vancouver, Canada, 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1041. URL https://aclanthology.org/P17-1041.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pp. 476–486, 2018a.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pp. 476–486. ACM, 2018b. doi: https://doi.org/10.1145/3196398.3196408.

Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, et al. Glm-130b: An open bilingual pre-trained model. *ArXiv preprint*, abs/2210.02414, 2022. URL https://arxiv.org/abs/2210.02414.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. Bertscore: Evaluating text generation with BERT. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=SkeHuCVFDr.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhengbao Jiang, and Graham Neubig. Docprompting: Generating code by retrieving the docs. In *International Conference on Learning Representations (ICLR)*, Kigali, Rwanda, May 2023. URL https://arxiv.org/abs/2207.05987.

# A CORRELATION METRICS

**Kendall-Tau ($\tau$)** $\tau$ measures the *ordinal/rank* association between a metric such as CodeBERTScore and the reference measurement. It is calculated as:

$$\tau = \frac{|\text{concordant}| - |\text{discordant}|}{|\text{concordant}| + |\text{discordant}|}$$

where |concordant| represents the number of pairs where two measurements agree on their relative rank. That is, if $f(\hat{y_1}, y_1^*) > f(\hat{y_2}, y_2^*)$, the reference measurement also yields $f^*(\hat{y_1}, y_1^*) > f^*(\hat{y_2}, c_2^*)$. Similarly, |discordant| represents the number of pairs where two measurements yield opposite ranks. Notably, in our experiments, we restrict the comparisons of ranks within the generations of the *same* question.

**Pearson ($r_p$)** $r_p$ measures the *linear* correlation between a metric and the reference measurement. It is defined as:

$$r_p = \frac{\sum_{i=1}^{N}(f(\hat{y_i}, y_i^*) - \bar{f})(f^*(\hat{y_i}, y_i^*) - \bar{f}^*)}{\sqrt{\sum_{i=1}^{N}(f(\hat{y_i}, y_i^*) - \bar{f})^2 \sum_{i=1}^{N}(f^*(\hat{y_i}, y_i^*) - \bar{f}^*)^2}}$$

where $N$ is the number of generations in the dataset, $\bar{f}$ is the mean CodeBERTScore of the dataset, and $\bar{f}^*$ is the mean similarity score calculated by the reference measurement.

**Spearman ($r_s$)** $r_s$ measures the Pearson correlation coefficient between the *ranks* produced by a metric and the reference measurement:

$$r_p = \frac{\text{cov}(R(f(\hat{\mathbf{Y}}), R(f^*(\mathbf{Y}^*)))}{\sigma_{R(f(\hat{\mathbf{Y}}))}\sigma_{R(f^*(\mathbf{Y}^*))}}$$

where $R$ returns the ranks of code snippets in a collection of code snippets $\mathbf{Y}$. $\text{cov}(\cdot, \cdot)$ is the covariance of two variables and $\sigma(\cdot)$ is the standard deviation.

# B DISTINGUISHING CODE WITH DIFFERENT SEMANTICS

We study how well can CodeBERTScore perform as a generic similarity function that measures the similarity between two arbitrary code snippets $y_i$ and $y_j$.

## B.1 DISTINGUISHABILITY METRIC

We evaluate CodeBERTScore using the distinguishability metric $d$ proposed by Eghbali & Pradel (2022) which is calculated as follows:

$$d = \frac{\sum_{y_i, y_j \in \text{Pairs}_{\text{intra}}} f(y_i, y_j)}{\sum_{y_i, y_j \in \text{Pairs}_{\text{inter}}} f(y_i, y_j)} \tag{1}$$

where $\text{Pair}_{\text{intra}}$ defines a set of code pairs from the same semantically equivalent clusters, and $\text{Pair}_{\text{inter}}$ defines a set of code pairs from two clusters of different functionality. Formally,

$$\begin{aligned}\text{Pair}_{\text{intra}} &= \{(y_i, y_j) \mid \exists k \text{ such that } y_i, y_j \in C_k\} \\ \text{Pair}_{\text{inter}} &= \{(y_i, y_j) \mid \exists k \text{ such that } y_i \in C_k, y_j \notin C_k\}\end{aligned}$$

where $C_k$ is the $k$-th cluster with semantically equivalent code snippets. Intuitively, a similarity function $f$ that can distinguish between similar and dissimilar code will produce $d$ larger than 1, meaning that a pair of code snippets from the same semantic cluster has a higher similarity score than a pair of snippets from different clusters. Since the number of intra-class and inter-class pairs grows quadratically with the number of code snippets, in our experiments we followed Eghbali & Pradel (2022) to sample $N$ inter- and $N$ intra-class pairs instead.

