CrystalBLEU: Precisely and Efficiently Measuring the Similarity of Code

Aryaz Eghbali
aryaz.eghbali@iste.uni-stuttgart.de
University of Stuttgart
Stuttgart, Germany

Michael Pradel
michael@binaervarianz.de
University of Stuttgart
Stuttgart, Germany

ABSTRACT

Recent years have brought a surge of work on predicting pieces of source code, e.g., for code completion, code migration, program repair, or translating natural language into code. All this work faces the challenge of evaluating the quality of a prediction w.r.t. some oracle, typically in the form of a reference solution. A common evaluation metric is the BLEU score, an n-gram-based metric originally proposed for evaluating natural language translation, but adopted in software engineering because it can be easily computed on any programming language and enables automated evaluation at scale. However, a key difference between natural and programming languages is that in the latter, completely unrelated pieces of code may have many common n-grams simply because of the syntactic verbosity and coding conventions of programming languages. We observe that these trivially shared n-grams hamper the ability of the metric to distinguish between truly similar code examples and code examples that are merely written in the same language. This paper presents *CrystalBLEU*, an evaluation metric based on BLEU, that allows for precisely and efficiently measuring the similarity of code. Our metric preserves the desirable properties of BLEU, such as being language-agnostic, able to handle incomplete or partially incorrect code, and efficient, while reducing the noise caused by trivially shared n-grams. We evaluate CrystalBLEU on two datasets from prior work and on a new, labeled dataset of semantically equivalent programs. Our results show that CrystalBLEU can distinguish similar from dissimilar code examples 1.9–4.5 times more effectively, when compared to the original BLEU score and a previously proposed variant of BLEU for code.

CCS CONCEPTS

• **General and reference** → **Metrics**; **Evaluation**; • **Software and its engineering** → **Software notations and tools**; • **Computing methodologies** → *Machine learning*.

KEYWORDS

BLEU, Evaluation, Metric

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556903

ACM Reference Format:
Aryaz Eghbali and Michael Pradel. 2022. CrystalBLEU: Precisely and Efficiently Measuring the Similarity of Code. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556903

1 INTRODUCTION

The desire to automate repetitive software development tasks has led to a surge in techniques for predicting pieces of code of varying sizes. Many of these techniques build on the huge amounts of available source code to learn some kind of predictive model, e.g., using deep neural networks [39]. Code prediction techniques include code completion [4, 22, 23, 26, 46], code translation between programming languages [32, 33, 43], automated program repair [11, 12, 16, 24, 49], predicting code from natural language descriptions of the desired functionality [15, 21, 27, 44, 47], injecting bugs and vulnerabilities into existing code [34, 37], and predicting test oracles and test cases [6]. The amount of predicted code differs across techniques, ranging from a few tokens, over multi-line statements, to entire code files.

A commonality of all these techniques is the need for a metric to evaluate the quality of the predicted code. One of the most popular ways to address this need is the BLEU score. BLEU, which stands for "bilingual evaluation understudy", has originally been introduced in natural language processing as a way to automate the evaluation of machine translation [36]. Because BLEU can be easily adopted to any language that can be tokenized, including any programming language, and because BLEU enables automated evaluation at scale, it has become popular also for evaluating code prediction techniques. While surveying recent papers in software engineering, we find at least 21 papers published since 2015 that use BLEU as a metric to evaluate code prediction.

The basic idea of BLEU is to compare a prediction against one or more reference solutions by computing the overlap between n-grams, i.e., contiguous sequences of code tokens. Figure 1 illustrates this idea for comparing two predicted hypotheses against a reference solution. These pieces of code are slightly modified examples taken from a dataset of programs submitted to an online environment for competitive programming practice, where subsets of programs are labeled as semantically equivalent. The figure highlights some of the matching n-grams by showing them in the same color. For example, the four tokens in "`());`" occur in both the hypotheses and the reference code, and hence are a shared 4-gram. BLEU summarizes the similarity between the code pieces based on this and other shared n-grams, giving a higher score when more n-grams are shared between a hypothesis and the reference.

```java
// Reference:
import java.util.*;
public class Main {
  public static void main(String[] args ) {
    Scanner in = new Scanner(System.in);
    int t = in.nextInt();
    in.nextLine();
    while ( t-- > 0 ) {
      System. out.println( new StringBuffer(in.nextLine()).reverse
      ());
    }
  }
}
// Hypothesis 1: equivalent to the reference
import java.util.Scanner;
public class Main {
  public static void main(String argv[] ) {
    int num_of_tests = 0;
    Scanner in = new Scanner(System.in);
    num_of_tests= Integer.parseInt(in.nextLine());
    for(int i=0; i<num_of_tests; i++ ) {
      StringBuilder rev_str = new StringBuilder(in.nextLine());
      System. out.println( rev_str.reverse ());
    }
  }
}
// Hypothesis 2: not equivalent to the reference
import java.util.Scanner;
public class Main {
  public static void main(String[] args ) {
    Scanner in = new Scanner(System.in);
    while ( in.hasNext())
      System. out.println( in.nextInt() + in.nextInt ());
  }
}

```

Figure 1: Example of Java programs.

The example not only illustrates how BLEU works, but also highlights an important weakness of applying the metric to code. In contrast to natural languages, programming languages are syntactically verbose in the sense that the grammar prescribes various n-grams to be shared across completely unrelated code examples. For example, the 2-gram "`) {`", which is shared between all three examples in Fig. 1, is simply the result of the grammar, but does not imply any semantic similarity. Beyond syntax verbosity, common coding conventions and widely used APIs create even more n-grams shared across unrelated code, such as the shared 4-grams "`out.println(`" in the example. We call this phenomenon *trivially shared n-grams*, i.e., n-grams that occur across code written in the same language without implying any deeper relationship or semantic similarity. Because BLEU handles every n-gram the same, trivially shared n-grams hamper the metric's ability to distinguish actually similar code examples from examples merely written in the same language.

This paper presents *CrystalBLEU*, a metric to precisely and efficiently evaluate code similarity despite trivially shared n-grams. The approach is an extension of BLEU that removes trivially shared n-grams before computing the n-gram overlap between two pieces of code. Trivially shared n-grams can vary between programming languages and domains of programs. To identify trivially shared n-grams, CrystalBLEU analyzes a corpus of code and identifies n-grams that occur frequently across many examples. While conceptually simple, we find this change to significantly improve the ability of the metric to distinguish between semantically similar and dissimilar code. Similar to BLEU, CrystalBLEU relies only on syntactic similarities and uses them as a proxy for estimating semantic similarity. As a way to quantify the improvement in differentiating semantically similar and dissimilar code, we formulate in a novel meta-metric called *distinguishability*. In a nutshell, distinguishability measures how much more similar code examples known to be semantically equivalent are compared to code examples that are not equivalent to each other.

We evaluate our work with three datasets. One dataset is the *Concode* dataset [21], which consists of more than 100,000 pairs of Java code and natural language descriptions. We use this dataset to show a case for which BLEU is not a suitable metric when comparing models, while CrystalBLEU can provide useful comparison. Another dataset is *BigCloneBench* [45, 51], which consists of more than 1.7 million pairs of clone and non-clone programs in Java. We also use a new dataset of Java and C++ programs with labels that indicate sets of semantically equivalent programs from ShareCode.io, which consists of more than 6,000 Java and more than 20,000 C++ programs. These two datasets allow us to evaluate similarity metrics on objective ground truths. We compare CrystalBLEU against two baselines: the original BLEU score [36] and CodeBLEU [41], a previously proposed variant of BLEU for code. Our results show that CrystalBLEU provides higher distinguishability than both baselines, i.e., it is more effective at distinguishing semantically similar code from code merely written in the same language. Despite these benefits, the running time of CrystalBLEU and BLEU are similar, making our metric attractive for large-scale and even online evaluation of code similarity, e.g., as part of a loss function for training a machine learning model.

