Semantically Descriptive Similarity

Oskar Hovmøller Dinesen[1 [0009−0006−2425−7849]] and Christian Gram Kalhauge[1 [0000−0003−1947−7928]]

Technical University of Denmark, Kgs. Lyngby, DK, ohodi@dtu.dk and chrg@dtu.dk

Abstract. Program equivalence is a notoriously undecidable problem, consequently, all approaches to determining whether two programs are equivalent are either imprecise or incomplete. In practical applications, developers often rely on proxy measures, such as the syntactic similarity of programs. In this paper, we formalize what it means for a similarity measure to be semantically descriptive and provide a framework for ranking measures based on their ability to differentiate between equivalent and non-equivalent program pairs. We refer to this metric as the average semantic loss of the similarity measure. We evaluate 17 established similarity measures, including ones based on edit-distance, compression, and vector embeddings. Using a subset of 450 thousand program pairs from the IBM ProjectCodeNet benchmark, we show that our semantic loss score replicates previous findings based on classifier performance. Finally, using this score, we demonstrate that compression-based measures outperform state-of-the-art vector-embedding-based approaches, such as CodeBERT and GraphCodeBERT, on unformatted code, but lag behind when the code is formatted or compiled and then decompiled. This indicates that these approaches are sensitive to the varying formatting of code.

Keywords: Program Equivalence · Similarity Measures

1 Introduction

A maintainer of a piece of software has to reason about semantic equivalence while developing. Essentially, the goal is to determine, given a change to a program, whether it is likely to be a semantic change. The current approach relies on developer intuition: a difference of a few lines of code might not have changed the semantics, while a difference of hundreds of lines makes it much more likely. Essentially, the developer reasons about *semantic* differences based on *syntactic* changes. This intuition stems from the fact that syntactically equivalent programs are semantically equivalent and is, through experience, extrapolated to the much more informal statement:

Syntactically similar programs are semantically similar.

Current approaches to determining program similarity use edit-distance [22], compressed distance [18, 17], and vector embeddings [2, 7, 5]. These methods essentially return a value between 0 and 1, reflecting the similarity between two

programs. While all of these approaches have been evaluated for their ability to classify equivalent programs or identify code clones (i.e., copies of code within the same program), it remains largely unexplored how well these syntactic similarities capture semantic similarity.

In this paper, we introduce the concept of *semantically descriptive similarity* and evaluate the descriptiveness of various known similarity measures. We define a similarity measure as semantically descriptive if it helps a developer reason about the probability of changes to a program's semantics. Essentially, if two programs are less similar than another pair, the probability that the former pair is equivalent should be lower than that of the latter. Concretely, we

- formalize the definition of semantically descriptive similarity measures and create a framework for evaluating them. We capture this via a single metric, *the average semantic loss*, which can be used to rank similarity measures by their semantic descriptiveness (Section 2).
- present 17 similarity measures within our framework (Section 3), including measures based on edit-distance, compression, and vector embeddings.
- demonstrate, in a preliminary evaluation, on a subset of 450.000 program pairs from the IBM ProjectCodeNet [20], that our score can replicate previous findings [22] based on classification (Section 4). Furthermore, we see that compression-based similarities have comparable performance to more computationally expensive embedding-based similarities. We also see that CodeBERT and GraphCodeBERT benefit dramatically from code-normalization.

Finally, we discuss related work in Section 5 and conclude in Section 6.

2 Semantically Descriptive Similarity

In this section, we formalize descriptive similarity measures and motivate the average semantic loss as a method for ranking these measures according to how well they capture information about the underlying semantics.

2.1 Approximate Program Equivalence

A developer often reasons about equivalence of programs. For example, (1) "Is this code change large enough that I should run the test-suite?", (2) "Is it safe to upgrade to a newer version of this library?", and (3) "Is this code something we already have implemented?". If we can determine if two programs are the same, up to some notion of equivalence, we better answer these questions. (1) If the code change is semantically different, we should probably re-run the test-suite; (2) If the library has the same semantic meaning, we can upgrade without fear; and (3) If the newly added code is semantically different from anything we have written before it is safe to add.

With enough effort, a developer can figure out if two programs are equivalent. However, since the general problem of program equivalence is undecidable, there

(a) A program that doubles the input.

```java
import java.util.Scanner;

public class Main {
  public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);
    int a = scan.nextInt();
    int b = a * 2;
    System.out.println(b);
  }
}
```

(b) also doubles an input

```java
import java.util.*;

public class Main {
  public static void main(String[]
      args) {
    Scanner scan = new Scanner(
        System.in);
    int input = scan.nextInt();
    int doubled = input * 2;
    System.out.println(doubled);
  }
}
```

(c) prints "Hello, World!"

```java
public class Main {
  public static void main(String[]
      args) {
    System.out.println("Hello, 
        World!");
  }
}
```

Fig. 1: Three Java programs, two that doubles and input an one that prints "Hello, World!". Can we detect that (a, b) are equivalent and (a, c) are not by looking at how similar they are?

exists no automatic solution. To be efficient, a developer instead answers these questions using their built-up knowledge, intuition, and hunches. We can view this as a probabilistic problem: given prior knowledge of the distribution of program pairs as well as an equivalence relation on these pairs, can we build a classifier which, given a new program pair, can predict equivalence?

Notation When working with probabilistic problems we always have to ask "from which distribution are we sampling from?" In this paper we denote the distribution Δ and sampling a pair from the distribution we denote $(A, B) \sim \Delta$. We use $\mathbf{P}$ to denote the set of all programs. We use $p, q \in \mathbf{P}$ to denote programs, unless they have been drawn from a distribution, in which case we call them stochastic variables and denote them A, B.

2.2 Similarity Measures and Threshold Classification

In Figure 1, we see three different programs. We can make several pairs out of these three programs. We can have the pair (a,b) which is an equivalent pair and (a,c) which is non-equivalent. A developer can relatively easily see the differences and similarities, but doing the same automatically and at scale is difficult.

