International Journal of Software Engineering and Knowledge Engineering
© World Scientific Publishing Company

Augmenting the Interpretability of GraphCodeBERT for Code Similarity Tasks

Jorge Martinez-Gil
Software Competence Center Hagenberg GmbH
Softwarepark 32a, 4232 Hagenberg, Austria
`jorge.martinez-gil@scch.at`

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Assessing the degree of similarity of code fragments is crucial for ensuring software quality, but it remains challenging due to the need to capture the deeper semantic aspects of code. Traditional syntactic methods often fail to identify these connections. Recent advancements have addressed this challenge, though they frequently sacrifice interpretability. To improve this, we present an approach aiming to augment the transparency of the similarity assessment by using GraphCodeBERT, which enables the identification of semantic relationships between code fragments. This approach identifies similar code fragments and clarifies the reasons behind that identification, helping developers better understand and trust the results. The source code for our implementation is available at `https://www.github.com/jorge-martinez-gil/graphcodebert-interpretability`.

Keywords: Software Engineering, GraphCodeBERT, Semantic Code, Source Code Similarity

1. Introduction

The growing complexity of software systems requires appropriate methods for analyzing code, particularly those that can recognize and compare code on a deeper level beyond syntax [27]. Traditional strategies tend to focus on surface-level similarity, which can result in missing important aspects, especially in cases where code is written using different styles. This means that these methods may yield incomplete or misleading results, impacting the effectiveness of code maintenance and optimization efforts within increasingly complex projects.

To face these problems, transformer models like GraphCodeBERT [11] have undergone pre-training on extensive datasets of source code. These models provide a promising solution for a more effective exploration of the code. They use a transformer architecture to analyze relationships and meaning. The idea behind learning from diverse examples across various programming languages allows GraphCodeBERT to identify the intent behind different code fragments, even when those fragments appear different on the surface. Its strong performance have made Graph-

CodeBERT a popular choice for code similarity tasks [11].

However, the complexity of these models poses a challenge, as their vast architectures make them difficult for humans to interpret [21]. It is, therefore, crucial to explore methods that improve our understanding of these models [5]. This work faces the interpretability challenge by introducing a GraphCodeBERT-driven framework that visualizes the model's inner workings while determining the similarity between code fragments. The key contributions of this research include:

- We introduce a new approach using GraphCodeBERT to visualize the semantic similarity between two code fragments. This approach breaks down the code into tokens and embeds them into high-dimensional vectors to assess the significance of each token relative to the others.
- The approach produces visual representations that illustrate how closely the tokens in two code fragments are related in terms of meaning. Our strategy proves particularly useful in assessing code quality, detecting potential plagiarism, or identifying opportunities for refactoring.
- Furthermore, this method has potential applications across various aspects of software engineering. For example, it can suggest code improvements or improve automated code generation systems. We aim to demonstrate how GraphCodeBERT's capabilities can contribute to developing more maintainable software systems.

The rest of our work guides the reader through the research: Section 2 examines existing methods and places this work within the context of related research. Section 3 outlines the approach, from tokenization to creating similarity matrices. Section 4 presents a use case for applying the proposed strategy to classical sorting algorithms. Section 5 discusses the strengths, limitations, and potential future lines of research. The conclusion summarizes the key contributions and the importance of the findings.

2. Literature Survey

Researchers have made significant progress in code representation, especially with the introduction of pre-trained models for programming languages. These models, primarily based on transformer architectures, achieve strong performance in tasks such as code completion [4] and similarity detection [14]. This section reviews the current approaches relevant to our work.

2.1. *Pre-trained Models for Code Understanding*

Pre-trained models have changed the field of automatic text processing. The idea is to use large-scale unsupervised training on vast amounts of textual data [7]. Building on these advances, researchers applied similar techniques to programming languages [10].