In our example from Fig. 1, the BLEU score for Hypothesis 1 and the Reference is 0.48, and for Hypothesis 2 and the Reference it is 0.55. However, we expect to get a higher similarity score for the equivalent pair, which is not the case in this example. The example shows how trivially shared n-grams can mislead the BLEU score to report a higher similarity for dissimilar code pieces than for similar ones. In contrast to BLEU, the CrystalBLEU score for Hypothesis 1 and the Reference is 0.24, and for Hypothesis 2 and the Reference it is 0.0. That is, CrystalBLEU more accurately represents the semantic (dis)similarities than BLEU.

Prior work has also highlighted weaknesses of applying BLEU to code and proposed two alternative metrics, RUBY [48] and CodeBLEU [41]. They exploit the fact that code has a well-defined structure, in the form of an AST, and well-defined relationships between code elements, in the form of data-flow and control-flow dependencies. However, both approaches are only applicable when it is possible to compute an abstract syntax tree and semantic dependencies of the analyzed code pieces. Because code prediction techniques often output isolated code fragments, e.g., catch blocks [54], and because the predicted code may contain syntactic and semantic mistakes, calculating ASTs and dependencies may not be possible. Even when that is possible, relying on program analysis means that a separate implementation is required for each programming

language and that the metrics are less efficient than BLEU and CrystalBLEU. Finally, RUBY and CodeBLEU partially build upon BLEU, and hence, also suffer from trivial n-gram matches. The CodeBLEU score for Hypothesis 1 and the Reference from Fig. 1 is 0.51, and for Hypothesis 2 and the Reference it is 0.57, which means the equivalent pair receives a higher score than the non-equivalent pair.

In summary, this paper contributes the following:

- The observation that the most common n-grams in programming languages appear relatively more often than the most common n-grams in natural languages, and that they appear in many different programs regardless of their semantic similarity.
- A meta-metric, called distinguishability, to measure the effectiveness of code similarity metrics at distinguishing semantically equivalent code from code merely written in the same language.
- A new metric for evaluating code similarity, called CrystalBLEU, which ignores trivially shared n-grams when comparing two pieces of code.
- Empirical evidence that, compared to BLEU, CrystalBLEU achieves higher distinguishability, while being as efficient as the original metric. We also show that CrystalBLEU helps avoid drawing misleading conclusions about neural code prediction models.

2 BACKGROUND

2.1 BLEU Score

The bilingual evaluation understudy, commonly referred to as BLEU, was proposed by Papineni et al. [36] as an inexpensive metric to evaluate the quality of translated natural language text. It is calculated by first computing the *modified precision* of n-grams, for $n \in \{1, 2, \ldots, maxN\}$. Then, the weighted geometric mean of these modified precision values, times a *brevity penalty*, yields the BLEU score, a number in $[0, 1]$. To be self-contained, we briefly explain how BLEU is computed with the assistance of the example in Fig. 1, and refer readers to Papineni et al. [36] and NLTK's implementation of BLEU[1] for more details.

Given a corpus of hypotheses and their reference sentences (one or more reference sentences per hypothesis sentence), the modified precision of n-grams, for a fixed value of n, is computed as follows. First, for each n-gram a *clipped count* is calculated, which is the minimum of the number of occurrences of that n-gram in the hypothesis sentence and the maximum number of occurrences in the reference sentences. For the example in Fig. 1, "`) {`" appears once in Hypothesis 1 and twice in the reference, so the clipped count for this 2-gram is two. The 3-gram "`for ( int`" appears once in Hypothesis 1 and the 3-gram "`hasNext()`" appears once in Hypothesis 2 but none of them appear in the reference, which make their clipped counts zero. Second, for each $n \in \{1, \ldots, maxN\}$ these clipped counts are summed for all n-grams in each hypothesis sentence and summed over all hypothesis sentences in the corpus. Then, this sum is divided by the sum of all n-gram counts of all the hypothesis sentences, hence the name *modified precision*.

[1] https://www.nltk.org/_modules/nltk/translate/bleu_score.html

The modified n-gram precision values for all $n \in \{1, \ldots, maxN\}$ are combined using the weighted geometric mean to mitigate the exponential drop for larger n. To decrease the calculation errors and avoid underflows, the weighted geometric mean is calculated using the weighted arithmetic mean over the logs of the modified precision values and exponentiated at the end, as shown in Eq. (1). In [36] and all the papers mentioned in this study, the weighted arithmetic mean is calculated using equal weights. To avoid favoring hypotheses that are shorter than their reference sentences, which gives a higher modified precision, a penalty is multiplied with this result, called the *brevity penalty*. This penalty is calculated over the whole corpus, to avoid punishing hypothesis sentences for not being of equal length to their corresponding references. Therefore, for each hypothesis sentence, the length of the reference sentence that is closest to the length of the hypothesis is added to a sum, r. The total length of the hypothesis corpus is denoted as c. If $r < c$, then the brevity penalty is 1, otherwise, it is $e^{1-r/c}$. In our example, since the first hypothesis is longer than the reference, the brevity penalty is one, i.e., no penalty, but because the second hypothesis is shorter than the reference, the brevity penalty is $e^{1-77/58}$.

To summarize, the BLEU score is calculated with the following formula

$$BLEU = exp\left(min(1 - \frac{r}{c}, 0) + \sum_{i=1}^{maxN} w_i \log p_i\right), \qquad (1)$$

where, w_i is the weight and p_i is the modified precision of i-grams.

Since the BLEU score is calculated using the geometric mean of the modified precision values. If at least one of the modified precision values is zero, the BLEU score is also zero. To mitigate this rapid drop, some smoothing techniques have been proposed [9].

2.2 BLEU on Code

In recent years research in automated code generation has increased, and with that increase comes the need for evaluation metrics. One of the commonly used metrics for code similarity is BLEU, since it can handle different programming languages ranging from VHDL [25], custom languages [31], and UI tags [10], to more conventional languages like Java [15, 27, 47, 52, 54], C++ [17, 43], and Python [29, 30, 44]. Another appeal of BLEU lies in its applicability to any piece of code, regardless of syntactic correctness or completeness. It has been used to assess a variety of code sizes such as code on the right-hand side of assignments [25], single statements [52], single lines of code [13, 35], sequences of API calls [15, 27, 47], blocks of code [54], full methods [21, 32, 33], and full classes [44].

Applying BLEU to source code raises the question what a sentence is. In natural language, one sentence from the hypothesis corpus is associated with one sentence from each corresponding reference, but for source code there exists no such obvious association. Therefore, BLEU is commonly applied to code by considering an entire source code snippet as one sentence.

3 APPROACH

This section presents the CrystalBLEU metric for measuring the similarity of code in a precise and efficient way. We start by motivating our work through the observation that natural languages and programming languages differ in that the latter has a higher number

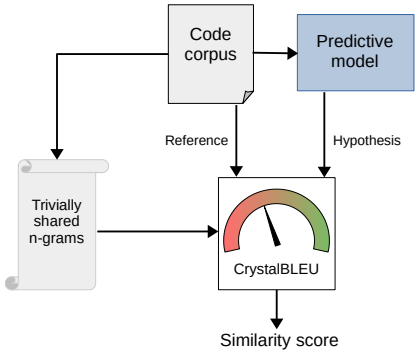

Figure 2: Overview of the approach.

of trivially shared n-grams (Section 3.1). We then describe a novel meta-metric, called distinguishability, for assessing to what extent a code similarity metric distinguishes between semantically similar code and code merely written in the same language (Section 3.2). Finally, we present the CrystalBLEU metric, which addresses the problem of trivially shared n-grams in a way that increases distinguishability compared to BLEU (Section 3.3).