In practice, to capture and aide the intuition of the developer, the software engineering community have developed numerous similarity measures [16]. Conceptually, a similarity measure is a mapping from program pairs to a number between 0 and 1. The goal of the similarity measure is to enable the developer to build a simple threshold classifier, to solve the problems mentioned in the previous section. Despite minor variations, these measures generally share a common theoretical foundation:

Definition 1 (Similarity Measure). *A similarity measure is a function* $\widetilde{t} : \boldsymbol{P} \times \boldsymbol{P} \to [0, 1]$ *that is positive definite* $\widetilde{t}(p, p) = 1$. *We will use* $\widetilde{f}$ *to indicate that a function* f *is similarity measure.*

It should be noted that most classic distance functions, like Euclidean-, Hamming-[8], Mahalanobis[1]-distance, can be turned into a similarity measure through some form of normalization. Instead of computing similarities directly from the programs, tools calculate the *dissimilarity* between them. We can treat *dissimilarity* as a pseudo-distance between programs that does not require the triangle inequality to hold.

Threshold Classification As mentioned above, the goal of the similarity measure is to enable the developer to setup a simple threshold classifier. A threshold classifier simply labels every program pair above a threshold k as equivalent, and everything below as different. To further motivate this, we can see this in action on the test-suite problem. Given a sample set of previous code changes $\Delta_K = \{(p_1, q_1), (p_2, q_2), \ldots\}$ where $(p_1, q_1), (p_2, q_2), \ldots \sim \Delta$, which we have run our test suite on, i.e., we know whether $p_i \equiv q_i$ or $p_i \not\equiv q_i$ for all pairs in Δ_K. It would be nice if a program change similar to previous accepted changes will also be accepted, or vice versa. Intuitively, the developer sets a threshold, and would like to know that the probability of failing a test is larger below the threshold, and lower above.

Example (diff) To provide a concrete idea of a similarity measure, we investigate edit-distance over lines, denoted $\widetilde{\text{diff}}$. To calculate the edit-distance, the tool first breaks down a program into a sequence of items (e.g., characters, tokens, or lines), represented by an embedding function f. The tool then counts the operations required to transform one sequence of items into another, typically including insertion, deletion, and substitution. The dissimilarity measure diff over two programs p and q, given an itemization function it, is defined as follows:

$$\text{diff}\langle \text{f} \rangle(p, q) = \|f(p), f(q)\|_{\text{edit}}$$

To calculate the similarity based on diff, Krinke et al. [22] propose the following:

$$\widetilde{\text{diff}}\langle \text{f} \rangle(p, q) = 1 - \frac{\min\left(|f(q)|, \text{diff}\langle \text{f} \rangle(p, q)\right)}{|f(q)|}$$

In the case of the programs in fig. 1, we see that pair (a,b) differs in lines 1 and 6-8. We get that:

$$\widetilde{\text{diff}}\langle\text{lines}\rangle(a,b) = 1 - \frac{\min(10,4)}{10} = 0.6 \qquad \widetilde{\text{diff}}\langle\text{lines}\rangle(a,c) = 1 - \frac{\min(5,6)}{5} = 0$$

Similarly using characters instead of lines we get: $\widetilde{\text{diff}}\langle\text{chars}\rangle(a,b) = 0.875$ and $\widetilde{\text{diff}}\langle\text{chars}\rangle(a,c) = 0$. As such both similarity measures manage to capture the fact that program (a) is more similar to program (b) than to program (c). In isolation, it is unclear which of the two capture the semantics best. While it is clear that the similarity measure in general should order semantically equivalent programs higher than non-equivalent, the actual values are largely uninteresting. For example, a threshold of 0.5 would work equally well in separating the pairs in the above toy example.

2.3 Semantic Ordering

We claim that a similarity measure is semantically descriptive if it can separate semantically equivalent program pairs from non-equivalent using a simple single threshold classifier. For this to be the case we need to ensure two things. Firstly, that a single threshold classifier is actually a good choice of classifier. Secondly, that the classifier performs well. To address the first property we give the following definition:

Definition 2 (Semantically Ordered Similarity). *We say that the similarity measure $\widetilde{t}$ is semantically ordered over a program pair distribution Δ, if*

$$k_1 \leq k_2 \quad \Longrightarrow \quad \Pr\left(A \equiv B \mid \widetilde{t}(A,B) \geq k_1\right) \leq \Pr\left(A \equiv B \mid \widetilde{t}(A,B) \geq k_2\right).$$

Essentially, this means that a simple threshold classifier will be a good choice of classifier. In many cases, when a feature is not semantically ordered, better performance can be achieved by setting multiple cutoffs, or varying the cutoff depending on an observation like in kNN classifiers. If a similarity measure is semantically ordered a single threshold classifier should be a good choice.

Note that the choice of semantic equivalence $\equiv$ is arbitrary, and we could use our framework to investigate whether a similarity measure captures any equivalence relation.

To provide some intuition for our definition, we investigate the following two idealized scenarios. The ideal similarity measure $\widetilde{\equiv}$, which is defined as the indicator function that returns 1 for all equivalent pairs and 0 otherwise. As well as the, the random similarity measure $\widetilde{R}(p,q) \in [0,1]$, is defined as a random assignment of values from the unit interval to each program pair.

$$\widetilde{\equiv}(p,q) = \begin{cases} 1 & p \equiv q \\ 0 & \text{otherwise} \end{cases} \qquad \widetilde{R}(p,q) \in [0,1]$$

The ideal similarity measure is clearly semantically ordered, but the same is the case for the random similarity.

Cross Entropy The fact that the ideal- and the random similarity measure both are semantically ordered exposes the fact that ordering is not enough to guarantee that a threshold classifier actually has good performance with respect to precision and recall. As such, we have to impose yet another requirement on our similarity measure, namely that the ratio should be steep.

