
CodeScore: Evaluating Code Generation by Learning Code Execution

Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li*, Zhuo Li, and Zhi Jin

Key Laboratory of High Confidence Software Technologies (Peking University),
Ministry of Education; School of Computer Science, Peking University, Beijing, China
 {dongyh, dingjz, jiangxue}@stu.pku.edu.cn, {lizhmq, lige, zhijin}@pku.edu.cn

Abstract

A proper code evaluation metric (CEM) profoundly impacts the evolution of code generation, which is an important research field in NLP and software engineering. Prevailing match-based CEMs (e.g., BLEU, Accuracy, and CodeBLEU) suffer from two significant drawbacks. 1. They primarily measure the surface differences between codes without considering their functional equivalence. However, functional equivalence is pivotal in evaluating the effectiveness of code generation, as different codes can perform identical operations. 2. They are predominantly designed for the Ref-only input format. However, code evaluation necessitates versatility in input formats. Aside from Ref-only, there are NL-only and Ref&NL formats, which existing match-based CEMs cannot effectively accommodate. In this paper, we propose CodeScore, a large language model (LLM)-based CEM, which estimates the functional correctness of generated code on three input types. To acquire CodeScore, we present UniCE, a unified code generation learning framework, for LLMs to learn code execution (i.e., learning PassRatio and Executability of generated code) with unified input. Extensive experimental results on multiple code evaluation datasets demonstrate that CodeScore absolutely improves up to 58.87% correlation with functional correctness compared to other CEMs, achieves state-of-the-art performance, and effectively handles three input formats.

1 Introduction

Automatic evaluation of code generation is significant and promising in the fields of natural language processing (NLP) and software engineering. Due to the great potential of code generation in reducing development costs and revolutionizing programming modes, both industry and academia have devoted substantial attention to it (Li et al., 2022; Mukherjee et al., 2021; Yin and Neubig, 2018; Chen et al., 2021; Shen et al., 2022; Dong et al., 2022). Code generation has achieved remarkable developments in the past few years (Fried et al., 2022; Nijkamp et al., 2022; Dong et al., 2023b; Jiang et al., 2023), but CEMs still need to catch up. It is challenging to evaluate the competitiveness of various approaches without proper CEM, which hampers the development of advanced techniques for code generation. A range of code generation subtasks would benefit from valid code evaluation, including code completion (Guo et al., 2022b; Lu et al., 2022), code translation (Rozière et al., 2020; Zhu et al., 2022), code search (Sun et al., 2022; Arakelyan et al., 2022), etc. Therefore, research on code evaluation is necessary and should be put on the agenda.

Some commonly used match-based CEMs treat code as text, such as BLEU (Papineni et al., 2002) and Accuracy, which focus on basic and lexical-level features. They compute scores mainly based on n-gram co-occurrence statistics. CodeBLEU (Ren et al., 2020) additionally takes into account the structure of code, i.e., abstract syntax tree and data flow. However, the preceding CEMs have

*Corresponding author

<pre><code>def bubbleSort(arr): n = len(arr) for i in range(n): for j in range(0, n-i-1): if arr[j] > arr[j+1]: arr[j], arr[j+1] = arr[j+1], arr[j]</code></pre>	Reference Code (a)	<ul style="list-style-type: none"> ▪ (a) $bubbleSort([5,3,2,1,4]) \rightarrow [1,2,3,4,5]$
<pre><code>def bubbleSort(arr): n = len(arr) for i in range(n): for j in range(0, n-i-1): if arr[j] == arr[j+1]: arr[j], arr[j+1] = arr[j+1], arr[j]</code></pre>	Generated Code (b)	<ul style="list-style-type: none"> ▪ (b) $bubbleSort([5,3,2,1,4]) \rightarrow \text{error}$ ▪ $BLEU(a, b) = 0.961$ ▪ $CodeBLEU(a, b) = 0.884$
<pre><code>def sortBubble (num_list): num_len = len(num_list) for j in range(num_len): sign = False for i in range(num_len - 1 - j): if a[i] > a[i+1]: a[i], a[i+1] = a[i+1], a[i] sign = True if not sign: break</code></pre>	Generated Code (c)	<ul style="list-style-type: none"> ▪ (c) $sortBubble([5,3,2,1,4]) \rightarrow [1,2,3,4,5]$ ▪ $BLEU(a, c) = 0.204$ ▪ $CodeBLEU(a, c) = 0.265$

Figure 1: Results of evaluating the generated code implementing bubble sort using different CEMs. BLEU and CodeBLEU score the truly functional correct code (c) lower than the incorrect code (b).

deficiencies in identifying code relationships, because code is mainly evaluated based on functional correctness rather than exact/fuzzy match to reference code, and match-based CEMs cannot account for the large and complex space of code functionally equivalent to reference code (Inala et al., 2022). For example, in Fig. 1, code (a) and code (b) have a much higher similarity of tokens or structures than code (c). But through execution, we realize that code (a) and code (c) are different renderings of the same function. By contrast, the execution result of code (b) differs dramatically from both other codes, and code (b) even fails to compile. As a result, merely measuring the similarity of token/structure is insufficient for code evaluation.

LLMs pre-trained on code have demonstrated outstanding results in code generation tasks (Chen et al., 2021; Fried et al., 2022; Li et al., 2022; Dong et al., 2023a), which are fundamentally dependent on exceptional code comprehension. Excellent code comprehension is a crucial element for facilitating code evaluation. We hypothesize that LLMs pre-trained on code possess the ability to evaluate code. However, due to the training strategy of predicting the next token according to context, they lack awareness of evaluating code for functional correctness. Our objective is to instruct LLMs to evaluate code effectively in terms of functional correctness.

Another issue that requires resolution is that the existing match-based CEMs are exclusively confined to the Ref-only input format. This restriction presents three inherent disadvantages. First, for any code generation task, the correct solutions are not finite, but rather, they are inexhaustible. In this context, the provided reference code merely represents one correct solution among a vast multitude. Therefore, it is overly narrow to compare the generated code solely with one correct solution. Second, they neglect the natural language (NL) description, which is a rich repository of information and a real requirement source. Third, these metrics are unusable in the absence of a reference code. This situation is quite commonplace in real-world evaluations where a correct solution is not always readily available. As a result, expanding the input format of CEM is necessary.

In this paper, we propose an effective LLM-based CEM, called CodeScore, which measures the functional correctness of generated codes on three input formats (Ref-only, NL-only, and Ref&NL). To obtain CodeScore, we present a code evaluation learning framework, UniCE, for tuning LLMs to estimate execution similarities with unified input. Specifically, we finetune LLMs to learn PassRatio and Executability of generated code, where Executability is devised to distinguish between compilation errors and output errors for code with PassRatio equal to 0. Generally, codes exhibiting higher functional correctness will pass more test cases, thereby achieving a higher PassRatio².

²Note that, although PassRatio varies across different test cases, it tends to yield a higher PassRatio for high-quality code, since we generate a large number of test cases. This phenomenon is somewhat akin to the process of human feedback. Despite the inherent variability in scores assigned by different human evaluators, the overarching trend remains consistent.

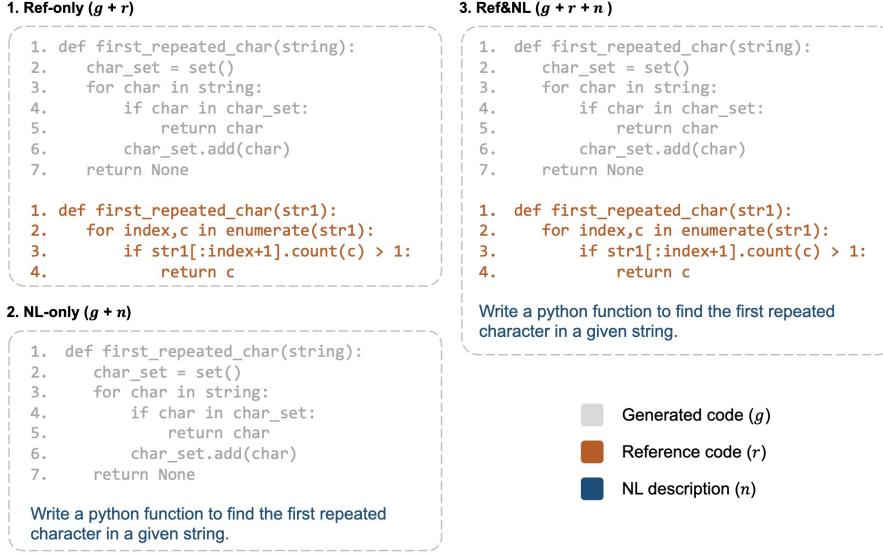


Figure 2: Examples of three input formats for code evaluation.