Figure 2 shows an overview of the approach for computing CrystalBLEU. The core idea is to identify trivially shared n-grams in a representative corpus of code, e.g., the corpus used to train the predictive model, and to then account for the n-grams while comparing a reference code example to the hypothesis predicted by the model.

3.1 Trivially Shared N-grams

Natural languages and programming languages share many commonalities [18], an observation exploited in various approaches that adapt techniques from natural language processing to code [1, 39]. In particular, these similarities have motivated the use of BLEU on code. However, we observe an important difference between the two kinds of languages, which affects the adoption of BLEU to code: *Code examples written in the same programming language trivially share various n-grams, irrespective of how semantically similar two code examples are.* We call such n-grams *trivially shared n-grams.*

Trivially shared n-grams are mainly due to two reasons. The first reason is that the syntax of a programming language often enforces multiple tokens to appear together. For example, in many programming languages the structure of a "`for`" loop is fixed and causes several n-grams to appear no matter what exactly the loop is about, such as "`for (`" or "`) {`". Grammar-induced, trivially shared n-grams are particularly common in programming languages because they have a well defined grammar and an often verbose syntax. This is inherently different from natural languages, where the grammar is more relaxed and enforces minimal syntactical notations.

The second reason for trivially shared n-grams are common coding conventions and popular APIs, which cause similar code fragments to appear across various programs. For example, all Java programs with a "`System.out.println(...);`" statement share several n-grams of different sizes, without having any strong semantic relationship beyond the fact that they print something. Another example would be the main function in languages like C++ and Java, where the signature and the name are in most cases fixed.

Table 1: Top common 2-grams and 4-grams of Java and English languages.

	2-grams	% of 2-grams	4-grams	% of 4-grams
Java	) ;	5.49	) ; } }	1.34
	( )	4.75	( ) { return	1.29
	) {	3.83	( ) ; }	1.14
	; }	3.81	)) ; }	1.12
	; import	2.75	) ; } public	1.00
	))	1.73	( )) ;	0.94
	} public	1.27	) { this .	0.68
	{ return	1.24	; } public void	0.61
	} }	1.07	; } @Override public	0.54
	) .	0.99	) { if (	0.54
English	of the	2.31	” , he said	0.56
	, and	1.45	, he said ,	0.32
	in the	1.31	, of course ,	0.31
	” .	1.01	” , I said	0.29
	, the	0.85	” , she said	0.29
	to the	0.85	, he said .	0.27
	. “	0.64	” . “ I	0.22
	” ,	0.63	he said , “	0.21
	on the	0.57	” ? asked .	0.19
	, but	0.53	, I said .	0.19

To validate our hypothesis that trivially shared n-grams are a programming language-specific phenomenon, we present three experiments.

Most frequent n-grams. Table 1 illustrates the phenomenon with examples from English and Java. The English language data is extracted from the Brown dataset[2], which is commonly used in the area of natural language processing. As a Java corpus, we use the Java-small dataset[3], which consists of open-source projects. The table shows the ten most frequent 2-grams and 4-grams in Java code and English text. Next to the n-grams, the table also shows the percentage of all occurrences that a specific n-gram contributes. The percentages of the most frequent are clearly higher for Java than for English, showing that common n-grams contribute a larger share in Java. For example, the five most common 2-grams in Java each contribute more than 2.5% of all 2-grams in Java, whereas even the single-most common 2-gram in English contributes less than 2.5%.

Distribution of frequent n-grams. To further validate our hypothesis that trivially shared n-grams are particularly prevalent in programming languages, we extend our measurement of the frequencies of common n-grams in both natural language and code corpora beyond the top 10 examples shown in Table 1. Figure 3 shows the frequencies of the top occurring n-grams in English and Java, using the same datasets as above. The chart axes are in log scale to show the fact that a few most occurring n-grams in programming languages are much more frequent than the most occurring n-grams in natural languages, but for the less frequent n-grams (i.e. the tails

[2]Available in NLTK's data at http://www.nltk.org/nltk_data/.
[3]Used in Alon et al. [3] and available at https://github.com/tech-srl/code2seq/#datasets.

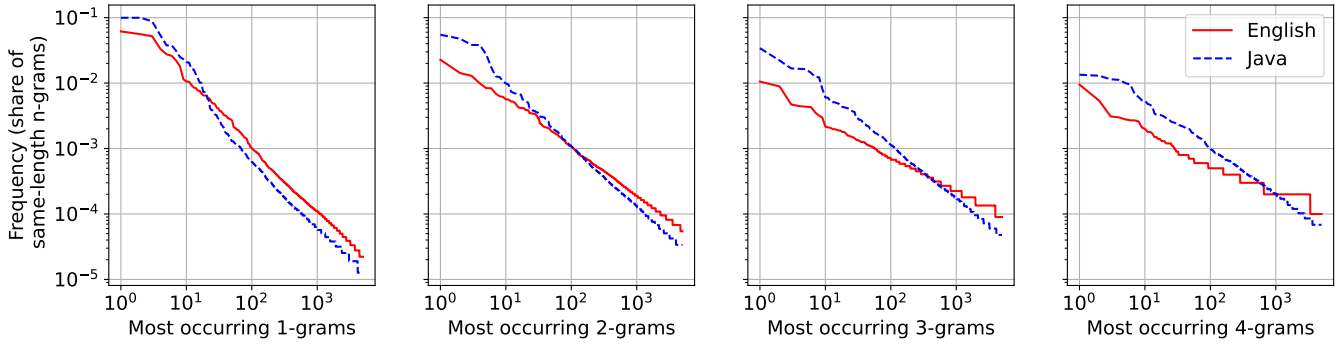

Figure 3: Frequency of the most common Java n-grams (blue, from Java-small dataset) and English n-grams (red, from Brown dataset). Both plot axes use log scale to better visualize the differences.

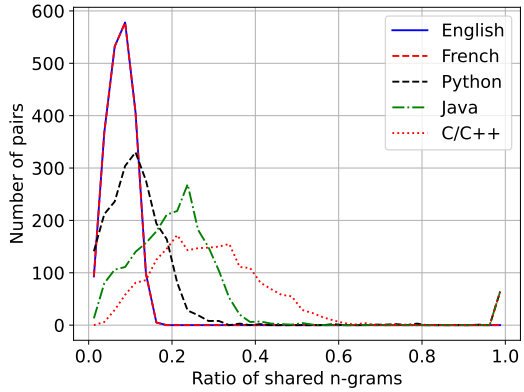

Figure 4: Ratio of shared n-grams in English, French, Python, Java, and C/C++.

of the distribution) it is the opposite. We see that there are tens of n-grams in Java that appear many times (much more than the most common n-grams of the English language), while less common n-grams in the English language appear more than the less common n-grams in Java. In other words, common n-grams occur more frequently in Java than in English.