CodeBERT [10], one of the seminal models in this area, uses a bi-directional transformer pre-trained on large datasets of source code. It is designed to learn both syntactic and semantic aspects of code and has been fine-tuned for tasks such as code repair [19], clone detection [2], or code plagiarism assessment [3].

An extension of CodeBERT, GraphCodeBERT [11], includes additional structural information. The idea behind incorporating data flow graphs is to allow for capturing deeper aspects, improving its performance for tasks requiring a detailed understanding of code behavior. Models of this kind have many applications, for example, identification of code vulnerabilities [26].

2.2. *Tokenization and Embedding Techniques*

Tokenization is a critical step in preparing code for machine learning models. Traditional tokenization methods often struggle with programming languages due to their unique syntax and use of various symbols. In recent time, novel models using specialized tokenization techniques have been developed. These models allow to handle a range of programming languages [12].

After tokenization, each token t_i is mapped to a high-dimensional vector embedding $\mathbf{v}_i \in \mathbb{R}^d$, where d is the dimensionality of the embedding space. These embeddings encode valuable information about each token [24]. In GraphCodeBERT, including data flow information enriches these embeddings, allowing for a richer representation that captures deeper aspects of the code [11].

2.3. *Code Similarity Detection*

Code similarity detection is essential in tasks like clone identification [28] and automated code review [15]. Earlier methods relied on syntactic matching or basic structural analysis, which often missed relationships between code fragments [18].

More recent methods use pre-trained models to measure code similarity through learned embeddings. These models, such as GraphCodeBERT, compute similarity scores between code fragments using embeddings, capturing more subtle similarities than syntax-based approaches can [16].

Advanced techniques also use attention mechanisms to focus on the most relevant parts of the code during similarity computation [17]. This helps the models assign importance to tokens based on their role in the structure, improving the accuracy of similarity assessments.

2.4. *Visualization Techniques for Code Comparison*

Visualization is an effective way to interpret the results of code similarity analysis [23]. Current methods often present similarity matrices or heatmaps, where the degree of similarity between tokens from different code fragments is represented with color gradients. This offers an intuitive way to explore relationships between code segments and allows for detailed analysis of similarity.

GraphCodeBERT's integration of data flow graphs would improve the clarity of these visualizations. It would allow users to see which tokens are similar and how data moves through the code. This is useful for tasks where understanding the code elements is crucial.

2.5. *Contribution over the State-of-the-Art*

Earlier approaches have faced the interpretability challenge using less advanced models [1]. This work proposes a method that improves transparency in code similarity by utilizing GraphCodeBERT to identify relationships between code fragments. This method can explain why two code pieces are similar by revealing their connections. These explanations allow developers to understand the matching process better and gain more confidence in the results.

3. Methodology

Our approach uses the pre-trained GraphCodeBERT transformer model to capture deep semantic relationships within code, providing an improved understanding beyond mere syntactical analysis [22]. The methodology is divided into several steps: starting with the use of the pre-trained GraphCodeBERT model, we proceed through the processes of tokenization, input representation, and the application of an attention mechanism to determine the relative significance of each token within the code. Finally, we describe the result generation, which is visualized in diverse ways, offering a graphical representation of the connections between tokens in different code fragments.

3.1. *Pre-trained GraphCodeBERT Model*

GraphCodeBERT has been fine-tuned on large-scale code datasets across multiple programming languages. This model captures complex patterns in code, making it ideal for tasks needing deep code understanding. Mathematically, the model can be expressed as a function $\mathcal{F} : \mathcal{T} \to \mathbb{R}^d$, where $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ is the input tokens, and $\mathcal{V} = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$ is the corresponding vector embeddings, such as:

$$\mathcal{V} = \mathcal{F}(\mathcal{T}) = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}, \quad \mathbf{v}_i \in \mathbb{R}^d \tag{1}$$

The model architecture, shown in Figure 1, consists of multiple components working in sequence. Source code and its data flow are the starting inputs, which undergo tokenization to break down the code into manageable parts. These tokens are combined with data flow embeddings, capturing additional relationships within the code. A transformer encoder then processes these embeddings to generate output vectors suitable for clone detection tasks.