Consequently, for unexecutable codes, the model tends to assign scores approaching zero. In contrast, for codes demonstrating superior functional correctness, the model is likely to assign higher scores. CodeScore has the following advantages: 1) CodeScore has excellent evaluation performance, which achieves the state-of-the-art performance correlation with functional correctness on multiple code evaluation datasets. 2) CodeScore provides three application scenarios (Ref-only, NL-only, and Ref&NL) for code evaluation with unified input, while traditional CEMs only consider Ref-only. Our major contributions can be summarized as follows:

- We propose an efficient and effective LLM-based CEM, CodeScore, that accommodates the functional correctness of generated codes from execution viewpoint.³
- We present UniCE, a unified code evaluation learning framework based on LLMs with unified input, which assists models in learning code execution and predicting an estimate of execution PassRatio.⁴
- We construct three code evaluation datasets based on public benchmark datasets in code generation, called APPS-Eval, MBPP-Eval, and HE-Eval, respectively. Each task of them contains an NL description, several reference codes, 10+ generated codes, and 100+ test cases.⁵
- CodeScore substantially outperforms match-based CEMs and LLM-based EMs, and achieves the state-of-the-art performance on multiple code evaluation datasets.

2 Methodology

In this section, we first introduce our proposed CEM CodeScore, and then describe a unified code evaluation learning framework (i.e., UniCE), which is used to yield the CodeScore.

2.1 CodeScore

For a code generation task $p \in P$, let the test case set of p as $C_p = \{(\mathcal{I}_{p,c}, \mathcal{O}_{p,c})\}_{c \in C_p}$, a set of paired test case input $\mathcal{I}_{p,c}$ and test case output $\mathcal{O}_{p,c}$. Although the potential program space can be boundless, test cases permit automatic evaluation of code generation capability. Thus, in contrast to most other text generation tasks, human judgment is not always necessary for code generation.

³<https://huggingface.co/dz1/CodeScore>

⁴<https://github.com/Dingjz/CodeScore>

⁵<https://github.com/YihongDong/CodeGenEvaluation>

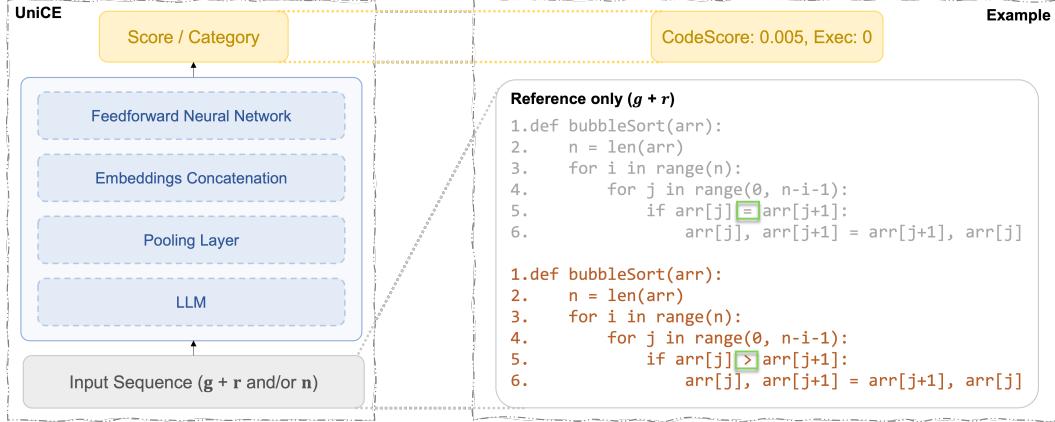


Figure 3: Diagram of UniCE, where the left side of the figure shows its model architecture, and the right side of the figure shows the example (case in Fig. 1) of input and output.

We measure the functional correctness with **PassRatio** $\in [0, 1]$, which is defined as

$$\text{PassRatio} = \frac{1}{|C_p|} \sum_{c \in C_p} \mathbb{I}\{\text{Eval}(\mathbf{g}_p, \mathcal{I}_{p,c}) = \mathcal{O}_{p,c}\}. \quad (1)$$

where $|\cdot|$ indicates the element number of a set, $\mathbb{I}(\cdot)$ is an indicator function, which outputs 1 if the condition is true and 0 otherwise, and $\text{Eval}(\mathbf{g}_p, \mathcal{I}_{p,c})$ represents an evaluation function that obtains outputs of code \mathbf{g}_p by way of executing it with $\mathcal{I}_{p,c}$ as input.

Our framework UniCE can learn existing CEMs, including PassRatio and Passability⁶. In this paper, we choose PassRatio since we want to study execution similarity and continuous PassRatio can better reflect the execution similarity of different codes than binary Passability. In the case of generated code with PassRatio equal to 0, we also use binary **Executability** to distinguish whether the generated code can be executed successfully with all given test cases, and thus measure its quality.

$$\text{Executability} = \begin{cases} 1, & \text{if code is executable,} \\ 0, & \text{otherwise.} \end{cases} \quad (2)$$

Given a unified input sequence \mathbf{x} that admits the following three types, as shown in Fig. 2:

1. **Ref-only** ($\mathbf{g} + \mathbf{r}$): Generated code concatenated with its reference code,
2. **NL-only** ($\mathbf{g} + \mathbf{n}$): Generated code concatenated with its NL description of requirements,
3. **Ref&NL** ($\mathbf{g} + \mathbf{r} + \mathbf{n}$): Generated code concatenated with both its reference code and NL.

UniCE yields a scalar **CodeScore** $\in [0, 1]$ and a binary number **Exec**:

$$\text{CodeScore}, \text{Exec} = \text{UniCE}(\mathbf{x}), \quad (3)$$

where $\text{Exec} = 1$ if \mathbf{g} can be executed successfully with all given test inputs otherwise 0, UniCE is our proposed learning framework, and details of UniCE are presented in Section 2.2.

We encourage UniCE to learn code execution (i.e., PassRatio and Executability) by minimizing loss function $\mathcal{L} = \mathcal{L}_C + \mathcal{L}_E$:

$$\mathcal{L} = \mathcal{L}_C + \mathcal{L}_E, \quad (4)$$

$$\mathcal{L}_C = (\text{CodeScore} - \text{PassRatio})^2, \quad (5)$$

$$\mathcal{L}_E = -\log \mathbf{p}(\text{Exec} | \text{Executability}), \quad (6)$$

where

$$\mathbf{p}(\text{Exec} | \text{Executability}) = \begin{cases} \mathbf{p}(\text{Exec}), & \text{if Executability} = 1, \\ 1 - \mathbf{p}(\text{Exec}), & \text{otherwise.} \end{cases} \quad (7)$$

⁶Passability is defined as $\frac{1}{|C_p|} \prod_{c \in C_p} \mathbb{I}\{\text{Eval}(\mathbf{g}_p, \mathcal{I}_{p,c}) = \mathcal{O}_{p,c}\}$.

2.2 UniCE

UniCE relies on LLMs to extract representations of \mathbf{x} and can work with existing pre-trained LLMs. A detailed illustration of the UniCE framework is presented in Fig. 3.