N-grams shared by randomly selected pairs. As a third experiment, we randomly select code or text examples in the same language and compute how many n-grams they share, as shown in Fig. 4. In addition to the datasets used above, we also show results for French using the Europarl French-English dataset[4], Python using the Python150k dataset[5], and C/C++ using the POJ-104 dataset[6]. We observe that, on average, two random programs share more n-grams than two random pieces of natural language text. This means that given two pieces of code, regardless of their semantic similarity, they are expected to have more n-grams in common than two pieces of text in a natural language like English. Note that the curves for English and French languages are on top of each other (the red and blue curve with the highest peak), but have a small difference in their distribution (the mean for English is 0.08 and the mean for French in 0.10).

How do n-grams that are common across a corpus influence the BLEU score? Because the metric does not differentiate between n-grams based on their frequency, each n-gram contributes equally to the computed similarity. That is, the more n-grams trivially match even for completely unrelated code examples, the less precisely can BLEU represent whether two code examples are semantically similar or dissimilar. The idea behind CrystalBLEU is to mitigate this effect by focusing on more informative n-gram matches.

3.2 Distinguishability

Before describing how CrystalBLEU addresses the challenge of trivially shared n-grams, the following presents a way to measure how precisely any code similarity metric represents semantic similarities. To this end, we exploit the fact that code, in contrast to natural language, provides an unambiguous oracle for semantic equivalence through its execution semantics. Intuitively, a metric should report a higher similarity for code examples that have the same execution behavior, while reporting a lower similarity for code examples with different behavior.

We formulate this intuition into the notion of the *distinguishability* of a similarity metric as follows. Suppose a similarity metric m and set C of source code examples. The metric is given a set of pairs, where each pair consists of a set of reference code examples and a hypothesis to evaluate against the references. For example, the references in such a pair may be one or more known to be correct code snippets and the hypothesis is a code snippet predicted by some model. Formally, $m : \mathcal{P}(\mathcal{P}(C) \times C)) \rightarrow \mathbb{R}_{\geq 0}$, where $\mathcal{P}$ denotes the power set, and $\mathbb{R}_{\geq 0}$ is the set of non-negative real-valued scores the metric may compute.

Further suppose the set C of source code examples is composed of a disjoint set of equivalence classes $C_1, C_2, \ldots, C_n$ that represent semantically equivalent code examples. That is, $C_1 \cup C_2 \cup \cdots \cup C_n = C$, and for each $i \in \{1, \ldots, n\}$, all programs in C_i are equivalent to each other but not equivalent to any program in C_j for $j \neq i$. Given

[4]Available at https://www.statmt.org/europarl/
[5]Available at https://www.sri.inf.ethz.ch/py150
[6]Available at https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Clone-detection-POJ-104

these sets, a metric has high distinguishability if it yields high similarity scores for code examples within an equivalence class, called *intra-class similarity*, but low similarity scores for code examples from different equivalence classes, called *inter-class similarity*:

Definition 3.1 (Distinguishability). Let $Pairs_{intra}$ and $Pairs_{inter}$ be inputs to a metric m where

$$Pairs_{intra} = \{(C_i \setminus \{c_a\}, c_a) \mid c_a \in C_i\}$$

$$Pairs_{inter} = \{(C_j, c_a) \mid c_a \in C_i, i \neq j\}$$

for $i, j \in \{1, \ldots, n\}$. The distinguishability d of m is:

$$d = \frac{m(Pairs_{intra})}{m(Pairs_{inter})}$$

Based on this definition, distinguishability is higher when the intra-class similarity is higher, and lower when the inter-class similarity is higher. Distinguishability is always a positive number. A metric that returns random similarity scores would have an expected distinguishability of one. Distinguishability above one means the metric can distinguish similar from dissimilar code examples, where higher distinguishability means the metric is more precise at this task.

Since the number of intra-class pairs is quadratic in the number of programs in each class and the number of inter-class pairs is quadratic in the number of total programs, computing the similarity metrics for all intra- and inter-class pairs may become impractical for large code corpora. Instead, we use a sampling-based approximation of distinguishability that randomly samples N intra-class pairs and N inter-class pairs, and then computes distinguishability for them. We use this approximation of distinguishability for datasets that have a large number of program pairs, with $N = 1,000$ as a default.

3.3 CrystalBLEU

We now present the CrystalBLEU metric, which in contrast to BLEU, increases distinguishability by accounting for trivially shared n-grams. The basic idea behind the metric is surprisingly simple: At first, we identify trivially shared n-grams. Then, we compute a revised BLEU score that accounts for the trivially shared n-grams. The following explain the full algorithm in more detail. To be self contained, we also describe those parts that are the same as in the original BLEU calculation.

3.3.1 Identifying Trivially Shared N-grams. It has been observed in many domains that the more frequent an n-gram, word, or phrase is, the less information it conveys when appearing in a document. For example, this observation is the basis for term frequency–inverse document frequency (TFIDF), which is commonly used as a weighting factor in information retrieval methods. To identify trivially shared n-grams in a corpus of code examples, we also exploit the frequency of an n-gram occurring in the corpus.

Concretely, before applying CrystalBLEU to a code corpus, we compute all n-grams, along with their frequencies, in this corpus. Then, we gather the k most common n-grams in set S, where k is a parameter of the algorithm. CrystalBLEU heuristically considers this set S as trivially shared n-grams. For example, when using CrystalBLEU to compute how accurately Concode [21] predicts functions, we compute S based on the corpus of functions that Concode is trained on. Throughout this paper we use $k = 500$ for our experiments and explain this decision in detail in Section 4.6.

Input: S: counts of k most occurring n-grams, *hyps*: list of tokenized hypotheses, *refs*: list of tokenized references, *options*: other BLEU options like weights, smoothing methods, etc.
Output: CrystalBLEU score

```
numerator, denominator ← [0...0]
for (ref, hyp) in (refs, hyps) do
    for i ∈ [1..maxN] do
        numerator_i, denominator_i ←
          numerator_i, denominator_i +
          modified_precision(ref, hyp, i, S)
    end
end
bp ← brevity_penalty(refs, hyps)
for i ∈ [1..maxN] do
    p_i ← numerator_i / denominator_i
end
if smoothing then
    apply smoothing
end
for i ∈ [1..maxN] do
    s_i ← weight_i * p_i
end
score ← bp * exp(∑_i s_i)
return score
Function modified_precision(ref, hyp, i, S) is
    refCounts ← n-grams of length i from each ref and
      their number of occurrences
    hypCount ← n-grams of length i from hyp and their
      number of occurrences
    remove any n-grams from refCounts and hypCount
      that is in S, or divide refCounts and hypCount by the
      logarithm of counts in S
    for ngram ∈ hypCount do
        clipped_count_ngram ←
          min(hypCount_ngram, max(refCounts_ngram))
    end
    numerator ← ∑_i clipped_count_i
    denominator ← max(1, ∑ hypCount)
    return numerator, denominator
end
```

Algorithm 1: CrystalBLEU. Differences to BLEU are highlighted.

3.3.2 CrystalBLEU Algorithm. Once the set S of the most common n-grams are computed, the next step is to decrease their impact on the computation of the BLEU score. To calculate the BLEU score, as mentioned in Section 2, the first step is to calculate the modified precision of n-grams for the hypotheses. We implement and evaluate two approaches for considering trivially shared n-grams in this

step. One approach is to completely remove all n-grams in S from the modified precision calculations. Another, more complex process is to apply weights proportional to the inverse frequency of the most common n-grams. We empirically observe both approaches to provide similar effects on the computed similarity metrics and on distinguishability, and hence, focus on the first, simpler approach in the remainder of the paper.