### B.2 Dataset with Semantically equivalent clusters

We follow Eghbali & Pradel (2022) to evaluate whether CodeBERTScore can distinguish similar and dissimilar code mined from ShareCode[6], an online coding competition platform. Semantically equivalent code snippets are from the same coding problem, and they all pass the unit tests provided by the platform. The dataset consists 6958 code snippets covering 278 problems in Java and C++. We use CodeBERTScore to calculate the similarity score for code pairs that share the same semantic class and code pairs that do not. We then measure the distinguishability of CodeBERTScore according to Equation B.1. The results are shown in Table 4.

| Metric | ShareCode Java | ShareCode C++ |
|---|---|---|
| BLEU | 2.36 | 2.51 |
| CodeBLEU | 1.44 | 1.42 |
| CrystalBLEU | 5.96 | 6.94 |
| CodeBERTScore | **9.56** | **9.13** |

Table 4: The distinguishability with different metrics as the similarity function. The best performance is **bold**.

We can see that CodeBERTScore achieves a higher distinguishability score compared to the CrystalBLEU baseline which proposed this meta-metric. This result confirms that CodeBERTScore assigns higher similarity scores to code pairs that are semantically similar, compared to two randomly paired snippets without semantic similarity.

**Can we hack the distinguishability metric?** The distinguishability metric as described in Equation B.1 is established through the calculation of the ratio between intra-class similarity and inter-class similarity, which makes it susceptible to potential manipulation. That is, if a constant transformation is applied to the output of a metric, it can make the results be as high as one wishes. To illustrate this, we conduct a distinguishability evaluation with the same configurations as before, but with a variant of CodeBERTScore that we call CodeBERTScore$^k$, and defined as the composition of CodeBERTScore with the $f(x) = x^k$ function, that is: $\text{CodeBERTScore}^k(y_1, y_2) = \text{CodeBERTScore}(y_1, y_2)^k$.

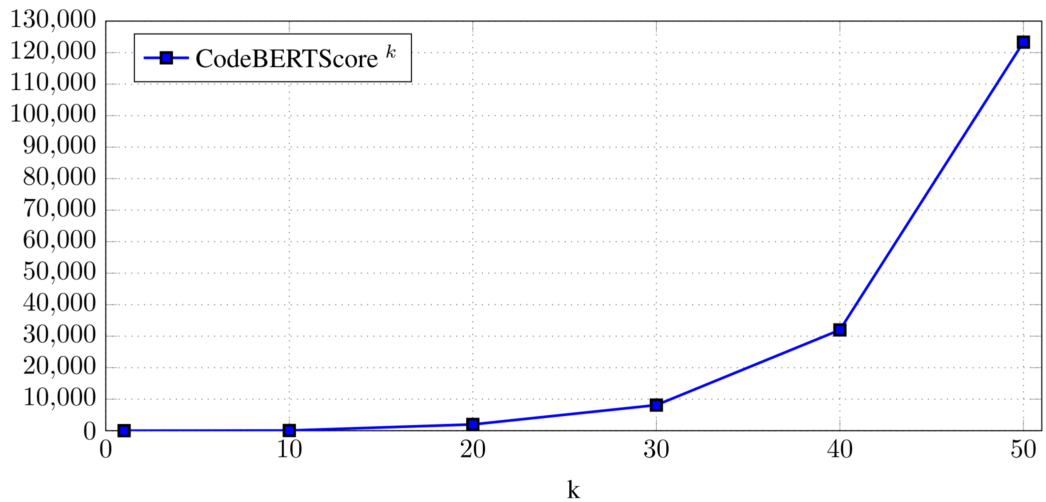



Figure 7: Distinguishability by exponentiating the original CodeBERTScore by $k$.