2.2.1 Pooling Layer

. For LLMs, the pooling layer plays a critical role in enhancing the model’s ability to capture and utilize information more effectively. The work (Tenney et al., 2019; Zhang et al., 2020; Rei et al., 2020) shows that exploiting information from different layers of LLM generally results in superior performance than only the last layer. Therefore, following the work (Peters et al., 2018), we pool information from different layers by using a layer-wise attention mechanism, and the final embedding of a token t can be computed as:

$$e_t = \gamma \sum_{k=1}^l e_t^k h^k, \quad (8)$$

where l indicates the number of layers, and γ and h^k are trainable weights.

2.2.2 Unified Embedding

. We require an efficient and comprehensive representation to encapsulate the unified input sequence x . Generally, there are two standard methods to extract the representation of x , i.e., averaging all token embeddings and using the first token embedding. While the first method is straightforward and includes information from all tokens, it may dilute the significance of more critical tokens and introduce extraneous noise. Moreover, the work (Ranasinghe et al., 2020; Wan et al., 2022) also proves the superiority of using the first token embedding compared to averaging all token embeddings in various applications. Thus, we employ the final embedding of first token e_{first} as the representation of unified input sequence x .

2.2.3 Unified training

. In UniCE, e_{first} is fed to a feedforward neural network to output a score and/or a category. To unify three evaluation input formats into UniCE, we apply multi-task learning for training. Specifically, for each step, we assign three sub-steps for three input formats, yielding \mathcal{L}^{Ref} , \mathcal{L}^{NL} , and \mathcal{L}^{Ref+NL} , respectively. A Ref&NL data can be regarded as three input format data to yield three losses, while Ref-only and NL-only data can only compute the corresponding \mathcal{L}^{Ref} and \mathcal{L}^{NL} . The final learning objective of UniCE is to minimize \mathcal{L}^{Uni} :

$$\mathcal{L}^{Uni} = \mathcal{L}^{Ref} + \mathcal{L}^{NL} + \mathcal{L}^{Ref+NL}, \quad (9)$$

where \mathcal{L}^{Ref} , \mathcal{L}^{NL} , and \mathcal{L}^{Ref+NL} are compute via Eq. 4 using corresponding format data as input.

3 Evaluation

We aim at answering the following research questions (RQs):

- RQ1: What is the performance of CodeScore on code evaluation tasks, compared to other EMs?
- RQ2: Can Exec effectively identify whether a generated code can be executed when all dependencies are met?
- RQ3: What is the contribution of L^{Uni} to UniCE for three input formats, compared to their respective losses?
- RQ4: How reasonable are the evaluations of CodeScore and other EMs from a human perspective?
- RQ5: How do CodeScore and other EMs perform on code evaluation tasks in a practical scenario?

3.1 Experiment Setup

In this section, we introduce datasets, baselines, correlation evaluation, and implementation details.

Table 1: Statistics of datasets (part 1).

Dataset	Examples Num			Avg Num / Task				Avg Length		
	Train	Dev	Test	NL	RefCode	GenCode	Extended (Original) TestCase	NL	RefCode	GenCode
APPS-Eval	267,162	33,395	33,395	1	13	42	181 (13)	263.8	86.3	76.8
MBPP-Eval	15,679	3,000	3,000	1	1	24	102 (3)	15.5	32.5	26.7
HE-Eval	-	-	1641	1	1	10	108 (8)	61.9	24.4	47.4

3.1.1 Datasets

We construct three public datasets (named APPS-Eval, MBPP-Eval, and HE-Eval) for code evaluation based on three public benchmark datasets in code generation, i.e., MBPP (Austin et al., 2021), APPS (Hendrycks et al., 2021), and HumanEval (Chen et al., 2021).

Table 2: Statistics of datasets (part 2).

Dataset	AvgPassRatio			Pass@1		
	Train	Dev	Test	Train	Dev	Test
MBPP-Eval	0.2832	0.2571	0.2890	0.0674	0.0494	0.0760
APPS-Eval	0.3196	0.1814	0.1790	0.0315	0.0007	0.0011
HE-Eval	-	-	0.3022	-	-	0.1499

To construct each code evaluation dataset, we first follow primitive NL and reference code in each corresponding base dataset. Then, for each paired NL and reference code in a code evaluation dataset, we generate an average of 20+ codes (generated from various LLMs, including CodeGen 350M&16B (Nijkamp et al., 2022), InCoder 1B&6B (Fried et al., 2022), and CodeX 13B&175B) (Chen et al., 2021) according to NL and additionally build an average of 100+ correct test cases according to reference code. To obtain these test cases, the following steps were implemented:

- 1) Infer the type of input from pre-existing test cases.
- 2) Enumerate a collection of inputs constrained by the type of input and task.
- 3) Feed the input into the original correct code and get the output by execution (We assume that all external dependencies including third-party libraries have been installed correctly).

Finally, we label each matched NL, reference code, and generated code by executing the generated code with all corresponding test cases to compute PassRatio via Eq. 1. Statistics of the datasets are presented in Table 1 and Table 2. As demonstrated in Table 1 and Table 2, there are notable disparities in the distributions of NL, RefCode (Reference Code), GenCode (Generated Code), and test cases across the three datasets.

3.1.2 Baselines

We select typical match-based CEMs, LLM-based EMs, and execution-based CEMs as baselines. We present each type of EMs as below.

Match-based CEMs include BLEU (Papineni et al., 2002), Exact Matching Accuracy (Accuracy), CodeBLEU (Ren et al., 2020), and CrystalBLEU (Eghbali and Pradel, 2022), specifically:

- **BLEU** (Papineni et al., 2002) is calculated based on n-gram, and the fluency and correctness of generated code are expressed by calculating the proportion of n consecutive tokens in the correct code, where n is usually set to 4 (i.e., BLEU-4). Considering that shorter codes usually have higher BLEU values, a penalty item is introduced to BLEU as:

$$\text{BLEU} = BP \cdot \exp \left(\sum_{m=1}^n \omega_m \log p_m \right),$$

$$BP = \begin{cases} 1, & l_g \geq l_r \\ e^{\left\{ 1 - \frac{r}{l_g} \right\}}, & l_g < l_r \end{cases},$$

where BP represents the penalty item, l_g represents the length of generated code, l_r represents the length of reference code, and ω_m and p_m represents the weighted coefficient and precision of m -gram, respectively.

- **Accuracy** indicates the percentage of exact matches between generated code and reference code.
- **CodeBLEU** (Ren et al., 2020) additionally takes into account the structure of code, which absorbs the advantages of BLEU in n-gram matching, and further injects code syntax through abstract syntax tree and code semantics through data flow.

$$\text{CodeBLEU} = \alpha \cdot \text{BLEU} + \beta \cdot \text{BLEU}_{\text{weight}} \\ + \delta \cdot \text{Match}_{\text{ast}} + \zeta \cdot \text{Match}_{\text{df}},$$

where α, β, δ and ζ are weights (usually set to 0.25, as well as in this paper), $\text{BLEU}_{\text{weight}}$ is a weighted BLEU with different weights for various tokens, $\text{Match}_{\text{ast}}$ is syntactic AST matching, which explores the syntactic information of the code, and Match_{df} is semantic dataflow matching, which considers the semantic similarity between generated code and reference code.

- **CrystalBLEU** (Eghbali and Pradel, 2022) is a metric that calculates BLEU by reducing the noise caused by trivially shared n-grams, such as ‘(’ and ‘;’.

LLM-based EMs contain two well-known and widely used text EMs (BERTScore (Zhang et al., 2020) and COMET (Rei et al., 2020)) and a concurrent work (CodeBERTScore (Zhou et al., 2023)), specifically:

- **BERTScore** (Zhang et al., 2020) is an automatic evaluation metric for text generation, which computes a similarity score for each token in the generated sentence with each token in the reference sentence with contextual embeddings of BERT (Devlin et al., 2019).

$$R_{\text{BERT}} = \frac{1}{|\mathbf{x}|} \sum_{\mathbf{x}_i \in \mathbf{x}} \max_{\hat{\mathbf{x}}_j \in \hat{\mathbf{x}}} \mathbf{x}_i^\top \hat{\mathbf{x}}_j, \quad P_{\text{BERT}} = \frac{1}{|\hat{\mathbf{x}}|} \sum_{\hat{\mathbf{x}}_j \in \hat{\mathbf{x}}} \max_{\mathbf{x}_i \in \mathbf{x}} \mathbf{x}_i^\top \hat{\mathbf{x}}_j, \\ F_{\text{BERT}} = 2 \frac{P_{\text{BERT}} \cdot R_{\text{BERT}}}{P_{\text{BERT}} + R_{\text{BERT}}}.$$

Following the setting in (Zhang et al., 2020), we compute BERTScore with inverse document frequency computed from test sets.