An naive approach to assess the descriptiveness of similarity measures is to simply compute the cross-entropy over a set of program pairs Δ_K, with respect to the ideal measure $\widetilde{\equiv}$:

$$\mathcal{L}(\Delta_K, \widetilde{t}) = -\sum_{(p,q)\in\Delta_K} \widetilde{\equiv}(p,q) \cdot \log(\widetilde{t}(p,q)) + (1 - \widetilde{\equiv}(p,q)) \cdot \log(1 - \widetilde{t}(p,q))$$

This does not measure the ordering, but rather the separation between classes. Intuitively, the loss will be smaller if equivalent pairs are more similar, and non-equivalent pairs are less similar. This yields a score for assessing how calibrated a similarity measure is relative to the ideal measure.

The problem with this approach is that two similarity measures might produce very different outputs. In the example of diff, we saw that the outputs of $\mathrm{diff}\langle\mathrm{chars}\rangle$ had values in $[0, 0.875]$, whereas $\mathrm{diff}\langle\mathrm{lines}\rangle$ had values in $[0, 0.6]$. In that particular case, it is the log-loss of $\mathrm{diff}\langle\mathrm{chars}\rangle$ is lower than that of $\mathrm{diff}\langle\mathrm{lines}\rangle$, even though there exists equally good classifiers for both. Since the threshold classifier does not care about the absolute thresholds, we cannot compute the cross-entropy directly. We first have to calibrate.

Calibration & Isotonic Regression There are several ways to calibrate our similarity measure, but since we are interested in the way a similarity measure orders our program pairs, we choose to use isotonic regression [26].

Concretely, to calculate $\mathrm{Iso}^{\widetilde{t}}_{\Delta_K}(\widetilde{t}(p_i, q_i)) = y_i$ we solve the following minimization problem:

$$\min \sum_{(p_i,q_i)\in\Delta_K} (\widetilde{\equiv}(p_i, q_i) - y_i)^2 \quad \text{where} \quad \widetilde{t}(p_i, q_i) \leq \widetilde{t}(p_j, q_j) \implies y_i \leq y_j$$

The constraint is essential for us, since it preserves the ordering imposed by the similarity measure, while enforcing that the calibrated similarity measure is semantically ordered. Being semantically ordered is a property of the underlying distribution of similarities. We can empirically model this posterior distribution with samples. Due to sampling error, this will never satisfy the semantically ordered property, even with many samples. By using isotonic regression, we project the similarities into the closest monotone probability function. If the regression has to make only slight corrections to the posterior distribution modeled by the similarity measure, we can say that the similarity measure is consistent with the semantically ordered property. On the contrary, if the changes imposed by the regression is large, the similarity measure is not consistent with the definition. The isotonic regression $\mathrm{Iso}^{\widetilde{t}}_{\Delta_K}(\widetilde{t}(p_i, q_i)) = k$ is an empirical approximation of $\Pr\big(A \equiv B \mid \widetilde{t}(A, B) = k\big)$, then as it holds that $\widetilde{t}(p_i, q_i) \leq \widetilde{t}(p_j, q_j) \implies y_i \leq y_j$

it implies that isotonic regression enforces that increasing k increases the amount of equivalent pairs with respect to the chosen subset:

$$\frac{\#\left\{(p,q) \in \Delta_K \mid p \equiv q \wedge \mathrm{Iso}^{\widetilde{t}}_{\Delta_K}(p,q) \geq k\right\}}{\#\left\{(p,q) \in \Delta_K \mid \mathrm{Iso}^{\widetilde{t}}_{\Delta_K}(p,q) \geq k\right\}}$$

This can only happen by excluding non-equivalent pairs, and as such we can infer that the ratio between equivalent and non-equivalent pairs increases with k, i.e. it is monotone and non-decreasing. As such, any isotonically calibrated similarity measure reveales whether a single threshold classifier captures all semantic information. At the same time, the calibration mitigates the distortion across various similarity measures. With respect to our example, 0.875 will be mapped to 1 for $\widetilde{\mathrm{diff}}\langle\mathrm{chars}\rangle(a,b)$, the isotonic regression will also map 0.6 to 1 for $\widetilde{\mathrm{diff}}\langle\mathrm{lines}\rangle$. Thus, scoring both similarity measures as equally descriptive, at least with respect to our toy example. As we will see later on, there are situation where these measures makes a wrong ordering with respect to the semantics.

The Average Semantic Loss This brings us to our framework for measuring the semantic descriptiveness of a similarity measure.

Definition 3 (Average Semantic Loss). *Given a set of program pairs Δ_K and a similarity measure $\widetilde{t}$, the average semantic loss is the average cross-entropy of the iso-calibrated $\widetilde{t}$ relative to $\widetilde{\equiv}$ over Δ_K:*

$$\mathcal{L}_{\equiv}(\Delta_K, \widetilde{t}) := \frac{\mathcal{L}(\Delta_K, \mathrm{Iso}^{\widetilde{t}}_{\Delta_K} \circ \widetilde{t})}{|\Delta_K|}$$

This measure, captures both the semantic ordering, as well as the steepness of the ordering ratio. We end up with a single scalar value, which provides insight into the average performance of a similarity measure when used for single threshold classification. This bring us to our definition of what it means for a similarity to be semantically descriptive:

Definition 4 (Semantically Descriptive Similarity). *A similarity measure $\widetilde{t}$ is semantically descriptive over a program pair distribution Δ iff*

$$\mathcal{L}_{\equiv}(\Delta, \widetilde{t}) < \mathcal{L}_{\equiv}(\Delta, \widetilde{\mathrm{R}}).$$

If we have two similarity measures, $\widetilde{t}_1$ and $\widetilde{t}_2$, we say that $\widetilde{t}_1$ is more semantically descriptive than $\widetilde{t}_2$ if $\mathcal{L}_{\equiv}(\Delta, \widetilde{t}_1) < \mathcal{L}_{\equiv}(\Delta, \widetilde{t}_2)$. As such, we have arrived at a way of ranking similarity measures based on their applicability for single threshold classification.