Tokenization splits the source code into meaningful units called tokens. These tokens are then transformed into high-dimensional vector embeddings that encode semantic information [9].

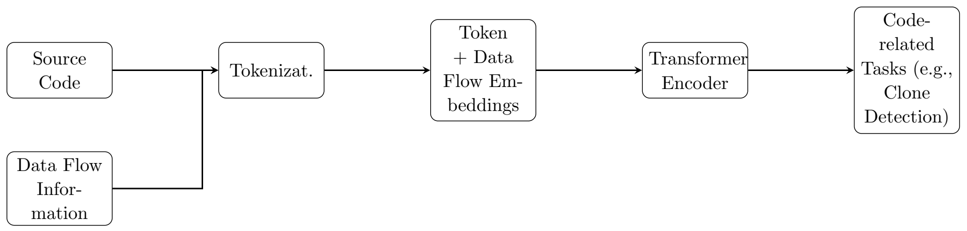

Source Code
Data Flow Information
Tokenizat.
Token + Data Flow Embeddings
Transformer Encoder
Code-related Tasks (e.g., Clone Detection)

Fig. 1: GraphCodeBERT architecture combines source code with data flow information, tokenizes the input, and uses a Transformer encoder to generate embeddings for code-related tasks.

The process of tokenizing a given code fragment c is performed by splitting it into a sequence of tokens $T = \{t_1, t_2, \ldots, t_n\}$. The tokenization can be formalized as:

$$T = \text{Tokenize}(c) = \{t_i \mid t_i \in \text{Tokens}(c)\} \tag{2}$$

where each token t_i is identified based on alphanumeric characters that are crucial to the code's syntax and structure. The output sequence T serves as the input to the embedding layer.

Each token t_i is mapped to a vector embedding $\mathbf{v}_i$ using $\mathcal{F}$ that represents a function that maps each token t_i to its corresponding vector embedding $\mathbf{v}_i$ through the pre-trained GraphCodeBERT model:

$$\mathbf{v}_i = \mathcal{F}(t_i), \quad \mathbf{v}_i \in \mathbb{R}^d \tag{3}$$

The input sequence $\mathcal{V}$ is then represented as:

$$\mathcal{V} = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\} \tag{4}$$

The attention mechanism in GraphCodeBERT calculates attention weights for each token. These weights indicate the relevance of one token to others in the sequence. This helps the model focus on the most relevant tokens when assessing the similarity between code fragments. Let $\mathbf{Q}$ and $\mathbf{K}$ represent the query and key matrices, respectively, derived from the input embeddings:

$$\mathbf{Q} = \mathbf{W}_Q \mathcal{V}, \quad \mathbf{K} = \mathbf{W}_K \mathcal{V}, \quad \mathbf{V} = \mathbf{W}_V \mathcal{V} \tag{5}$$

where $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V \in \mathbb{R}^{d \times d}$ are learnable weight matrices. The scaled dot-product attention is computed as:

$$\mathbf{A} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V} \tag{6}$$

where d_k is the dimensionality of the key vectors. The resulting attention matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ represents the weight assigned to each token pair, indicating the influence of token t_j on token t_i.