- **COMET** (Rei et al., 2020) provides a text EM by learning human judgments of training data, which leverages cross-lingual pre-trained language modeling to predict the quality of generated text more accurately.
- **CodeBERTScore** (Zhou et al., 2023) is a concurrent work that tries to use the same way as BERTScore with LLM pre-trained on code.

Execution-based CEMs consist of AvgPassRatio (Hendrycks et al., 2021).

- **AvgPassRatio** (Hendrycks et al., 2021) is defined as the average proportion of test cases that generated codes \mathbf{g}'_p s pass:

$$\text{AvgPassRatio} = \frac{1}{|P|} \sum_{p \in P} \frac{1}{|C_p|} \sum_{c \in C_p} \mathbb{I}\{\text{Eval}(\mathbf{g}_p, \mathcal{I}_{p,c}) = \mathcal{O}_{p,c}\}, \quad (10)$$

where $|\cdot|$ indicates the element number of a set, $\mathbb{I}(\cdot)$ is an indicator function, which outputs 1 if the condition is true and 0 otherwise, and $\text{Eval}(\mathbf{g}_p, \mathcal{I}_{p,c})$ represents an evaluation function that obtains outputs of code \mathbf{g}_p by way of executing it with $\mathcal{I}_{p,c}$ as input.

As mentioned above, continuous PassRatio (the item of AvgPassRatio) can better reflect the execution similarity of different codes than binary Passability (the item of Pass@1⁷). Therefore, in this paper, we mainly compare the correlation between CodeScore and AvgPassRatio in Execution-based CEMs.

The input format of the proceeding baselines is Ref-only and each of them except COMET is in the range of 0 to 1.

⁷Pass@1 (Kulal et al., 2019) is defined as the percentage of \mathbf{g}'_p s that pass all test cases of the corresponding p : $\frac{1}{|P|} \sum_{p \in P} \frac{1}{|C_p|} \prod_{c \in C_p} \mathbb{I}\{\text{Eval}(\mathbf{g}_p, \mathcal{I}_{p,c}) = \mathcal{O}_{p,c}\}$, where Pass@1 is a more stringent CEM, also known as Strict Accuracy.

3.1.3 Correlation Evaluation

We use three major correlation coefficients in statistics (i.e., Kendall-Tau(τ), Spearman R (r_s), and Pearson R (r_p) to evaluate the correlation between each EM and functional correctness. Furthermore, we use Mean Absolute Error (MAE) to assess the absolute error between them.

Kendall-Tau (τ) (Kendall, 1938) is a statistic used to measure the ordinal association between two measured data:

$$\tau = \frac{\text{Concordant} - \text{Discordant}}{\text{Concordant} + \text{Discordant}}, \quad (11)$$

where *Concordant* indicates the number of occurrences that two evaluation data M^1 and M^2 exist either both $M_i^1 > M_j^1$ and $M_i^2 > M_j^2$ or both $M_i^1 < M_j^1$ and $M_i^2 < M_j^2$, and *Discordant* indicates the number of occurrences opposite to *Concordant*.

- **Spearman R (r_s)** (Mood, 1950) is a nonparametric measure of rank correlation (statistical dependence between the rankings of two data):

$$r_s = \frac{\text{cov}(\text{R}(M^1), \text{R}(M^2))}{\sigma_{\text{R}(M^1)} \sigma_{\text{R}(M^2)}}, \quad (12)$$

where $\text{R}(M^1)$ and $\text{R}(M^2)$ represent the rankings of M^1 and M^2 , $\text{cov}(\cdot, \cdot)$ means the covariance function, and σ_M means the standard deviation of M .

- **Pearson R (r_p)** (Bravais, 1844) is a measure of linear correlation between two data:

$$r_p = \frac{\text{cov}(M^1, M^2)}{\sigma_{M^1} \sigma_{M^2}}. \quad (13)$$

- **Mean Absolute Error (MAE)** is a measure of errors between paired data:

$$\text{MAE} = \frac{\sum_{i=1}^N |M_i^1 - M_i^2|}{N}, \quad (14)$$

where $|\cdot|$ means the absolute-value function.

3.1.4 Implementation Details

In this paper, UniXcoder (Guo et al., 2022a) is employed as the base LLM of UniCE, which has the similar parameter size of LLMs in BERTScore (Zhang et al., 2020) and COMET (Rei et al., 2020), and larger LLMs can usually lead to better results. We train UniCE with Adam (Kingma and Ba, 2015) optimizer on a single GPU of Tesla A100-PCIe-40G. Empirically, the learning rate is set to 0.001. The feedforward neural network of UniCE consists of 3 linear transitions with the hyperbolic tangent (Tanh) activation functions, where the corresponding output dimensions are 3,072, 1,024, and 2, respectively. The input token length is limited to 1024. To mitigate the instability of model training, we exhibit the average performance of UniCE running five times.

3.2 Experimental Results

In this section, we conduct extensive experiments to verify the effectiveness and generalization of CodeScore.

3.2.1 RQ1: Effect of CodeScore

As illustrated in Table 3, CodeScore exhibits a significantly stronger correlation with functional correctness than existing match-based CEMs and LLM-based EMs, which display weak or extremely weak correlations with Ground Truth on APPS-Eval. Compared with the top-performing EM among other EMs, CodeScore achieved absolute improvements of 40.56%, 55.07%, and 58.87% on τ , r_s , and r_p , respectively. With an r_s value greater than 0.6, it is evident that there is a strong correlation between CodeScore and Ground Truth. Furthermore, CodeScore has the lowest MAE compared to other EMs. The execution time of CodeScore is similar to other LLM-based EMs and slightly longer than existing Match-based CEMs. However, compared to the $20.7k\times$, $28.7k\times$, and $22.1k\times$ execution time of execution-based CEMs in three code evaluation datasets, CodeScore reduces

Table 3: Correlation comparison of functional correctness on APPS-Eval dataset.

Method	Value	$\tau \uparrow$	$r_s \uparrow$	$r_p \uparrow$	MAE \downarrow	Execution Time \downarrow
Match-based CEM						
BLEU (Papineni et al., 2002)	0.0094	0.1055	0.1156	0.0959	0.1164	1.0 \times (26.0s)
Accuracy	0.0001	0.0079	0.0095	0.0196	-	0.1 \times
CodeBLEU (Ren et al., 2020)	0.2337	0.1035	0.1533	0.1085	0.2005	7.8 \times
CrystalBLEU (Eghbali and Pradel, 2022)	0.0242	0.0906	0.1347	0.0887	0.1709	0.3 \times
LLM-based EM						
BERTScore (Zhang et al., 2020)	0.8629	0.0916	0.1375	0.0718	0.6874	56.7 \times
COMET (Rei et al., 2020)	0.0165	0.0904	0.1126	0.1187	0.1751	84.0 \times
CodeBERTScore (Zhou et al., 2023)	0.7583	0.1219	0.1801	0.1323	0.5885	27.8 \times
CodeScore						
Ref-only ($g + r$)						
UniCE with \mathcal{L}^{Ref}	0.1996	0.4760	0.6473	0.6620	0.1202	
UniCE with \mathcal{L}^{Uni}	0.1977	0.5033	0.6693	0.6929	0.1128	33.7 \times
NL-only ($g + n$)						
UniCE with \mathcal{L}^{NL}	0.2035	0.4679	0.6359	0.6855	0.1189	
UniCE with \mathcal{L}^{Uni}	0.2016	0.4901	0.6486	0.6905	0.1120	37.9 \times
Ref&NL ($g + r + n$)						
UniCE with \mathcal{L}^{Ref+NL}	0.1837	0.3865	0.5419	0.6152	0.1274	
UniCE with \mathcal{L}^{Uni}	0.1820	0.5275 ($\uparrow 40.56\%$)	0.7040 ($\uparrow 55.07\%$)	0.7210 ($\uparrow 58.87\%$)	0.1044	44.2 \times
Execution-based CEM						
13 test cases per task	0.0978	0.3360	0.4108	0.4987	0.1327	1.5k \times
181 test cases per task	0.1790	-	-	-	-	20.7k \times

Table 4: Correlation comparison of functional correctness on MBPP-Eval and HE-Eval datasets.