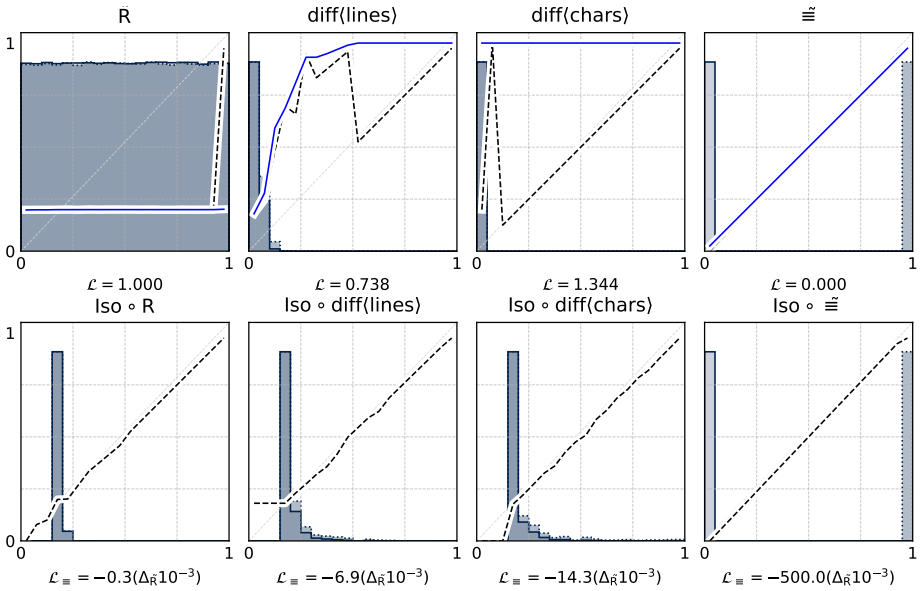

Fig. 2: The similarities $\widetilde{\text{diff}}\langle\text{lines}\rangle$, $\widetilde{\text{diff}}\langle\text{char}\rangle$, $\widetilde{\text{R}}$, and $\widetilde{\equiv}$. Each diagram shows the distribution of equivalent (dotted bins) and non-equivalent (solid bins) programs over their similarity scores. We scaled the distributions to fit the graph. The x-axis represents the similarity measure, and the y-axis (dashed black line) reflects the proportion of program pairs with that similarity measure that are equivalent. The solid blue line in the first row represents the isotonic regression, the second row shows the similarity scores calibrated using this regression. $\mathcal{L}$ denotes the average loss over the similarity.

Example Figure 2 shows the calibration results for the two diff similarity measures, a random $\widetilde{\text{R}}$ measure, and a simulated ideal $\widetilde{\equiv}$ measure. The $\widetilde{\equiv}$ measure provides clear separation between equivalent and non-equivalent pairs. The top row displays the distributions of non-equivalent (solid line, gray area) and equivalent (dotted line, gray area) program pairs across these four measures. The results show average cross-entropy L values of 1.0 for $\widetilde{\text{R}}$, 0.8 for $\widetilde{\text{diff}}\langle\text{lines}\rangle$, 1.4 for $\widetilde{\text{diff}}\langle\text{char}\rangle$, and 0.0 for $\widetilde{\equiv}$. Notably, $\widetilde{\text{diff}}\langle\text{char}\rangle$ is less calibrated than the random baseline. In the bottom row, the average cross-entropy is computed on the isotonic calibration, thus yielding the average semantic loss $\mathcal{L}_{\equiv}$. To better illustrate the effect, we subtract the *random* baseline of 0.5 and scale the difference by 10^{-3}. We denote this unit as $(\Delta_{\text{R}} 10^{-3})$. As expected, the random baseline scores very close to 0. The $\mathcal{L}_{\equiv}$ values for $\widetilde{\text{diff}}\langle\text{lines}\rangle$ (-4.3) and $\widetilde{\text{diff}}\langle\text{char}\rangle$ (-10.3) are significantly more informative, revealing that lines are less descriptive than characters. However, a large gap remains relative to the ideal $\widetilde{\equiv}$ score of -500.0.

In conclusion, we have presented a framework to evaluate the semantic descriptiveness of a similarity measure by calculating the cross-entropy of an iso-

$$\widetilde{\mathrm{diff}}\langle \mathrm{it}\rangle(p,q) = 1 - \frac{\min\{|\mathrm{it}(q)|, \mathrm{diff}\langle \mathrm{it}\rangle(p,q)\}}{|\mathrm{it}(q)|} \qquad \mathrm{diff}\langle \mathrm{it}\rangle(p,q) = \|\mathrm{it}(p), \mathrm{it}(q)\|_{\mathrm{edit}}$$

(a) The edit-distance similarity measures, where $\mathrm{it} \in \{\mathrm{lines}, \mathrm{char}\}$.

$$\mathrm{ncd}\langle \mathrm{Z}, \mathrm{n}\rangle(p,q) = \frac{Z^n(p \cdot q) - \min\{Z^n(p), Z^n(q)\}}{\max\{Z^n(p), Z^n(q)\}} \qquad \mathrm{cdm}\langle \mathrm{Z}, \mathrm{n}\rangle(p,q) = \frac{Z^n(p \cdot q)}{Z^n(p) + Z^n(q)}$$
$$\widetilde{\mathrm{ncd}}\langle \mathrm{Z}, \mathrm{n}\rangle(p,q) = 1 - \mathrm{ncd}\langle \mathrm{Z}, \mathrm{n}\rangle(p,q) \qquad \widetilde{\mathrm{cdm}}\langle \mathrm{Z}, \mathrm{n}\rangle(p,q) = 1 - \mathrm{cdm}\langle \mathrm{Z}, \mathrm{n}\rangle(p,q)$$

(b) The compression based similarity measures, given binary length of compressor Z at compression level n: $Z^n : P \to \mathbb{N}$. $p \cdot q$ is the source concatenation of p and q.

$$\widetilde{\cos}\langle \mathrm{B}\rangle(p,q) = \frac{1 + \cos(\mathrm{emb}_{\mathrm{B}}(p), \mathrm{emb}_B(q))}{2}$$

(c) The cosine similarity, given vector embedding function $\mathrm{emb}_B : P \to \mathbb{R}^n$, for some dimensionality n.