Given two code fragments c_1 and c_2, with their respective token sequences $T_1 = \{t_1^{(1)}, t_2^{(1)}, \ldots, t_{n_1}^{(1)}\}$ and $T_2 = \{t_1^{(2)}, t_2^{(2)}, \ldots, t_{n_2}^{(2)}\}$, and their corresponding attention matrices $\mathbf{A}_1$ and $\mathbf{A}_2$, the computation of the similarity matrix $\mathbf{S}$ should be:

$$\mathbf{S} = \mathbf{A}_1 \mathbf{A}_2^\top \in \mathbb{R}^{n_1 \times n_2} \tag{7}$$

The element S_{ij} in the similarity matrix represents the semantic similarity between token $t_i^{(1)}$ from T_1 and token $t_j^{(2)}$ from T_2. In order to capture higher-order relationships, the similarity matrix can be refined through multiple attention heads:

$$\mathbf{S} = \frac{1}{H} \sum_{h=1}^{H} \mathbf{A}_1^{(h)} \mathbf{A}_2^{(h)\top} \tag{8}$$

where H is the number of attention heads.

3.2. *Dimensionality Reduction and Visualization*

In order to visualize the high-dimensional embeddings generated by GraphCodeBERT, we use Principal Component Analysis (PCA) [20], t-distributed Stochastic Neighbor Embedding (t-SNE) [8], and Uniform Manifold Approximation and Projection (UMAP) [8].

PCA is intended to reduce the dimensionality while preserving the data structure. In fact, it can identify the principal components that capture the maximum variance in the data. PCA is defined as:

$$\mathbf{Z} = \mathbf{V}\mathbf{W}_{PCA}, \quad \mathbf{Z} \in \mathbb{R}^{n \times p}, \quad \mathbf{W}_{PCA} \in \mathbb{R}^{d \times p} \tag{9}$$

where $p < d$ is the reduced dimensionality, $\mathbf{V}$ represents the original data matrix with n samples and d features, $\mathbf{W}_{PCA}$ contains the top p principal component vectors, and $\mathbf{Z}$ is the transformed data in the lower p-dimensional space.

t-SNE focuses on preserving the local structure of the data, making it well-suited for visualizing clusters. t-SNE maps high-dimensional points $\mathbf{v}_i$ to lower-dimensional points $\mathbf{y}_i \in \mathbb{R}^q$ by minimizing the Kullback-Leibler divergence between high-dimensional and low-dimensional points:

$$\mathrm{KL}(P \parallel Q) = \sum_{i \neq j} P_{ij} \log \frac{P_{ij}}{Q_{ij}} \tag{10}$$

where P_{ij} and Q_{ij} are the joint probabilities of pairs of points in the high-dimensional and low-dimensional spaces, respectively. t-SNE is effective for visualizing small-scale patterns, but it can be computationally intensive.

UMAP is a more recent technique that combines the strengths of both PCA and t-SNE while addressing some limitations. UMAP aims to preserve the data's local and global structures. To do that, it builds a high-dimensional graph representation of the data, which optimizes for a low-dimensional layout.

The idea is to build a weighted graph $G = (V, E)$ where V are the vertices representing points and E are the edges connecting these points. The edge weights are determined by pairwise similarities between the points and are calculated as follows:

$$w_{ij} = \exp\left(-\frac{d(v_i, v_j) - \rho_i}{\sigma_i}\right) \tag{11}$$

where:

- $d(v_i, v_j)$ is the distance between points v_i and v_j in the original high-dimensional space.
- ρ_i is the distance to the nearest neighbor of v_i.
- σ_i is a scaling factor controlling the distance distribution's spread.

The UMAP method then optimizes the low-dimensional embedding Y by minimizing a cross-entropy loss between both the high-dimensional and the low-dimensional representation such as:

$$\text{argmin}_Y \sum_{(i,j)\in E} w_{ij} \log\left(\frac{w_{ij}}{\text{dist}(y_i, y_j)}\right) + (1 - w_{ij}) \log\left(\frac{1 - w_{ij}}{1 - \text{dist}(y_i, y_j)}\right) \tag{12}$$

where:

- $\text{dist}(y_i, y_j)$ is the distance between the low-dimensional points y_i and y_j in the embedding.
- Y represents the coordinates of all points in the low-dimensional space.