Method	Value	MBPP-Eval		HE-Eval		Execution Time \downarrow
		$r_s \uparrow$	Execution Time \downarrow	Value	$r_s \uparrow$	
Match-based CEM						
BLEU (Papineni et al., 2002)	0.1186	0.1784	1.0 \times (0.87s)	0.2249	0.0678	1.0 \times (0.78s)
Accuracy	0.0004	0.0299	0.1 \times	0.0006	0.0367	0.1 \times
CodeBLEU (Feng et al., 2020)	0.1827	0.2902	5.0 \times	0.3826	0.4084	6.4 \times
CrystalBLEU (Eghbali and Pradel, 2022)	0.0295	0.1645	0.3 \times	0.0158	0.2013	0.4 \times
LLM-based EM						
BERTScore (Zhang et al., 2020)	0.8842	0.1522	62.0 \times	0.8862	0.0069	57.7 \times
COMET (Rei et al., 2020)	-0.5001	0.2681	69.0 \times	0.0642	0.0716	58.6 \times
CodeBERTScore (Zhou et al., 2023)	0.7863	0.2490	44.9 \times	0.7917	0.2604	47.5 \times
CodeScore						
Ref-only ($g + r$)						
UniCE with \mathcal{L}^{Ref}	0.2975	0.5864		0.3115	0.5250	
UniCE with \mathcal{L}^{Uni}	0.3253	0.5999	17.2 \times	0.4055	0.6009	30.8 \times
NL-only ($g + n$)						
UniCE with \mathcal{L}^{NL}	0.3364	0.4492		0.4748	0.5217	
UniCE with \mathcal{L}^{Uni}	0.3327	0.5719	12.6 \times	0.5357	0.5755	31.0 \times
Ref&NL ($g + r + n$)						
UniCE with \mathcal{L}^{Ref+NL}	0.2905	0.5926		0.3866	0.5153	
UniCE with \mathcal{L}^{Uni}	0.3247	0.6054 ($\uparrow 31.52\%$)	20.7 \times	0.4505	0.6048 ($\uparrow 19.64\%$)	33.3 \times
Execution-based CEM						
8 test cases per task	0.2670	0.6826	1.0k \times	0.5652	0.7426	1.9k \times
108 test cases per task	0.2890	-	28.7k \times	0.3022	-	22.1k \times

execution time by three orders of magnitude. We also find that computing execution-based CEMs for code evaluation with a small number of original test cases is insufficient. They have a significant reduction in correlation coefficients compared to using larger extended test cases. In cases where test cases are rare or low-quality, such as on APPS-Eval, the correlation between our CodeScore and Ground Truth even far exceeds that of execution-based CEMs.

We also sought to determine the generality of CodeScore. In Table 4, we utilize CodeScore, trained on APPS-Eval, to evaluate the code in MBPP-Eval and HE-Eval with fine-tuning and zero-shot settings, respectively. It is important to note that the distributions of NL, RefCode, GenCode, and test cases across these three datasets are quite different, as evidenced by their respective statistics shown in Table 1 and Table 2. Table 4 reveals the effectiveness of CodeScore on MBPP-Eval and HE-Eval. Remarkably, CodeScore continues to achieve the best correlation compared to other match-based CEMs and LLM-based EMs in these two settings.

Another intriguing finding is that the quality of CodeBLEU inversely correlates with code length. In other words, the longer code, the poorer correlation between CodeBLEU and Ground Truth. This is

likely due to the fact that longer codes tend to incorporate more variations in their syntactic structure. Therefore, for longer codes, the evaluation effect of CodeBLEU gradually degrades to BLEU.

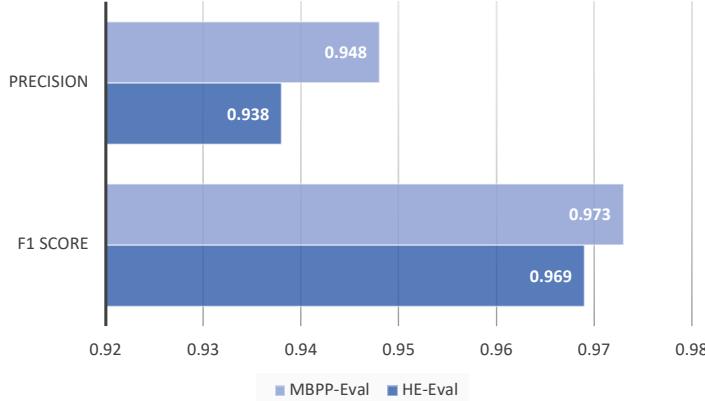


Figure 4: The performance of Exec on MBPP-Eval and HE-Eval datasets.

3.2.2 RQ2: Effect of Exec

We also evaluate the performance of Exec on MBPP-Eval and HE-Eval datasets, as shown in Fig. 4. The experimental results indicate that Exec demonstrates remarkably high performance in terms of Precision and F1 Score. These findings suggest that Exec possesses a robust capability in identifying whether the code can be executed (when all dependencies are met), and the results also prove that using UniCE to learn code execution is effective for code evaluation.

3.2.3 RQ3: Effect of \mathcal{L}^{Uni}

As observed from Tables 3 and 4, our proposed \mathcal{L}^{Uni} demonstrates enhancements across all input formats when compared to their respective losses on APPS-Eval, MBPP-Eval, and HE-Eval datasets. With changes in the input format, both the correlation coefficients and MAE between CodeScore and Ground Truth also vary. Generally, the Ref&NL input format yields superior results, which shows that accommodating NL has a positive effect on evaluating the generated code, while the traditional Ref-only input format omits the valuable information in NL. Additionally, according to the Avg Length data presented in Table 1, we discovered that the execution time of CodeScore exhibits a linear, positive relationship with the input length. Regardless of the input formats, our proposed CodeScore provides a commendable evaluation of generated code. This is attributable to the fact that \mathcal{L}^{Uni} aids in training a code evaluation model with a unified input.

Table 5: Human evaluation for functional correctness.

EM	Reasonableness of Evaluation
BERTScore (Zhang et al., 2020)	1.3 ± 0.4
CodeBLEU (Ren et al., 2020)	2.1 ± 0.5
CodeBERTScore (Zhou et al., 2023)	2.2 ± 0.7
CodeScore	3.4 ($\uparrow 54.6\%$) ± 0.3
Ground Truth	4.6 ± 0.2

3.2.4 RQ4: Human Evaluation

In this section, we conduct a human evaluation to gauge the validity of our CodeScore. Considering the costliness of human evaluation, we select only five representative EMs for this task, namely, CodeScore, CodeBLEU, BERTScore, CodeBERTScore, and Ground Truth (i.e., PassRatio). All of these EMs are continuous and range from 0 to 1. In accordance with previous work (Hao et al., 2022)

and our experimental setup, we manually assess the validity of each EM in gauging the functional correctness of the generated code. The score for this evaluation is an integer ranging from 0 to 5, where 0 denotes poor and 5 signifies excellent performance.