Fig. 3: The similarities investigated in this paper.

tonic regression fit. Our score provides a direct measure of a similarity measure's ability to separate semantically equivalent from semantically different pairs using a single threshold classifier.

3 Similarity Measures

We now introduce the similarity measures compared in this paper. Each subsection of this section introduces a distinct group of similarities. The resulting measures are listed in fig. 3. Since we previously described the edit-distance, we omit its description here but provide the derived similarity measures in fig. 3a.

3.1 Compression-based Similarities

Besides edit-distance, another well-studied similarity measure comes from the compressed distance [18]. This measure stems from the idea that a program encodes information. We argue that two programs are similar if the information they encode overlaps significantly. Similarly, two programs are dissimilar if the intersection of their encoded information is small. To measure the informational distance between two programs, we can consider the shortest sequence of bit-measured actions interpretable by an "editor" that transforms p into q. This measure is also known as the Kolmogorov distance [18] and is denoted by $K(p\,q)$. The chain rule states that the informational complexity of the concatenation of p and q approximates the sum of the complexity of p and the conditional complexity $K(p \mid q)$, up to an additive constant c, such that $K(p \cdot q) \simeq K(q) + K(p \mid q)$.

Consequently, if we know the informational complexities of p and $p \cdot q$, we can estimate the conditional complexity $K(p \mid q)$ as $K(p \mid q) \simeq K(p \cdot q) - K(p)$. Computing the informational complexity of a program is undecidable, but compression algorithms offer a workable approximation. Consequently, Li and Vitányi [18] defined the *normalized compressed distance* ncd, which is presented in Figure 3b. Subtracting the normalized compressed distance from 1 converts it into a similarity measure. Figure 3b lists the definition of the $\widetilde{\text{ncd}}$ similarity. Alternatively, we use the *compressed dissimilarity measure* [13], denoted by $\widetilde{\text{cdm}}$, its definition is also provided in fig. 3b. We include this, since it has been succesfully used as a simplified version of the ncd [9], but produces similarities in the range $[0.5, 1]$.

3.2 Vector Embedding-based Similarities

Another approach to measuring similarity between programs is to embed them as vectors. Modern AI tools, such as CodeBERT [5], GraphCodeBERT [7] and Code2Vec [2], employ this method. Specifically, they function as an embedding $\text{emb} : P \to \mathbb{R}^n$. Code2Vec directly produces a single vector as an embedding. The BERT models, however, are transformer-based models, Transformer-based embedding models output a sequence of token-level embeddings. To get a single vector embedding from these token-level embeddings, we need to pool them. There are different pooling strategies, where CLS and Mean Pooling are the most common. For our evaluation, we choose mean pooling because this method is generally considered more robust and effective for semantic similarity tasks. [10] We quantify the similarity between vectors using the cosine similarity, which is a widely adopted metric for assessing semantic similarity.

Definition 5 (Cosine Similarity). *Given vectors $\overrightarrow{x}, \overrightarrow{y} \in \mathbb{R}^n$, the cosine similarity* cos *is calculated as*

$$\cos(\overrightarrow{x}, \overrightarrow{y}) = \frac{\overrightarrow{x} \cdot \overrightarrow{y}}{|\overrightarrow{x}||\overrightarrow{y}|},$$

where $\cdot$ is the inner product of the two vectors.

The cosine similarity maps to the interval $[-1, 1]$, where 1 indicates similarity and -1 indicates dissimilarity. As this mapping conflicts with our definition of a similarity measure, we transform it according to the definition in fig. 3c.

4 Preliminary Evaluation

In this section, we provide a preliminary evaluation of our framework, by comparing findings by Krinke et al. [22] based on classification, with our semantic loss score. We do this by calculating and comparing 17 different similarity measures (table 1) on 450 thousand program pairs from IBM's ProjectCodeNet [20]. We first investigate to what extent the described similarity measures are semantically descriptive. Then, we show that code normalization greatly improves the

performance of similarity measures. Finally, we validate the average semantic loss directly demonstrating its correlation with the similarity's performance as a problem classifier. In summary, we conducted three experiments to address these research questions:

- **RQ1** How descriptive are similarity measures?
- **RQ2** Does code-normalization improve semantic descriptiveness?
- **RQ3** Is semantic descriptiveness related to kNN classification performance?

Our findings are preliminary because we have only tested our experiments on a subset of 5 out of 250 program classes. We deliberately limited the scope to avoid contaminating the data set while generating initial hypotheses about the domain. The following section details the experimental setup. We then address the research questions.

4.1 Experimental Setup

To evaluate different similarity measures over the above described Java programs, we have created a benchmark suite, which evaluates the average semantic loss across all the similarity measures described. It is available at Zenodo [3].

Benchmarks. The benchmark suite utilizes the Java250 dataset, which is part of IBM's ProjectCodeNet [20]. The dataset contains 250 problems, each with approximately 300 user-written solutions that passed the respective problem's test suite. Each of the solutions is written as a single Java class and contains roughly between one and six methods. In this evaluation, we consider programs belonging to the same problem class to be equivalent, and programs that are solutions to different problems to be non-equivalent.

Tools. We wrote the benchmark in Python and utilized Rust for performance-critical components. We instantiate edit-distance based similarity measures from Python's built-in `difflib` using its official Python 3 bindings. For the ncd and cdm similarity measures, we instantiate them using the Rust bindings for three established compressors: ZSTD, GZip, and Zlib. These compressors support various configurations, with the compression rate proving to be the most significant parameter. To evaluate cosine similarity, we use embeddings generated by CodeBert [5], GraphCodeBert [7], and Code2Vec [2].

We employ several dedicated tools for code normalization. The `javac` compiler handles the compilation step. We utilize Google's Java Formatter [6] for formatting and the Procyon Java Decompiler [24] for decompilation. We also incorporate a tokenizer which places each token on a separate line. The tokenization includes the comments.