Algorithm 1 shows the pseudo-code of CrystalBLEU and how it differs with BLEU. The complete code is available as part of our artifact. The important difference to BLEU is in line 22, where the algorithm removes n-grams in S (or decreases their weight) from those considered when computing the modified precision. After the modified precisions of n-grams are computed, they are summed up and then the brevity penalty, weights, and smoothing method are applied to obtain the final score, as in the original BLEU score.

Our implementation of CrystalBLEU extends the NLTK implementation of BLEU. Specifically, we use the corpus BLEU evaluation (`corpus_bleu` method).

4 EVALUATION

In this section we answer the following research questions with empirical evidence.

RQ1: How well does CrystalBLEU distinguish between similar and dissimilar code?
RQ2: Can CrystalBLEU avoid misleading results provided by BLEU?
RQ3: How efficient is computing CrystalBLEU?
RQ4: How sensitive is CrystalBLEU to the number k of trivially shared n-grams?

4.1 Baselines

For RQ1 to RQ3, we compare our CrystalBLEU metric to the existing BLEU metric [36], as it is widely used on code, and to CodeBLEU [41], because it is a recently proposed alternative to BLEU specifically designed for code. As an implementation of BLEU, we use the vanilla implementation provided by NLTK. The CodeBLEU implementation is extracted from the CodeXGLUE repository[7].

4.2 Datasets

ShareCode: Semantically equivalent, human-written code. The first dataset is a collection of solutions to programming challenges. We include this dataset because it provides us with sets of diverse yet semantically equivalent code snippets. The dataset is from ShareCode[8], an online coding competition website that offers programming problems and an online judge. The online judge runs the code submitted by programmers on comprehensive test suites and accepts only those solutions that pass all tests. Because the accepted code pieces in a problem set pass all tests, we consider them to be semantically equivalent. The semantically equivalent pairs can be any of the four clone types, but the labeling criteria is only functionality. We use the Java and C++ submissions of ShareCode, and remove problems with fewer than five accepted solutions. In total, there are 6,958 code pieces in this dataset, which cover 278 programming problems. N-grams "`= 0;`", "`.out.println`", and "`i++`" are some of the most common n-grams in the Java language part of this dataset.

BigCloneBench: Clone and non-clone pairs. This dataset contains more than 1.7 million inter-project pairs of Java programs, each labeled with "clone" or "not clone". Because of the labeling process, it contains clone types one to four. The data is split into train (~901k), validation (~415k), and test (~415k) sets. It is available on the CodeXGLUE repository.[9]

Concode: Code generation task. This dataset, which originates from Iyer et al. [21], is extracted from the CodeXGLUE repository. It contains pairs of Java code and natural language descriptions of the code. The task for which this dataset is curated for is predicting a function, given a natural language description that also contains some context information. The dataset contains 100,000 training data points, and we have the predictions of a neural model and ground truth for 100 test data points. Because of reproducibility issues we were not able to obtain the predictions for the full test set.

Table 2: Distinguishability of different code similarity metrics (higher is better).

Language	BLEU	CodeBLEU	CrystalBLEU
ShareCode Java	2.47	1.44	**6.50**
ShareCode C++	2.82	N/A	**8.29**
BigCloneBench	1.44	1.18	**2.77**

4.3 RQ1: Distinguishing Similar and Dissimilar Code

We measure the ability of CrystalBLEU and the baseline metrics to distinguish semantically equivalent from semantically non-equivalent code examples in two ways. Section 4.3.1 uses our novel distinguishability metric. Section 4.3.2 applies CrystalBLEU and other metrics in a simple, threshold-based classifier that predicts whether two examples are equivalent.

We perform these experiments on the two datasets that provide equivalent and non-equivalent code pairs, i.e., ShareCode and BigCloneBench. For ShareCode, we use the equivalency classes inherent to the dataset, i.e. two programs are considered equivalent, and hence, considered to be intra-class (Definition 3.1), if they solve the same task. For BigCloneBench, we assume pairs of clone programs as edges representing intra-class relations and non-clones as edges showing inter-class relations.

4.3.1 Distinguishability Metric. Table 2 shows the distinguishability of CrystalBLEU, BLEU, and CodeBLEU for the two datasets. We find CrystalBLEU to achieve a clearly higher distinguishability than the two baseline metrics. For example, on the ShareCode Java and C++ programs, CrystalBLEU outperforms BLEU by a factor of 2.6x and 2.9x, respectively. Compared to CodeBLEU, CrystalBLEU achieves a more than 4x higher distinguishability on the ShareCode

[7] https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/code-to-code-trans/evaluator/CodeBLEU
[8] https://sharecode.io/
[9] https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Clone-detection-BigCloneBench

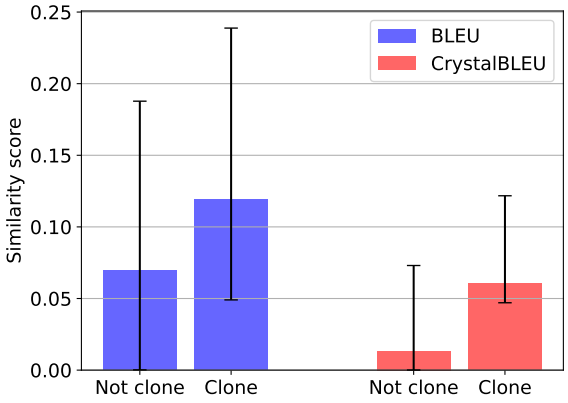

Figure 5: Average and 95% confidence interval of BLEU and CrystalBLEU scores for each class in the BigCloneBench test set.

Table 3: Code clone detection task results for BLEU- and CrystalBLEU-based models.

	ShareCode		BigCloneBench	
	BLEU	CrystalBLEU	BLEU	CrystalBLEU
TP	2,789	2,131	29,305	26,064
FP	2,253	531	113,668	44,397
TN	21,147	22,869	244,928	314,199
FN	811	1,469	27,515	30,756
Accuracy	0.88	**0.92**	0.66	**0.82**
Precision	0.55	**0.80**	0.20	**0.37**
Recall	**0.77**	0.59	**0.52**	0.46
F1 score	0.64	**0.68**	0.20	**0.37**

Java dataset. As there is no C++ support in CodeBLEU, the distinguishability of the C++ part is not available for CodeBLEU. Overall, these experiments show that CrystalBLEU is much more effective at distinguishing equivalent from non-equivalent code examples.

4.3.2 Threshold-based Classification. As another way to answer RQ1, we assess the ability of CrystalBLEU and other metrics to decide whether two code examples are equivalent. To illustrate the idea, Figure 5 shows the mean and the 0.95 confidence interval of BLEU and CrystalBLEU scores for pairs of non-clones and clones in the BigCloneBench dataset. The figure shows that, while both metrics return lower similarity scores for non-clones than for clones, the differences are more accentuated with CrystalBLEU. First, the relative difference between non-clones and clones is higher for CrystalBLEU than for BLEU. Second, the confidence intervals of CrystalBLEU are less overlapping, which is a result of its higher distinguishability.