We randomly select 100 generated codes and evaluations scored by the five EMs mentioned above on these samples. Finally, we obtain 500 (100*5) paired generated code with reference code and NL and its EM score for human evaluation. The evaluators are computer science Ph.D. students and are not co-authors. They all have programming experience ranging from 3+ years. The 500 code snippets are divided into 10 groups, with each questionnaire containing one group. We randomly list the generated code with reference code and NL and the corresponding EM score on the questionnaire. Each group is evaluated anonymously by one evaluator, and the final score is the average of all evaluators' scores. Evaluators are allowed to search the Internet for unfamiliar concepts.

We present the results of the human evaluation in Table 5. Remarkably, our proposed CodeScore significantly outperforms all other EMs. Relative to these, CodeScore shows an improvement of at least 54.6% in the human evaluation. All p-values are substantially less than 0.005, underscoring that these improvements are statistically significant.

NL	Write a function to find numbers divisible by m and n from a list of numbers using lambda function.		
Reference Code	<pre>def div_of_nums(nums,m,n): result = list(filter(lambda x: (x % m == 0 and x % n == 0), nums)) return result</pre>		
	Generated Code I.1 ✘	Generated Code I.2 ✓	
	<pre>def div_of_nums(nums,m,n): ans = [] for n in nums: if n%m == 0: ans.append(n) return ans</pre>	CodeScore: 0.4159 CodeBLEU: 0.2283 BERTScore: 0.8862 CodeBERTScore: 0.8615	CodeScore: 0.9971 CodeBLEU: 0.2456 BERTScore: 0.9005 CodeBERTScore: 0.8484

(a) Case I

NL	Write a python function to check whether the given two numbers have same number of digits or not.		
Reference Code	<pre>def same_Length(A,B): while (A > 0 and B > 0): A = A / 10; B = B / 10; if (A == 0 and B == 0): return True; return False;</pre>		
	Generated Code II.1 ✘	Generated Code II.2 ✘	
	<pre>def same_Length(iterable, n): p = (n-1) *iterable return p</pre>	CodeScore: 0.1272 CodeBLEU: 0.1383 BERTScore: 0.8542 CodeBERTScore: 0.7021	CodeScore: 0.5712 CodeBLEU: 0.1744 BERTScore: 0.8454 CodeBERTScore: 0.7816

(b) Case II

NL	Write a function to return true if the given number is even else return false.		
Reference Code	<pre>def even_num(x): if x%==0: return True else: return False</pre>		
	Generated Code III.1 ✘	Generated Code III.2 ✓	
	<pre>def even_num(n,count = 0): if (n == 0): return False if (count % 2 == 101): return (n % 2 == 3) else: return (2 * n - 1) + (n % 4 - 1) > 1</pre>	CodeScore: 0.3590 CodeBLEU: 0.3582 BERTScore: 0.8882 CodeBERTScore: 0.7764	CodeScore: 0.9904 CodeBLEU: 0.3421 BERTScore: 0.8665 CodeBERTScore: 0.8345

(c) Case III

Figure 5: Case study. For each case, the second generated code is superior to the first one.

3.2.5 RQ5: Case Study

Fig. 5 displays a selection of generated codes and their corresponding EM scores (as per Section 3.2.4) on MBPP-Eval dataset. It becomes evident that CodeBLEU, BERTScore, and CodeBERTScore each exhibit unique issues. From these examples, we glean the following insights: 1) CodeBLEU tends to assign relatively low scores to generated code, even when the code is functionally correct. Furthermore, it appears to favor generated codes that maintain structural consistency with the reference code. For instance, even though Generated Code III.2 is functionally correct, it receives a lower CodeBLEU score than III.1, which is fundamentally incorrect. 2) Both BERTScore and CodeBERTScore have a propensity to award relatively high scores to generated code, even when the code is essentially flawed. Additionally, they often assign lower scores to better generated codes. For example, Generated Code II/III.2 has a lower BERTScore than II/III.1, and Generated Code I.2 has a lower CodeBERTScore than I.1. In contrast, CodeScore performs admirably in all of these scenarios. In summary, our proposed CodeScore aligns more closely with Ground Truth compared to other EMs. This suggests that CodeScore is more effective in estimating the functional correctness of generated code.

4 Discussion

While we have demonstrated that CodeScore is an effective LLM-based metric for code evaluation, we acknowledge that it still has certain limitations.

- First, learning code execution for code evaluation requires collecting a certain amount of data, including sufficient test cases, generated codes, reference codes, and NL descriptions. However, collecting this data is far less expensive than performing human evaluation.
- Second, in this paper, CodeScore is more suitable for evaluating function-level code in Python. Nevertheless, our work establishes the viability of code evaluation based on UniCE, and this approach can feasibly be extended to other scenarios. We aim to broaden CodeScore to encompass a wider range of codes in our future work.
- Third, employing CodeScore for code evaluation entails additional computation and time. However, we maintain that this is still within an acceptable range, considering the benefits it provides in terms of the accuracy and reliability of code evaluation.

5 Related Work

In this section, we primarily discuss three types of EMs, including Match-based CEMs, Execution-based CEMs, and LLM-based EMs.

Match-based CEMs. Besides these commonly used BLEU (Papineni et al., 2002), Accuracy, and CodeBLEU (Ren et al., 2020), some niche CEMs (Popovic, 2015) are also applied to code evaluation, e.g., METEOR (Banerjee and Lavie, 2005), ROUGE (Lin, 2004), and CrystalBLEU (Eghbali and Pradel, 2022). However, these aforementioned match-based CEMs merely measure the surface-level differences in code and do not take into account the functional correctness of the generated code.

Execution-based CEMs. They attempt to handle these issues by running tests for generated code to verify its functional correctness (Kulal et al., 2019; Hendrycks et al., 2021; Hao et al., 2022). However, they come with several caveats: 1) It assumes that test cases have been given and all dependencies have been resolved. For each code generation task, supplying adequate test cases is a burden in practice, and the dependencies required vary from task to task. 2) Enormous computational overhead needs to be afforded. All generated code requires execution separately for each corresponding test case, which leads to enormous CPU and I/O overhead. 3) Execution with isolation mechanisms. The generated code could have some security risks, such as deleting files on the disk or implanting computer viruses, especially if the training data of code generation models is attacked. In a word, they are usually costly, slow, and insecure, which are often unavailable or ineffective in real-world scenarios.

LLM-based EMs. Effective evaluation of generated results is hard for both text and code generation. They likewise face the same issue of poor evaluation metrics (EMs). A recent popular trend in evaluating text generation is the design of automatic EMs based on LLMs. A part of LLM-based

EMs (Rei et al., 2021; Wan et al., 2022; Rei et al., 2022) follows COMET (Rei et al., 2020) to learn high-quality human judgments of training data, which is a problem for code evaluation to obtain. Another part relies on LLM extracting token embeddings to calculate scores like BERTScore (Zhang et al., 2020), such as (Zhao et al., 2019; Sellam et al., 2020; Yuan et al., 2021; Reimers and Gurevych, 2019). A concurrent work named CodeBERTScore (Zhou et al., 2023) tries to use the same way as BERTScore with LLM pre-trained on code. However, they do not teach LLMs to learn code evaluation effectively, in other words, LLMs are still confused about how to evaluate code. Therefore, they exhibit suboptimal performance in code evaluation, as evidenced by our experimental results.

6 Conclusion and Future Work

In this paper, we have proposed a code evaluation learning framework based on LLMs with a unified input, which we refer to as UniCE. UniCe is designed to learn the code execution of generated code. In response to the imprecise evaluations provided by existing match-based CEMs and LLM-based EMs, we introduced CodeScore based on UniCE, which is an effective CEM to measure the functional correctness of generated code. Furthermore, our CodeScore can be applied to three application scenarios (Ref-only, NL-only, and Ref&NL) for code evaluation with a unified input. This is in contrast to traditional CEMs, which typically only consider the Ref-only scenario. To validate CodeScore, we constructed three code evaluation datasets (i.e., APPS-Eval, MBPP-Eval, and HE-Eval), which correspond to three popular benchmark datasets in code generation (i.e., MBPP, APPS, and HumanEval). Experimental results affirm the efficacy of CodeScore, which achieves state-of-the-art performance on multiple code evaluation datasets.