In order to make the analysis more accessible to developers, we incorporate a range of visualization techniques. These methods improve the clarity of relationships between fragments and allow us to explore the data more deeply [25]. In addition, we use saliency maps to interpret the importance of individual tokens in the embeddings. Saliency maps help identify which tokens contribute the most to the similarity to generate a visual representation that shows the influential tokens [13].

Let the similarity score between two code fragments be denoted as $\text{sim}(\mathbf{F}_1, \mathbf{F}_2)$. The saliency map for a token embedding $\mathbf{e}_i^{(1)}$ from the first fragment is computed as the gradient of the similarity score concerning that embedding:

$$\text{saliency}(\mathbf{e}_i^{(1)}) = \left\| \frac{\partial \text{sim}(\mathbf{F}_1, \mathbf{F}_2)}{\partial \mathbf{e}_i^{(1)}} \right\| \tag{13}$$

The capability to visualize these saliency maps alongside the code fragments aims to help developers better understand which parts of the code are driving the assessment of the similarity.

3.3. *Similarity Computation*

In order to quantify the similarity between two fragments, their respective embeddings should be compared using cosine similarity. Let $\mathbf{e}_i^{(1)}$ and $\mathbf{e}_j^{(2)}$ denote the embeddings of the i-th token in the first fragment and the j-th token in the second fragment, respectively. The similarity between these embeddings can be calculated as follows:

$$\text{cosine\_sim}(\mathbf{e}_i^{(1)}, \mathbf{e}_j^{(2)}) = \frac{\mathbf{e}_i^{(1)} \cdot \mathbf{e}_j^{(2)}}{\|\mathbf{e}_i^{(1)}\| \|\mathbf{e}_j^{(2)}\|} \tag{14}$$

The resulting matrix $\mathbf{S}$, where $S_{ij} = \text{cosine\_sim}(\mathbf{e}_i^{(1)}, \mathbf{e}_j^{(2)})$, serves as the basis for further analysis.

3.4. *Interpretation Strategy*

Our strategy for augmenting interpretability involves multiple visualization methods to process the embeddings and their similarities:

- PCA and t-SNE are helpful to reduce the dimensionality of the embeddings. This allows for visual inspection of the embedding space. These methods reveal clusters in the data.
- UMAP is used to visualize high-dimensional data. This method is intended to preserve more of the global structure of the data compared to t-SNE and can often produce more meaningful clusters.
- Saliency maps are used to visualize the contribution of individual features in the embedding to the output. The idea is to identify the parts of the input that influence the model's predictions most.

Each method adds value to understanding how embeddings represent and organize information. The idea is to understand how embeddings represent and organize information differently. This approach helps analyze patterns and the specific features driving model behavior.

4. Use Case

We introduce a method to improve understanding of how semantic similarity between classical sorting algorithms can be evaluated. Sorting algorithms are fundamental to computer science and the foundation for many computer-related systems. We have chosen to work with sorting algorithms due to their fundamental nature, well-documented structure, and widespread use in benchmarks [6].

We focus on five well-known sorting algorithms: Bubble Sort swaps adjacent elements to move the largest to the end (Listing 1). Selection Sort finds the smallest unsorted element and swaps it with the first unsorted position (Listing 2). Insertion Sort places each element into its correct position in the sorted part (Listing 3). Merge Sort splits, sorts, and merges sublists recursively (Listing 4). Quick Sort partitions around a pivot and sorts recursively (Listing 5). The goal is to improve the analysis of code representations to show details that may not be immediately obvious through standard methods.

It is necessary to note that GraphCodeBERT may sometimes produce results that differ from expectations due to issues like tokenization errors, gaps in training data, or difficulties handling complex code structures. These issues can be addressed by expanding training datasets to include diverse programming languages and styles, improving tokenization processes to align more closely with programming syntax, and adding post-processing steps to ensure correctness. Regular testing across varied cases helps maintain reliability, and methods like TF-IDF with Cosine Similarity or AST-based Tree Kernels can serve as useful benchmarks for evaluating these improvements.

```
def bubble_sort(arr):
    for i in range(len(arr)):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Listing 1: An implementation of the Bubble Sort algorithm as a simple comparison-based sorting algorithm that repeatedly steps through the list.