Motivated by these observations, we design a simple, threshold-based classifier that predicts whether two code pieces are equivalent based on their similarity score. If the similarity score of a pair is above some threshold, the classifier predicts the pair to be equivalent, and to be non-equivalent otherwise. To determine the threshold, we calculate the average similarity score s_{equiv} of all equivalent and the similarity score $s_{nonEquiv}$ of all non-equivalent pairs in a training set, and then use the means of these two values as the threshold. Table 3 shows the results of such a classifier on ShareCode and BigCloneBench. Comparing BLEU and CrystalBLEU shows that CrystalBLEU's ability to better distinguish non-equivalent from equivalent pairs translates into a classifier with higher accuracy and higher F1 score. For example, the accuracy at identifying clones and non-clones increases from 0.66 with BLEU to 0.82 with CrystalBLEU. Higher recall for BLEU comes from the fact that BLEU tends to overestimate the similarity of pairs, because of the trivially shared n-grams. This overestimation of BLEU results in more positive predictions and fewer negative predictions compared to CrystalBLEU, which in turn results in higher recall and lower precision. Note that these results are not intended to compete with state of the art clone detection techniques, which achieve even higher accuracy, but instead serve to illustrate the benefits of CrystalBLEU over BLEU.

Finding 1: CrystalBLEU is clearly more effective at distinguishing equivalent from non-equivalent code pairs compared to the existing BLEU and CodeBLEU metrics, providing a more precise code similarity metric.

4.4 RQ2: Avoiding Misleading Results

The noise caused by trivially shared n-grams might affect BLEU and cause misleading conclusions when comparing different techniques that predict code. The following shows an example of such a misleading result and that CrystalBLEU avoids it. To this end, we use Concode [21], which generates code from a natural language text using a neural model. As a hypothetical competitor to Concode, we create an artificial dummy model designed to appear to be as good as Concode, while actually producing clearly worse code than Concode. We assume that this dummy model knows how many tokens are in the expected code, some of the correct tokens, and the trivially shared n-grams in this domain. The model generates its output as follows: Until the length of the prediction is shorter than the correct solution, with 82% probability add a trivially shared n-gram to the beginning of the prediction. With probability $1 - 0.82$, append a token from the correct solution to the end of the prediction. The probability of 82% is chosen to yield a dummy model that achieves exactly the same BLEU score as the Concode model. This means that the dummy model is generating code that in most cases starts with large section of nonsensical but common tokens, followed by a small piece of correct code. Because most of the code predicted by the dummy model simply consists of trivially shared n-grams, its predictions clearly do not succeed in the task of predicting code for a given natural language description.

Fig. 6 shows the results of evaluating Concode's neural model and the dummy with BLEU and CrystalBLEU. By design, the BLEU score of both models are very similar, i.e., BLEU may lead to the wrong conclusion that the two models are equally successful. In contrast, CrystalBLEU shows that the neural model is 1.2x better than the dummy model. Following the algorithm suggested by Riezler and Maxwell III [42] we perform a statistical test to verify

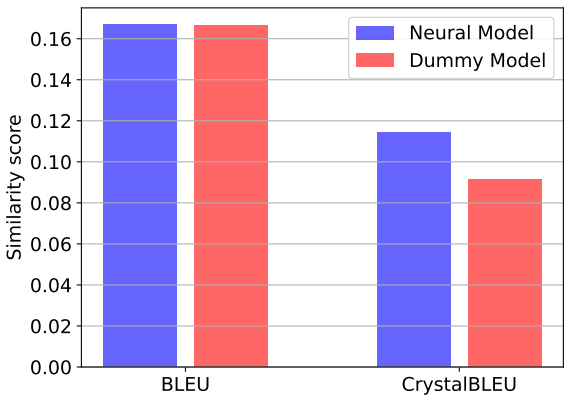

Figure 6: Similarity scores of the neural and dummy models.

the significance of the differences for both BLEU and CrystalBLEU. Setting the null hypothesis as "the dummy model is as good as the neural model", we obtain p-values of 0.22 for BLEU and 0.01 for CrystalBLEU. This rejects the null hypothesis for CrystalBLEU, but not for BLEU, showing that CrystalBLEU helps distinguish the two models, whereas BLEU shows them to be the same.

Finding 2: CrystalBLEU can avoid misleading results, e.g., caused by a model that predicts trivially shared n-grams instead of solving the actual task.

4.5 RQ3: Scalability

One of the most important selling points of BLEU is its running speed. It is so fast that it is even used as an online metric [8]. We conduct a set of experiments to evaluate the running time of CrystalBLEU and compare it to BLEU and CodeBLEU. For these experiments we use a regular laptop running a 64 bit Ubuntu 20.04, with an Intel Core i7 CPU (8 x 1.8 GHz) and 16 GB of RAM. We use Python 3.9's "`time.process_time`" method for all time measurements.

The running time for the CrystalBLEU score calculations depend on the size of set S. Table 4 shows the running times of BLEU, CodeBLEU, and CrystalBLEU for different datasets with $|S| = k = 500$. In the two variants of CrystalBLEU we present in this paper, the one that ignore the trivial n-grams gets faster when the size of S increases, but the one that divides the n-grams counts by their frequency in the sample corpus gets slower with an increase in the size of S.

We also measure the preprocessing time required to obtain the n-gram counts on a sample corpus from each dataset. For ShareCode, we use a sample of all programs in the dataset. It takes less than 5 seconds to process a sample corpus of 580K tokens, and less than 30 seconds for a corpus of 2.6M tokens.

Table 5 shows the preprocessing time required for CrystalBLEU to gather the most common n-grams from a code corpus. It is worth emphasizing that the preprocessing phase is only needed to be done once for a task-language pair.

Table 4: Running times (in milliseconds per 100 pairs of code pieces) of BLEU, CodeBLEU, and CrystalBLEU.

	BLEU	CodeBLEU	CrystalBLEU
ShareCode Java Intra-class	1036.9	5382.3	953.6
ShareCode Java Inter-class	868.9	3848.3	743.6
BigCloneBench Intra-class	83.5	1445.1	85.7
BigCloneBench Inter-class	81.5	1269.4	81.7
Concode Java	14.2	133.8	13.6

Table 5: Preprocessing time (in seconds) of CrystalBLEU on different datasets.

Dataset	Number of tokens	Preprocessing time (s)
ShareCode Java	580K	4.8
ShareCode C++	1.8M	19.9
BigCloneBench	2.6M	22.3
Concode (tokenized)	2.6M	4.1

Our results show that CrystalBLEU can be used at scale without degradation in running time performance. The preprocessing time of under one minute is also acceptable for almost all use cases, as it is a one-time calculation.

In terms of memory usage, CrystalBLEU first extracts a list of all n-grams and then stores a dictionary of n-grams and their frequencies. The length of the list of all n-grams is at most $maxN^2$ times the number of tokens, which is in the same order as the input corpus. Moreover, the dictionary storing the frequencies has fewer entries than the list of all n-grams. Overall, for our largest sample corpus, the whole Python process uses at most 920MB of memory. Hence, CrystalBLEU can be used for all use cases in code that BLEU has been used for previously.

Finding 3: CrystalBLEU is as efficient as BLEU, allowing for large-scale evaluations of code similarity in little time.

4.6 RQ4: Parameter Choice

Choosing a suitable parameter k for CrystalBLEU is important as it affects the core competence of this new metric. This parameter also affects the running time of CrystalBLEU. However, there is a sizable range of suitable ks for each task and language pair by design, that yields benefits over BLEU. CrystalBLEU score decreases by increasing k down to the point where it reaches 0, where all n-grams matched are in set S. Therefore, increasing k causes an increase in distinguishability, as shown in Fig. 8. Our analysis shows the best performance of CrystalBLEU on complete Java and C++ programs with $100 \leq k \leq 1000$. The empirical results match the intuitive choice of k from frequencies of n-grams, which is that n-grams that appear more frequently convey less information regarding similarity. In Fig. 7, from 100 to 1000 most common n-grams the curve becomes more flat than for more common n-grams, while the main decrease in the inter-class scores are also in the same range in Fig. 8. So choosing the right k for each task-language pair can be

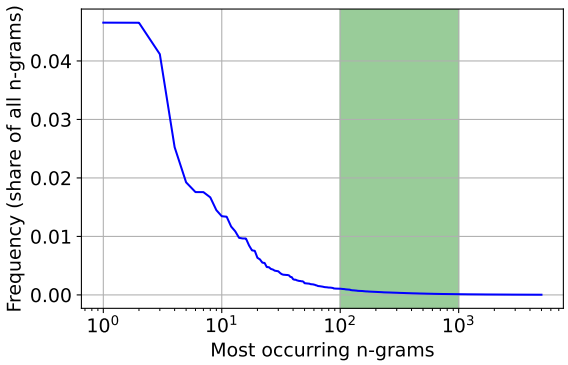

Figure 7: Frequency plot of most common n-grams in the Java language.