We hope this work sheds light on future work in the direction of LLM-based code evaluation. Our code evaluation dataset can serve as a benchmark for evaluating the functional correctness of generated code. Furthermore, our work can be applied to facilitate the training of code generation models by providing positive feedback.

References

- Shushan Arakelyan, Anna Hakhverdyan, Miltiadis Allamanis, Christophe Hauser, Luis Garcia, and Xiang Ren. 2022. NS3: Neuro-Symbolic Semantic Code Search. *CoRR* abs/2205.10674 (2022).
- Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *CoRR* abs/2108.07732 (2021).
- Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In *IEEvaluation@ACL*. Association for Computational Linguistics, 65–72.
- Auguste Bravais. 1844. *Analyse mathématique sur les probabilités des erreurs de situation d'un point*. Impr. Royale.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebbgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021).
- Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT (1)*. Association for Computational Linguistics, 4171–4186.
- Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023a. Self-collaboration Code Generation via ChatGPT. *CoRR* abs/2304.07590 (2023).
- Yihong Dong, Ge Li, and Zhi Jin. 2022. Antecedent Predictions Are Dominant for Tree-Based Code Generation. *CoRR* abs/2208.09998 (2022).
- Yihong Dong, Ge Li, and Zhi Jin. 2023b. CODEP: Grammatical Seq2Seq Model for General-Purpose Code Generation. In *ISSTA*.
- Aryaz Eghbali and Michael Pradel. 2022. CrystalBLEU: Precisely and Efficiently Measuring the Similarity of Code. In *ASE*. ACM, 28:1–28:12.
- Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *EMNLP (Findings) (Findings of ACL, Vol. EMNLP 2020)*. Association for Computational Linguistics, 1536–1547.
- Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999* (2022).
- Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022a. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *ACL (1)*. Association for Computational Linguistics, 7212–7225.
- Daya Guo, Alexey Svyatkovskiy, Jian Yin, Nan Duan, Marc Brockschmidt, and Miltiadis Allamanis. 2022b. Learning to Complete Code with Sketches. In *ICLR*. OpenReview.net.
- Yiyang Hao, Ge Li, Yongqiang Liu, Xiaowei Miao, He Zong, Siyuan Jiang, Yang Liu, and He Wei. 2022. AixBench: A Code Generation Benchmark Dataset. *CoRR* abs/2206.13179 (2022).

- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. In *NeurIPS Datasets and Benchmarks*.
- Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andrés Codas, Mark Encarnación, Shuvendu K. Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-Aware Neural Code Rankers. *CoRR* abs/2206.03865 (2022).
- Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. 2023. Self-planning Code Generation with Large Language Model. *CoRR* abs/2303.06689 (2023).
- Maurice G Kendall. 1938. A new measure of rank correlation. *Biometrika* 30, 1/2 (1938), 81–93.
- Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR*.
- Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy Liang. 2019. SPoC: Search-based Pseudocode to Code. In *NeurIPS*. 11883–11894.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittweiser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.
- Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out*. Association for Computational Linguistics, 74–81.
- Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. In *ACL*. Association for Computational Linguistics, 6227–6240.
- Alexander McFarlane Mood. 1950. Introduction to the Theory of Statistics. (1950).
- Rohan Mukherjee, Yeming Wen, Dipak Chaudhari, Thomas W. Reps, Swarat Chaudhuri, and Christopher M. Jermaine. 2021. Neural Program Generation Modulo Static Analysis. In *NeurIPS*. 18984–18996.
- Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. *arXiv preprint arXiv:2203.13474* (2022).
-]ChatGPT OpenAI. [n. d.]. *ChatGPT: Optimizing Language Models for Dialogue*. <https://openai.com/blog/chatgpt/>
- Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *ACL*. ACL, 311–318.
- Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep Contextualized Word Representations. In *NAACL-HLT*. Association for Computational Linguistics, 2227–2237.
- Maja Popovic. 2015. chrF: character n-gram F-score for automatic MT evaluation. In *WMT@EMNLP*. The Association for Computer Linguistics, 392–395.
- Tharindu Ranasinghe, Constantin Orasan, and Ruslan Mitkov. 2020. TransQuest: Translation Quality Estimation with Cross-lingual Transformers. In *COLING*. International Committee on Computational Linguistics, 5070–5081.
- Ricardo Rei, José GC De Souza, Duarte Alves, Chrysoula Zerva, Ana C Farinha, Taisiya Glushkova, Alon Lavie, Luisa Coheur, and André FT Martins. 2022. COMET-22: Unbabel-IST 2022 Submission for the Metrics Shared Task. In *WMT@EMNLP*. Association for Computational Linguistics, 578–585.
- Ricardo Rei, Ana C. Farinha, Chrysoula Zerva, Daan van Stigt, Craig Stewart, Pedro G. Ramos, Taisiya Glushkova, André F. T. Martins, and Alon Lavie. 2021. Are References Really Needed? Unbabel-IST 2021 Submission for the Metrics Shared Task. In *WMT@EMNLP*. Association for Computational Linguistics, 1030–1040.

- Ricardo Rei, Craig Stewart, Ana C. Farinha, and Alon Lavie. 2020. COMET: A Neural Framework for MT Evaluation. In *EMNLP (1)*. Association for Computational Linguistics, 2685–2702.
- Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *EMNLP/IJCNLP (1)*. Association for Computational Linguistics, 3980–3990.
- Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *CoRR* abs/2009.10297 (2020).
- Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *NeurIPS*.
- Thibault Sellam, Dipanjan Das, and Ankur P. Parikh. 2020. BLEURT: Learning Robust Metrics for Text Generation. In *ACL*. Association for Computational Linguistics, 7881–7892.
- Sijie Shen, Xiang Zhu, Yihong Dong, Qizhi Guo, Yankun Zhen, and Ge Li. 2022. Incorporating domain knowledge through task augmentation for front-end JavaScript code generation. In *ESEC/SIGSOFT FSE*. ACM, 1533–1543.
- Weisong Sun, Chunrong Fang, Yuchen Chen, Guanhong Tao, Tingxu Han, and Quanjun Zhang. 2022. Code Search based on Context-aware Code Translation. In *ICSE*. ACM, 388–400.
- Ian Tenney, Dipanjan Das, and Ellie Pavlick. 2019. BERT Redisovers the Classical NLP Pipeline. In *ACL (1)*. Association for Computational Linguistics, 4593–4601.
- Yu Wan, Dayiheng Liu, Baosong Yang, Haibo Zhang, Boxing Chen, Derek F. Wong, and Lidia S. Chao. 2022. UniTE: Unified Translation Evaluation. In *ACL (1)*. Association for Computational Linguistics, 8117–8127.
- Pengcheng Yin and Graham Neubig. 2018. TRANX: A Transition-based Neural Abstract Syntax Parser for Semantic Parsing and Code Generation. In *EMNLP*. 7–12.
- Weizhe Yuan, Graham Neubig, and Pengfei Liu. 2021. BARTScore: Evaluating Generated Text as Text Generation. In *NeurIPS*. 27263–27277.
- Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2020. BERTScore: Evaluating Text Generation with BERT. In *ICLR*. OpenReview.net.
- Wei Zhao, Maxime Peyrard, Fei Liu, Yang Gao, Christian M. Meyer, and Steffen Eger. 2019. MoverScore: Text Generation Evaluating with Contextualized Embeddings and Earth Mover Distance. In *EMNLP/IJCNLP (1)*. Association for Computational Linguistics, 563–578.
- Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. 2023. CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code. *CoRR* abs/2302.05527 (2023).
- Ming Zhu, Karthik Suresh, and Chandan K. Reddy. 2022. Multilingual Code Snippets Training for Program Translation. In *AAAI*. AAAI Press, 11783–11790.