4.2 RQ1: How descriptive are similarity measures?

In Table 1, we report the results for the 17 most interesting similarities. For the compression-based tools, we selected compression levels 1 and 9. Level 1 is

Table 1: Evaluation results for similarity measures on original programs. From left to right, the columns denote, the name of the similarity measure, the average time to compute one similarity, the average raw cross-entropy of the similarity measure, and finally the average semantic loss relative to random.

Similarity Measure	Time (ms)	$\mathcal{L}$	$\mathcal{L}_{\equiv}(\Delta_{\mathrm{R}}10^{-3})$
diff⟨lines⟩	0.02	−3.80	−1.90
diff⟨chars⟩	0.62	−4.10	−11.39
ncd⟨gzip, 1⟩	0.08	0.46	−12.42
ncd⟨gzip, 9⟩	0.09	0.44	−15.74
ncd⟨zstd, 1⟩	0.02	0.47	−14.94
ncd⟨zstd, 9⟩	0.10	0.47	−16.74
ncd⟨zlib, 1⟩	0.08	0.47	−12.90
ncd⟨zlib, 9⟩	0.12	0.45	−16.34
cdm⟨gzip, 1⟩	0.08	0.49	−12.97
cdm⟨gzip, 9⟩	0.09	0.50	−17.27
cdm⟨zstd, 1⟩	0.02	0.50	−15.78
cdm⟨zstd, 9⟩	0.10	0.50	−17.17
cdm⟨zlib, 1⟩	0.08	0.48	−13.42
cdm⟨zlib, 9⟩	0.12	0.50	−17.84
cos⟨CodeBERT⟩	942.05	−2.46	−12.09
cos⟨GraphCodeBERT⟩	721.81	−1.71	−12.49
cos⟨Code2Vec⟩	356.92	−0.29	−3.22

the fastest, and we observed no notable improvement in performance beyond level 9. We grouped the similarities by type and highlighted the best results in each column. We present the average semantic loss as a distance from a random baseline. Thus, a negative loss indicates superior performance.

Analysis. cdm instantiated from the zlib compressor with a compression rate of 9, as shown in table 1, provides the most semantically descriptive similarity measure. Increasing the compression rate reduces the loss for all compressors. This observation aligns with the theory [18] and previous findings [22].

The raw cross-entropy suggests that compression-based methods perform no better than random sampling, whereas the remaining methods perform slightly better. However, calibration reveals that compression-based methods generally capture the best semantic information while also being fast. Notably, embedding-based methods are four orders of magnitude slower than compression while only slight outperforming character-level differencing.

Conclusion We find that we can reproduce the findings from Krinke et al. [22], that compression based similarity measures improve with the compression levels. We also see that the compression based approaches in general outperform the vector-embeddings, while being four orders of magnitude faster.

Table 2: The Average Semantic Loss of the different types of normalization for each similarity measure. The values are reported in ($\Delta_R 10^{-3}$) relative to random, as in the table 1. The columns are the names of the normalizations.

Similarity Measure	Original	Formatted	Tokens	Round-Tripped
diff⟨lines⟩	−1.90	−8.12	−33.04	−39.27
diff⟨chars⟩	−11.39	−16.20	−12.96	−37.04
ncd⟨zstd, 9⟩	−16.74	−25.94	−26.85	−48.47
cdm⟨zlib, 9⟩	−17.84	−28.05	−29.28	−52.86
cos⟨CodeBERT⟩	−12.09	−34.48	−26.50	−54.19
cos⟨GraphCodeBERT⟩	−12.49	−54.52	−69.03	−68.61
cos⟨Code2Vec⟩	−3.22	−3.22	−3.22	−28.69

4.3 RQ2: Does code-normalization improve semantic descriptiveness?

Table 2 shows how different code-normalization steps improve the results. The *original* represents the code exactly as written by the developers. The *formatted* represents the code after formatting with the Google Java formatter [6]. The *tokens* representation lists each token on a separate line. The *round-tripped* code is the result of compiling the original code with javac and subsequently decompiling it using the Procyon Java Decompiler.

Analysis. Both formatting and tokenization of programs improve the performance of all tools except Code2Vec. This difference is logical for diff- and compression-based tools, as differences in whitespace, for example, result in insertions and deletions for diff-based tools, which do not carry semantic meaning. Similarly, formatting the programs reduces entropy due to syntactic structure, which should improve the performance of compression-based methods.

Since Code2Vec embeddings are based on extracting an Abstract Syntax Tree (AST), formatting plays no role in its process, this is reflected in the results. This assumption does not hold for BERT methods. The lower semantic loss observed when using tokenized programs for CodeBERT is likely attributable to the bimodal architecture of BERT models, as they are trained on code as well as comments, and documentation. Consequently, these models also leverage naming conventions and comments to achieve better results. Because round-tripping removes all comments and gives generic names to variables, it is outperformed by the tokenized programs which retain the information.

Compiling and decompiling greatly improves all similarity measures. This happens because enabling compiler optimizations reduces structural differences by choosing the constructs such as `for`-loops over `while`-loops.

Interestingly, after any formatting, GraphCodeBERT performs as well as after code round-tripping. This observation suggests two key points. First, GraphCodeBERT (and CodeBERT) appears to be more negatively impacted by whites-

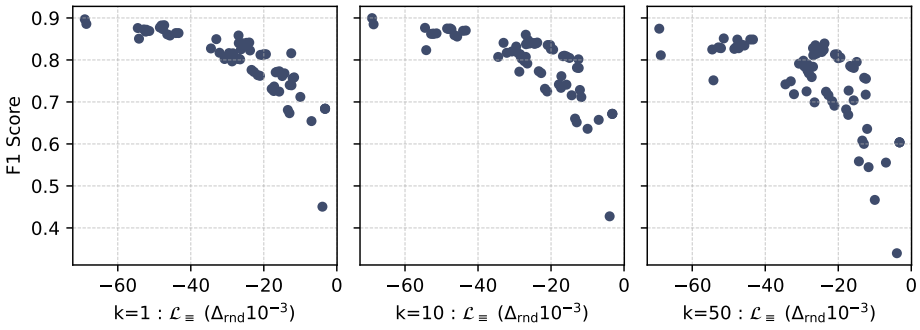

Fig. 4: A scatter plot over the semantic descriptiveness vs F1 score in k-nearest neighbors, k=1, k=10 and k=50. The x-axis shows the milli- (10^{-3}) average semantic loss relative to random. The y-axis is the F1 score of the classifier.

pace characters than it benefits from them. Second, GraphCodeBERT seems to have captured the semantic information encoded by the compiler.