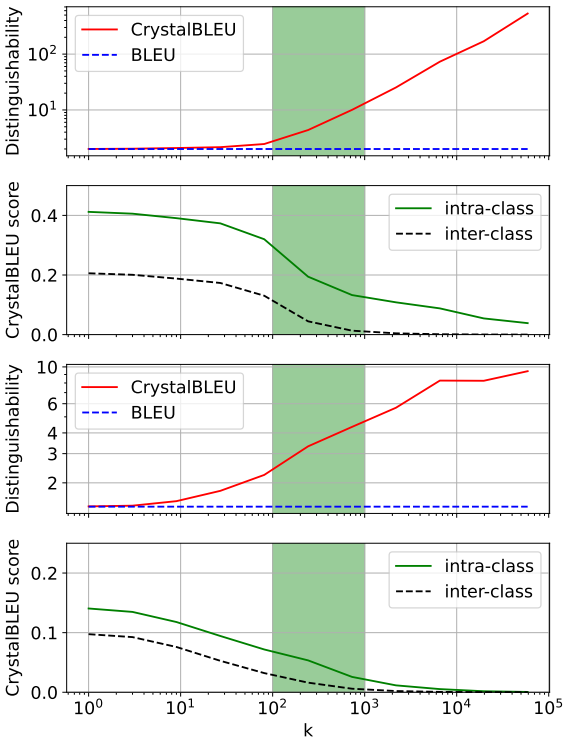

Figure 8: Distinguishability of CrystalBLEU vs BLEU based on the value of k for Java programs in the ShareCode dataset (top 2 charts) and the BigCloneBench dataset (bottom 2 charts).

done using the frequency plot of a sample corpus to find the range where the frequencies drop to low values.

Finding 4: The choice of the number k of trivially shared n-grams to consider influences the results of CrystalBLEU, but there is a relatively large range of reasonable values.

Table 6: Comparison of BLEU and alternative metrics.

Property	BLEU	CodeBLEU	RUBY	CrystalBLEU
Language-agnostic	✓	x	x	✓
Handle incomplete and partially incorrect code	✓	x	x	✓
Efficient	✓	x	x	✓
High distinguishability	x	x	N/A	✓

5 THREATS TO VALIDITY

Our analysis is limited to three datasets, and all experiments are on the Java and C++ languages. We select the languages due their popularity in the relevant literature, and the availability of data. Our datasets cover a wide range of programs in terms of domain and size. Nevertheless, our findings may not generalize to other datasets, and other programming languages may be more or less affected by the effects shown in our experiments.

6 RELATED WORK

Studies of BLEU in NLP. Since its invention in 2002 [36], BLEU has become extremely popular in natural language processing (NLP), but also received some criticism. Callison-Burch et al. [7] question how well BLEU matches the actual translation quality of machine translation systems and recommend to use it only for systems that use similar translation strategies. Reiter [40] provide a systematic review of papers that use BLEU in NLP and criticize using BLEU as the only metric, as well as using BLEU in domains other than machine translation. Post [38] discusses potential problems that may arise from different variants of BLEU, and the lack of details in many papers about which variant is used. Babych and Hartley [5] study the effects of applying weights to words based on the TF-IDF of each word when calculating BLEU. Although their approach is similar to ours, their goal is to increase the correlation of translation adequacy with human judgment. On the other hand, our approach applies to all n-grams (not just 1-grams), and the goal is to increase distinguishability. Lin and Hovy [28] studies some shortcomings of BLEU in natural language summarization and proposes an alternative. They show that the uni-gram co-occurrence correlates well with human judgement, but the BLEU score performs poorly in some cases. Graham [14] presents a detailed analysis of different variants of ROUGE and BLEU with regards to their correlation with the human judgement. One of the settings used is removing stop words, which is a special case of removing common n-grams.

Alternative Metrics for Code. Recent work also criticizes the use of BLEU for code-related tasks. One line of work [48] studies how BLEU relates to a human judgment of code similarity in the context of code translation between programming languages. The work by Tran et al. [48], and also a more recent paper by Ren et al. [41], propose alternative metrics to assess code similarity. Tran et al. [48] also study BLEU on code, focusing on a specific application domain, code migration, and on comparing BLEU against a human-created oracle of semantic similarity. Both metrics rely on parsing the code, which only works if the code to evaluate is syntactically well-formed.

To summarize the differences in features that are important for similarity metrics in this domain, Table 6 shows to what extent the existing metrics, i.e. BLEU [36], RUBY [48], and CodeBLEU [41], and our approach provide the following five desirable properties. First, an ideal metric should be language-agnostic, a property BLEU and CrystalBLEU provide by relying only on tokenization, but not on any form of deeper program analysis. Second, a metric should be able to handle incomplete and partially incorrect code, as such code may be predicted by the evaluated techniques. RUBY and CodeBLEU fail to provide this property as they must parse code into an AST. Third, the metric should be efficient enough to be applied to large code corpora or as an online metric as used in Chakraborty et al. [8], a property that is more difficult to maintain when relying on more semantic program analysis. Fourth, the metric should evaluate well with human judgment. Finally, the metric should provide high distinguishability, which CrystalBLEU does, as shown in our evaluation. Note that the implementation of RUBY was not publicly available, so we were not able to assess its distinguishability. However, the existence of other features were deduced from the paper.

There is a related approach in the previous work by Ren et al. [41], called "weighted n-gram matching", which is assigning weights to n-grams to affect the importance of some n-grams. However, in weighted n-gram matching they assign higher weights, i.e. higher focus, on keywords, and only 1-gram keywords, which are predefined for each language. These weights are higher than the normal weights for other n-grams. This type of weight assignment is the opposite of our approach, and shows worse distinguishability compared to our approach.

Other Uses of BLEU Score. Beyond the use on code discussed in this paper, BLEU has been used to evaluate natural language prediction technique in software engineering, such as code summarization [2, 53], log message generation [19], and comment generation [20]. Wan et al. [50] propose an actor and critic-style deep reinforcement learning technique to predict a comment from code, which uses BLEU as a feedback metric during learning. Our study focuses on evaluating code prediction, not on natural language.

7 CONCLUSION

This paper presents CrystalBLEU, a BLEU-based metric to evaluate source code similarity at scale, regardless of the programming language. Our metric is as fast and easy-to-use as BLEU, while considering an inherent difference between source code and natural language, namely the presence of trivially shared n-grams. We show through experiments on different languages and datasets that CrystalBLEU outperforms existing metrics in its ability to distinguish semantically equivalent from non-equivalent pairs of code. We envision CrystalBLEU to provide a more precise metric for future evaluations of techniques that predict source code. Moreover, our novel meta-metric, distinguishability, will support the development of even better code similarity metrics.