Appendix

A Effect of Binary CodeScore

In Table 6, we convert all EMs to binary EMs using thresholds picked in a similar way to binary CodeScore, and we compare their correlation with Pass@1 on three code evaluation datasets. As we can see, our Binary CodeScore still exhibits the best correlation with Ground Truth, compared to binary match-based CEMs and LLM-based EMs. Specifically, binary CodeScore is moderately correlated with Ground Truth, while binary match-based CEMs and LLM-based EMs have weak or extremely weak correlations with Ground Truth@. Moreover, the execution time of binary EMs does not increase significantly and is almost equal to EMs. Thus, we do not report it additionally, and it can refer to the execution time of EM in Table 3 and Table 4.

Table 6: Correlation comparison of functional correctness with binary EMs on code evaluation datasets, where three correlation coefficients are equal (i.e., $\tau = r_s = r_p$) for two Bernoulli distributed data.

Method (Binary)	MBPP-Eval	APPS-Eval	HE-Eval
	$r_s \uparrow$	$r_s \uparrow$	$r_s \uparrow$
Match-based CEM			
Binary BLEU	0.0928	0.0907	0.0887
Accuracy	0.0636	0.0369	0.0586
Binary CodeBLEU	0.2059	0.1666	0.2409
LLM-based EM			
Binary BERTScore	0.0972	0.0851	0.0375
Binary COMET	0.2358	0.1897	0.1181
CodeScore			
Binary CodeScore	0.4164	0.4176	0.4159

B Comparison with Execution-based CEMs

In Table 7, we can find that computing execution-based CEMs for code evaluation with a small number of original test cases is insufficient. They have a significant reduction in correlation coefficients compared to using larger extended test cases. The value of Pass@1 (ET) is 82.8% lower than Pass@1, meaning that 82.8% of correct codes under Pass@1 are misjudged on APPS. In cases where test cases are rare or low-quality, such as on APPS-Eval, the correlation between our CodeScore and Ground Truth even far exceeds that of AvgRassRatio. Therefore, the quality of AvgRassRatio’s evaluation depends on the quality and quantity of test cases.

In conclusion, AvgPassPatio and Pass@1 are effective but costly, since computing AvgPassPatio and Pass@1 not only requires adequate test samples but also dramatically increases the execution time. Therefore, in the situation of insufficient test cases, our CodeScore is a suitable alternative to them for quality and cost.

C Preliminary knowledge

For a task $p \in P$, let the test case set of p as $C_p = \{(\mathcal{I}_{p,c}, \mathcal{O}_{p,c})\}_{c \in C_p}$, a set of paired test case input $\mathcal{I}_{p,c}$ and test case output $\mathcal{O}_{p,c}$. Although the potential program space can be boundless, test cases permit automatic evaluation of code generation capability. Thus, in contrast to most other text generation tasks, human judgment is unnecessary for code generation. As the gold standards for code evaluation, AvgPassRatio and Pass@1 mirror the performance of generated codes on test cases.

AvgPassRatio The average proportion of test cases that generated codes \mathbf{g}'_p s pass:

$$\frac{1}{|P|} \sum_{p \in P} \frac{1}{|C_p|} \sum_{c \in C_p} \mathbb{I}\{\text{Eval}(\mathbf{g}_p, \mathcal{I}_{p,c}) = \mathcal{O}_{p,c}\}, \quad (15)$$

Table 7: Comparison of CodeScore and other EMs on APPS-Eval.

Method	Value	$\tau \uparrow$	$r_s \uparrow$	$r_p \uparrow$	MAE \downarrow	Execution Time \downarrow
Match-based CEM						
BLEU	0.0094	0.1055	0.1156	0.0959	0.1164	$1.0 \times (26.0s)$
Accuracy	0.0001	0.0079	0.0095	0.0196	-	$0.1 \times$
CodeBLEU	0.2337	0.1035	0.1533	0.1085	0.2005	$7.8 \times$
CrystalBLEU	0.0242	0.0906	0.1347	0.0887	0.1709	$0.3 \times$
LLM-based EM						
BERTScore	0.8629	0.0916	0.1375	0.0718	0.6874	$56.7 \times$
COMET	0.0165	0.0904	0.1126	0.1187	0.1751	$84.0 \times$
CodeBertScore	0.7583	0.1219	0.1801	0.1323	0.5885	$27.8 \times$
Execution-based CEM						
AvgPassRatio	0.0978	0.3360	0.4108	0.4987	0.1327	
Pass@1	0.0064	0.0894	0.1983	0.1598	-	$1.5k \times$
Pass@1 (ET)	0.0011 ($\downarrow 82.8\%$)	0.0470	0.0569	0.1174	-	20.7k \times
UniCE	0.1820	0.5275	0.7040	0.7210	0.1044	44.2 \times

where $|\cdot|$ indicates the element number of a set, $\mathbb{I}(\cdot)$ is an indicator function, which outputs 1 if the condition is true and 0 otherwise, and $\text{Eval}(\mathbf{g}_p, \mathcal{I}_{p,c})$ represents an evaluation function that obtains outputs of code \mathbf{g}_p by way of executing it with $\mathcal{I}_{p,c}$ as input.

Pass@1 The percentage of \mathbf{g}'_p s that pass all test cases of the corresponding p :

$$\frac{1}{|P|} \sum_{p \in P} \frac{1}{|C_p|} \prod_{c \in C_p} \mathbb{I}\{\text{Eval}(\mathbf{g}_p, \mathcal{I}_{p,c}) = \mathcal{O}_{p,c}\}. \quad (16)$$

Pass@1 is a more stringent CEM, which is a representative of the Pass@k family, also known as Strict Accuracy. In this paper, we use original and extended test cases to compute Pass@1 and Pass@1 (ET), respectively.

D Test case generation via ChatGPT

We randomly select 100 code generation tasks from the MBPP dataset and use the NL description and reference code of tasks to generate test cases via ChatGPT. Fig. 6 shows an example of ChatGPT generating test cases. ChatGPT generates an average of 1.53 test cases per task.

```

# Write a python function to find the first repeated character in a given string.
def first_repeated_char(str1):
    for index,c in enumerate(str1):
        if str1[:index+1].count(c) > 1:
            return c

# test the function with some example inputs

# test cases
assert first_repeated_char("abcdefg") == None
assert first_repeated_char("abcdabcd") == "a"
assert first_repeated_char("abcdcabcd") == "c"

```

Figure 6: Example of ChatGPT generating test cases.

The results shown in Fig. 7 indicate that **LLMs have the potential to judge the functional correctness of most programs with appropriate guidance**. Only 1.29% Generations consistent with private test cases means that ChatGPT generates test cases by itself instead of copying private test cases.

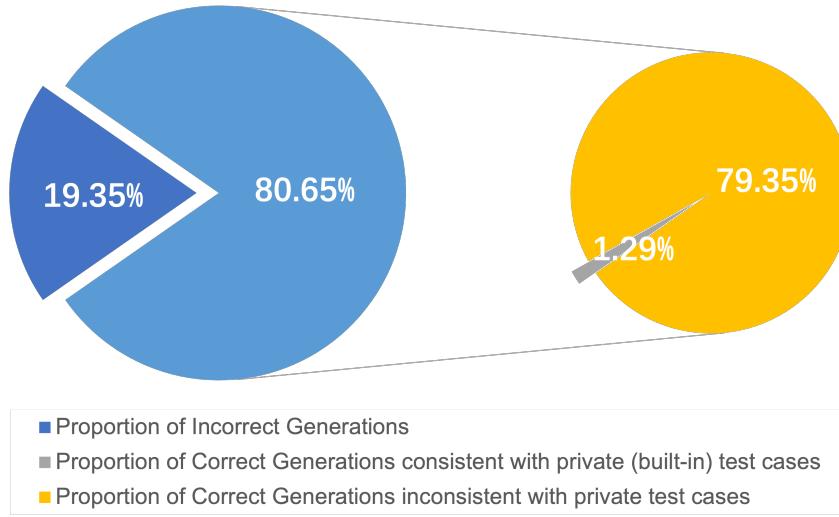


Figure 7: Test case generation via ChatGPT in zero-shot setting.