Conclusion We replicate the findings of previous work by Krinke et al. [22, 21], demonstrating that increased code normalization enhances the similarity between semantically equivalent programs. Although compression-based measures outperform embedding-based measures in unformatted code, this relationship reverses as the code becomes more normalized. An essential insight is that BERT-based tools benefit significantly from whitespace removal.

4.4 RQ3: Is semantic descriptiveness related to classification?

As described in section 2.3 semantic ordering should yield a good threshold classifier. Since the similarity measures are not intrinsically semantically ordered, we want to investigate if more semantically descriptive similarity measures imply good performance when using the similarity measures to construct other classifiers. Therefore, we built k-Nearest Neighbor (kNN) classifiers [23] using each similarity measure with $k \in \{1, 10, 50\}$. Specifically, given any program and its k most similar neighbors, we select the problem class that is most representative. We run this classifier across all normalized versions of our benchmark suite, and compute the F1 score. We select the F1 score as it remains a standard metric for classifier evaluation.

Analysis We plot the F1 scores against the average semantic loss in Figure 4, for the kNN on $k \in \{1, 10, 50\}$. A trend suggests that a similarity measure with low average semantic loss yields a high F1 score. This correlation is strongest for $k = 1$. This relationship is intuitive: a semantically descriptive similarity measure ensures that the nearest neighbor of a program p is the most likely to be equivalent to p. For $k \in \{10, 50\}$, the trend remains evident, although uncertainty increases.

Conclusion We demonstrate that the semantic descriptiveness of a similarity measure correlates with its effectiveness as a kNN classifier. However, the advantage of the average semantic loss is twofold, it is: a quantitative measure of how good a single threshold classifier works on the similarities, computationally more efficient than kNN, as we sample program pairs from the set of programs rather than computing the $O(n|P|)$ comparisons required to classify n programs from a set P.

4.5 Threats to Validity

Our primary threat to validity stems from the fact that the dataset consists of relatively small programs. Consequently, the results may not generalize to larger programs. Furthermore some of the normalizations may change the semantics of the programs. Because we utilize off-the-shelf tools, we cannot guarantee the soundness of these tools. However, we tested that all normalized programs compile, which assures us that the normalizations produce valid Java code. Secondly, we only use pairs of equivalent programs and cross compare them. As such it is possible that small destructive pertubations to some of these equivalent pairs, may reveal some deficiencies in this methodology, as a small destructive change may change the semantics a lot, while not changing the similarity significantly.

5 Related Work

We have identified two kinds of related work, the first being other approaches of detecting semantic equivalence. Secondly, work on benchmark suites for reasoning on program semantics.

Other Approaches to Detecting Semantic Equivalence. Our work is also closely related to work on code clone detection. The idea of normalizing using decompilation stems from Ragkhitwetsagul and Krinke [21]. Following this idea, they investigate different normalization techniques and how they improve clone detection in an extensive comparison of code clone tools by Ragkhitwetsagul, Krinke and Clark [22]. There is a significant overlap of tools in their survey and ours, but they do not include comparisons with vector embedding methods, instead they include dedicated code clone detection tools like CCFX [12] and Deckard [11]. Our work also differs in the evaluation approach. Their work focuses on discrete evaluation methods and how well the tools are at finding clones. Instead, we focus on a more continuous evaluation approach using similarity measures. This provides a good common base for comparing across different types of program analysis, and requires less configuration and tuning. Another important different is that the clones in their benchmark are created using static techniques and obfuscators from a small pool of programs. In the code clone literature clones are referred to as type I-IV clones, which are clones created by reformatting (I), renaming (II), and restructuring (III) programs. The IV clones, are programs where only guarantee is that they are semantically equivalent to the program

they clone. Our programs are exactly of this type, since all programs are written by humans and come in many different implementations. As such, our data set contains more advanced types of code clones.

Other Benchmarks and Evaluation Approaches. In recent years there has been a large increase in the number of benchmark suites for evaluating how well AI tools can assess semantics of code. Among recent ones are EquiBench [25] and work by Laneve et al [15]. These benchmark studies are similar to ours in the sense that they evaluate the performance of AI tools against a labeled data set. These data sets, like the code clone survey [22], are synthesized by using different compiler and obfuscation techniques. This seems to be one of the most prevalent approaches when creating benchmarks for evaluating LLMs and other AI methods [19, 4]. They are rarely concerned with how they compare to non-AI tools and as such we believe that many of these data sets can be utilized in our framework, to compare AI tools with other semantic reasoning tools. These works are also based of a few specific types of programs solving specific tasks like computing the fibonacci sequence or sorting an array. Work by Bacchelli et al have found that the performance of AI tools drop, when they are exposed to programs they have not seen before. [14]. As such, our work is more in this vain, as the problems solved are varied, and not of a very specific kind.

6 Conclusion

In this paper, we introduce semantically descriptive similarity measures and a framework for scoring their descriptiveness. We refer to this score as the average semantic loss. Preliminary evaluations of our semantic loss on 17 different similarity measures, indicate that it captures how effectively the similarity measure separates equivalent program pairs from non-equivalent pairs. We demonstrated this through two distinct findings: First, increasing the compression levels of compression-based similarity measures improves descriptiveness, aligning with the findings of Ragkhitwetsagul, Krinke and Clark [22]. Furthermore, our results show that code normalization improves the semantic descriptiveness of similarity measures, which also aligns with their findings. Second, the average semantic loss correlates with the performance of the similarity measure as a kNN classifier. We believe, this new method for measuring the descriptive performance of similarity tools, has a lot of potential. As we have observed interesting findings in the preliminary evaluation. First, inexpensive compression-based similarity methods compete with state-of-the-art embedding-based tools. Second, whitespace in code dramatically degrades the performance of embedding-based measures. We are excited about future work in this field. We believe that the ability to evaluate embeddings against various types of normalization can serve as a helpful tool for assessing what models learn.