DATA AVAILABILITY

The implementation of CrystalBLEU and scripts used for experiments presented in this paper are publicly available at https://github.com/sola-st/crystalbleu.

ACKNOWLEDGMENTS

This work was supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and DeMoCo projects. We thank Amir Saboury for providing us with the ShareCode dataset.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81.

[2] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. 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*. https://openreview.net/forum?id=H1gKYo09tX

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A General Path-Based Representation for Predicting Program Properties. In *PLDI*.

[4] Gareth Ari Aye and Gail E. Kaiser. 2020. Sequence Model Design for Code Completion in the Modern IDE. *CoRR* abs/2004.05249 (2020). arXiv:2004.05249 https://arxiv.org/abs/2004.05249

[5] Bogdan Babych and Tony Hartley. 2004. Extending the BLEU MT evaluation method with frequency weightings. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*. 621–628.

[6] Patrick Bareiß, Beatriz Souza, Marcelo d'Amorim, and Michael Pradel. 2022. Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code. *CoRR* abs/2206.01335 (2022). https://doi.org/10.48550/arXiv.2206.01335 arXiv:2206.01335

[7] Chris Callison-Burch, Miles Osborne, and Philipp Koehn. 2006. Re-evaluation the Role of Bleu in Machine Translation Research. In *EACL 2006, 11st Conference of the European Chapter of the Association for Computational Linguistics, Proceedings of the Conference, April 3-7, 2006, Trento, Italy*, Diana McCarthy and Shuly Wintner (Eds.). The Association for Computer Linguistics. https://www.aclweb.org/anthology/E06-1032/

[8] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. 2018. Tree2Tree Neural Translation Model for Learning Source Code Changes. *CoRR* abs/1810.00314 (2018). arXiv:1810.00314 http://arxiv.org/abs/1810.00314

[9] Boxing Chen and Colin Cherry. 2014. A systematic comparison of smoothing techniques for sentence-level bleu. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*. 362–367.

[10] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*. 665–676.

[11] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE TSE* (2019).

[12] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=SJeqs6EFvB

[13] Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent J Hellendoorn. 2020. Patching as Translation: the Data and the Metaphor. *arXiv preprint arXiv:2008.10707* (2020).

[14] Yvette Graham. 2015. Re-evaluating automatic summarization with BLEU and 192 shades of ROUGE. In *Proceedings of the 2015 conference on empirical methods in natural language processing*. 128–137.

[15] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *FSE*. 631–642. https://doi.org/10.1145/2950290.2950334

[16] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, Satinder P. Singh and Shaul Markovitch (Eds.). AAAI Press, 1345–1351. http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603

[17] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher P. Reale, Rebecca L. Russell, Louis Y. Kim, and Sang Peter Chin. 2018. Learning to Repair Software Vulnerabilities with Generative Adversarial Networks. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*. 7944–7954. http://papers.nips.cc/paper/8018-learning-to-repair-software-vulnerabilities-with-generative-adversarial-networks

[18] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 837–847.

[19] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: Distributed Representations of Code Changes. In *ICSE*.

[20] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, Foutse Khomh, Chanchal K. Roy, and Janet Siegmund (Eds.). ACM, 200–210. https://doi.org/10.1145/3196321.3196334

[21] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping Language to Code in Programmatic Context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*. 1643–1652. https://www.aclweb.org/anthology/D18-1192/

[22] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code != big vocabulary: open-vocabulary models for source code. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1073–1085. https://doi.org/10.1145/3377811.3380342

[23] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code Prediction by Feeding Trees to Transformers. In *IEEE/ACM International Conference on Software Engineering (ICSE)*.

[24] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. https://doi.org/10.1145/3318162

[25] Jaeseong Lee, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2020. On the Naturalness of Hardware Descriptions. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 530–542.

[26] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code Completion with Neural Attention and Pointer Networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence* (Stockholm, Sweden) *(IJCAI'18)*. AAAI Press, 4159–25.

[27] Xiaochen Li, He Jiang, Yasutaka Kamei, and Xin Chen. 2018. Bridging semantic gaps between natural languages and APIs with word embedding. *IEEE Transactions on Software Engineering* (2018).

[28] Chin-Yew Lin and Eduard Hovy. 2003. Automatic evaluation of summaries using n-gram co-occurrence statistics. In *Proceedings of the 2003 human language technology conference of the North American chapter of the association for computational linguistics*. 150–157.

[29] Hui Liu, Mingzhu Shen, Jiaqi Zhu, Nan Niu, Ge Li, and Lu Zhang. 2020. Deep Learning Based Program Generation from Requirements Text: Are We There Yet? *IEEE Transactions on Software Engineering* (2020).

[30] Xinyue Liu, Xiangnan Kong, Lei Liu, and Kuorong Chiang. 2018. TreeGAN: Syntax-Aware Sequence Generation with Generative Adversarial Networks. *ArXiv e-prints* (2018). arXiv:1808.07582

[31] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: learning to repair compilation errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 925–936.

[32] Anh Tuan Nguyen, Trong Duc Nguyen, Hung Dang Phan, and Tien N. Nguyen. 2018. A Deep Neural Network Language Model with Contexts for Source Code. In *SANER*.

[33] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2015. Divide-and-conquer approach for multi-phase statistical migration for source code (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 585–596.

[34] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2022. Generating Realistic Vulnerabilities via Neural Code Editing: An Empirical Study. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[35] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 574–584.

[36] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Annual Meeting on Association for Computational Linguistics (ACL)*. 311–318.

[37] Jibesh Patra and Michael Pradel. 2021. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 906–918. https://doi.org/10.1145/3468264.3468623

[38] Matt Post. 2018. A Call for Clarity in Reporting BLEU Scores. In *Proceedings of the Third Conference on Machine Translation: Research Papers*. 186–191.

[39] Michael Pradel and Satish Chandra. 2022. Neural software analysis. *Commun. ACM* 65, 1 (2022), 86–96. https://doi.org/10.1145/3460348

[40] Ehud Reiter. 2018. A Structured Review of the Validity of BLEU. *Computational Linguistics* 44, 3 (2018), 393–401.

[41] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[42] Stefan Riezler and John T Maxwell III. 2005. On some pitfalls in automatic evaluation and significance testing for MT. In *Proceedings of the ACL workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*. 57–64.

[43] Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html

[44] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A Grammar-Based Structural CNN Decoder for Code Generation. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. 7055–7062. https://doi.org/10.1609/aaai.v33i01.33017055

[45] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 476–480.

[46] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode compose: code generation using transformer. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1433–1443. https://doi.org/10.1145/3368089.3417058

[47] Yanfei Tian, Xu Wang, Hailong Sun, Yi Zhao, Chunbo Guo, and Xudong Liu. 2018. Automatically Generating API Usage Patterns from Natural Language Queries. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 59–68.

[48] Ngoc M. Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien N. Nguyen. 2019. Does BLEU score work for code migration?. In *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*, Yann-Gaël Guéhéneuc, Foutse Khomh, and Federica Sarro (Eds.). IEEE / ACM, 165–176. https://doi.org/10.1109/ICPC.2019.00034

[49] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 25–36. https://dl.acm.org/citation.cfm?id=3339509

[50] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 397–407. https://doi.org/10.1145/3238147.3238206

[51] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 261–271.

[52] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On Learning Meaningful Assert Statements for Unit Test Cases. In *ICSE*.

[53] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based Neural Source Code Summarization. In *ICSE*.

[54] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Yanjun Pu, and Xudong Liu. 2020. Learning to Handle Exceptions. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*.