The distinguishability results of CodeBERTScore$^k$ with different values of $k$ are shown in Figure 7. As Figure 7 shows, the distinguishability increases almost exponentially with the increasing value

[6]https://sharecode.io/

| Metric | $\tau$ | $r_p$ | $r_s$ |
|---|---|---|---|
| BLEU | .374(±.025) | .604(±.016) | .543(±.018) |
| CodeBLEU | .350(±.037) | .539(±.033) | .495(±.037) |
| ROUGE-1 | .397(±.023) | .604(±.016) | .570(±.018) |
| ROUGE-2 | .429(±.025) | .629(±.015) | .588(±.022) |
| ROUGE-L | .420(±.037) | .619(±.014) | .574(±.022) |
| METEOR | .366(±.033) | .581(±.016) | .540(±.022) |
| chrF | .470(±.029) | .635(±.023) | .623(±.018) |
| CrystalBLEU | .411(±.030) | .598(±.019) | .576(±.034) |
| CodeBertScore | **.517**(±.024) | **.674**(±.012) | **.662**(±.012) |

Table 5: The Kendall-Tau ($\tau$), Pearson ($r_p$) and Spearman ($r_s$) correlation with human preference. The best performance is **bold**. The correlation coefficients are reported as the average across three runs. Numbers inside parentheses indicate the standard deviations.

| | Java | | C++ | | Python | | Javascript | |
|---|---|---|---|---|---|---|---|---|
| Metric | $\tau$ | $r_s$ | $\tau$ | $r_s$ | $\tau$ | $r_s$ | $\tau$ | $r_s$ |
| BLEU | .481(±.030) | .361(±.037) | .112(±.059) | .301(±.054) | .393(±.083) | .352(±.064) | .248(±.075) | .343(±.052) |
| CodeBLEU | .496(±.034) | .324(±.037) | .175(±.021) | .201(±.037) | .366(±.079) | .326(±.075) | .261(±.065) | .299(±.043) |
| ROUGE-1 | .516(±.052) | .318(±.043) | .262(±.073) | .260(±.024) | .368(±.092) | .334(±.054) | .279(±.092) | .280(±.068) |
| ROUGE-2 | .525(±.049) | .315(±.047) | .270(±.073) | .273(±.036) | .365(±.094) | .322(±.077) | .261(±.077) | .292(±.057) |
| ROUGE-L | .508(±.060) | .344(±.038) | .258(±.091) | .288(±.027) | .338(±.103) | .350(±.064) | .271(±.078) | .293(±.046) |
| METEOR | **.558**(±.058) | **.383**(±.027) | .301(±.061) | .321(±.023) | .418(±.090) | .402(±.049) | **.324**(±.075) | **.415**(±.022) |
| chrF | .532(±.067) | .319(±.035) | .319(±.056) | .321(±.020) | .394(±.096) | .379(±.058) | .302(±.073) | .374(±.044) |
| CrystalBLEU | .471(±.024) | .273(±.067) | .046(±.009) | .095(±.064) | .391(±.080) | .309(±.073) | .118(±.057) | .059(±.069) |
| CodeBERTScore | **.553**(±.068) | .369(±.049) | **.327**(±.086) | **.393**(±.048) | **.422**(±.090) | **.415**(±.071) | **.319**(±.054) | .402(±.030) |

Table 6: The Kendall-Tau ($\tau$) and Spearman ($r_s$) correlations of each metric with the functional correctness on HumanEval in multiple languages. The correlation coefficients are reported as the average across three runs, along with the standard deviation.

of $k$. We thus argue that distinguishability is not a reliable meta-metric and is no substitute for execution-based- or human-rating. We further suspect that any meta-metric that compares exact, absolute, scores across different metrics is susceptible to such manipulations, and the reliable way to compare metrics is according to the way they *rank* different examples, rather than the exact scores.

## C Full Results with Standard Deviations

We report the numbers with standard deviations in Table 5 and Table 6.