```
def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

Listing 2: An implementation of the Selection Sort algorithm that repeatedly selects the smallest element from the unsorted list portion.

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]
        merge_sort(L)
        merge_sort(R)

        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1; k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1; k += 1
```

Listing 3: An implementation of the Merge Sort algorithm that recursively divides the array into two halves sorts each half and then merges them.

```python
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1

    for j in range(low, high):
        if arr[j] <= pivot:
            i = i + 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)

        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)
```

Listing 4: An implementation of the Quick Sort algorithm that selects a pivot partitions the array around the pivot and recursively sorts the subarrays.

```python
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i-1
        while j >=0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

Listing 5: An implementation of the Insertion Sort algorithm that builds the sorted array one element at a time by repeatedly inserting elements.

Each algorithm was implemented in Python and represented as a code string. These code strings were tokenized and processed through GraphCodeBERT to generate vector embeddings, which encapsulate the structural and semantic properties of the code. The resulting similarities were visualized in a heatmap, as shown in Figure 2.

The heatmap reveals that algorithms sharing similar structural patterns or operational steps, such as Insertion Sort and Bubble Sort, exhibit higher similarity scores. At the same time, algorithms with fundamentally different approaches, like Quick Sort and Bubble Sort, display lower similarity scores. This approach provides interesting insights into different sorting algorithms' inherent relationships. Our goal is that our approach can help us understand these results in greater depth.

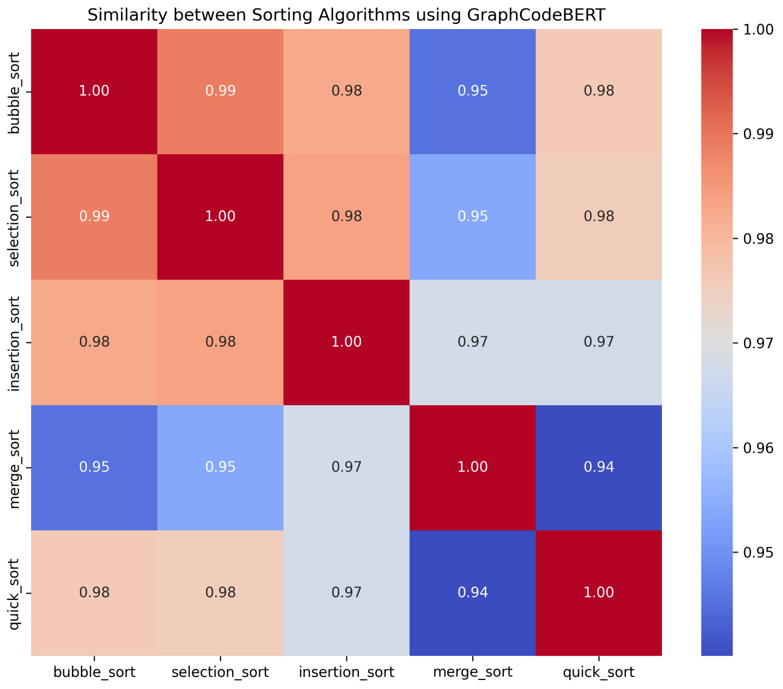
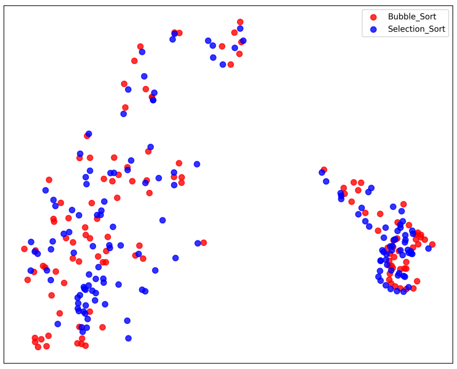


Fig. 2: Visualization of the similarity relationships among various classical sorting algorithms using GraphCodeBERT. This heatmap shows how similar sorting algorithms are, based on their structure and behavior, using GraphCodeBERT. Algorithms like Bubble Sort and Insertion Sort, which follow similar step-by-step processes, show higher similarity scores.