Acknowledgment

We like to thank the anonymous reviewers for their insightful comments, which have improved the paper greatly. This work is supported by the Innovation Fund Denmark through the project AI4SE1DK (4354-00006B) "Human-Centered Adoption of Artificial Intelligence for Software Engineering in Denmark".

References

1. Reprint of: Mahalanobis, p.c. (1936) "on the generalised distance in statistics.". Sankhya A **80**(1), 1–7 (Dec 2018). https://doi.org/10.1007/s13171-019-00164-5, https://doi.org/10.1007/s13171-019-00164-5
2. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: learning distributed representations of code. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019). https://doi.org/10.1145/3290353, https://doi.org/10.1145/3290353
3. Dinesen, O., Kalhauge, C.: ohodi-sse/simbench: Artifact for perr26 (2026). https://doi.org/10.5281/ZENODO.21087361, https://zenodo.org/doi/10.5281/zenodo.21087361
4. Ding, Y., Peng, J., Min, M.J., Kaiser, G., Yang, J., Ray, B.: Semcoder: Training code language models with comprehensive semantics reasoning (2024), https://arxiv.org/abs/2406.01006
5. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M.: Codebert: A pre-trained model for programming and natural languages (2020), https://arxiv.org/abs/2002.08155
6. Google: google-java-format. https://github.com/google/google-java-format.git (2026)
7. Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S.K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D., Zhou, M.: Graphcodebert: Pre-training code representations with data flow (2021), https://arxiv.org/abs/2009.08366
8. Hamming, R.W.: Error detecting and error correcting codes. The Bell System Technical Journal **29**(2), 147–160 (1950). https://doi.org/10.1002/j.1538-7305.1950.tb00463.x
9. Hannenhalli, S., Pevzner, P.A.: Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. J. ACM **46**(1), 1–27 (Jan 1999). https://doi.org/10.1145/300515.300516, https://doi.org/10.1145/300515.300516
10. Hugging Face: Pooling strategies. https://deepwiki.com/huggingface/text-embeddings-inference/4.4-pooling-strategies (2024), deepWiki documentation for Text Embeddings Inference
11. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: Deckard: Scalable and accurate tree-based detection of code clones. In: 29th International Conference on Software Engineering (ICSE'07). pp. 96–105 (2007). https://doi.org/10.1109/ICSE.2007.30
12. Kamiya, T., Kusumoto, S., Inoue, K.: Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering **28**(7), 654–670 (2002). https://doi.org/10.1109/TSE.2002.1019480
13. Keogh, E., Lonardi, S., Ratanamahatana, C.A.: Towards parameter-free data mining. In: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 206–215 (2004)

14. Kitsios, K., Sovrano, F., Barr, E.T., Bacchelli, A.: Detecting semantic clones of unseen functionality. In: 2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1312–1324 (2025). https://doi.org/10.1109/ASE63991.2025.00112
15. Laneve, C., Spanò, A., Ressi, D., Rossi, S., Bugliesi, M.: Understanding code semantics: a benchmark study of llms. International Journal on Software Tools for Technology Transfer **28**(3), 329–343 (Jun 2026). https://doi.org/10.1007/s10009-026-00842-4, https://doi.org/10.1007/s10009-026-00842-4
16. Levy, A., Shalom, B.R., Chalamish, M.: A guide to similarity measures (2024), https://arxiv.org/abs/2408.07706
17. Li, M., Chen, X., Li, X., Ma, B., Vitanyi, P.: The similarity metric. IEEE Transactions on Information Theory **50**(12), 3250–3264 (2004). https://doi.org/10.1109/TIT.2004.838101
18. Li, M., Vitányi, P.: An Introduction to Kolmogorov Complexity and Its Applications (01 1997). https://doi.org/10.1007/978-1-4757-2606-0
19. Liu, J., Xia, C.S., Wang, Y., Zhang, L.: Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation (2023), https://arxiv.org/abs/2305.01210
20. Puri, R., Kung, D.S., Janssen, G., Zhang, W., Domeniconi, G., Zolotov, V., Dolby, J., Chen, J., Choudhury, M.R., Decker, L., Thost, V., Buratti, L., Pujar, S., Finkler, U.: Project codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. CoRR **abs/2105.12655** (2021), https://arxiv.org/abs/2105.12655
21. Ragkhitwetsagul, C., Krinke, J.: Using compilation/decompilation to enhance clone detection. In: 2017 IEEE 11th International Workshop on Software Clones (IWSC). pp. 1–7 (2017). https://doi.org/10.1109/IWSC.2017.7880502
22. Ragkhitwetsagul, C., Krinke, J., Clark, D.: A comparison of code similarity analysers. Empirical Software Engineering **23** (08 2018). https://doi.org/10.1007/s10664-017-9564-7
23. Silverman, B.W., Jones, M.C.: E. fix and j.l. hodges (1951): An important contribution to nonparametric discriminant analysis and density estimation: Commentary on fix and hodges (1951). International Statistical Review / Revue Internationale de Statistique **57**(3), 233–238 (1989), http://www.jstor.org/stable/1403796
24. Strobel, M.: Procyon java decompiler. https://github.com/mstrobel/procyon (2022)
25. Wei, A., Cao, J., Li, R., Chen, H., Zhang, Y., Wang, Z., Liu, Y., Teixeira, T.S.F.X., Yang, D., Wang, K., Aiken, A.: Equibench: Benchmarking large language models' reasoning about program semantics via equivalence checking (2025), https://arxiv.org/abs/2502.12466
26. Zadrozny, B., Elkan, C.: Transforming classifier scores into accurate multiclass probability estimates. In: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. p. 694–699. KDD '02, Association for Computing Machinery, New York, NY, USA (2002). https://doi.org/10.1145/775047.775151, https://doi.org/10.1145/775047.775151