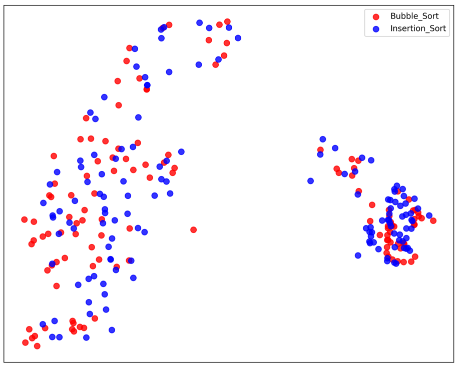
4.1. *Pairwise Comparisons of Sorting Algorithms*

We present pairwise comparisons of the sorting algorithms by projecting their token embeddings into a two-dimensional space using PCA, t-SNE, and UMAP, respectively. Our strategy helps to visualize the relationships between the embeddings in a simplified form. Each image provided shows the distribution of token embeddings for a specific pair of sorting algorithms, with different colors assigned to each algorithm for a clear distinction. These visualizations allow us to observe how the embeddings of different algorithms spread out in the 2D space.

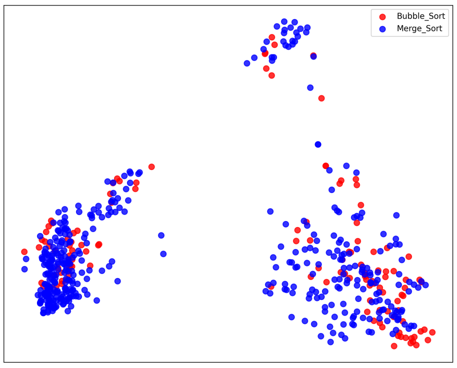
Figures 3 and 4 use PCA to compare sorting algorithms through their token embeddings. Similarly, Figures 5 and 6 apply t-SNE, focusing on the differences and similarities in representation. Figures 7 and 8 rely on UMAP to map structural patterns among these embeddings, providing an alternative view of the relationships.



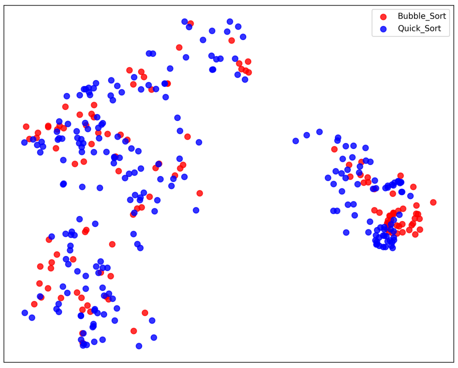
(a) Bubble Sort vs. Selection Sort



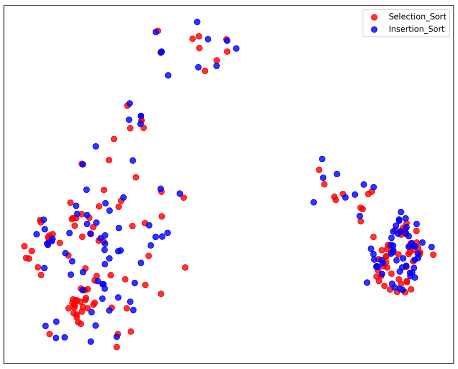
(b) Bubble Sort vs. Insertion Sort



(c) Bubble Sort vs. Merge Sort

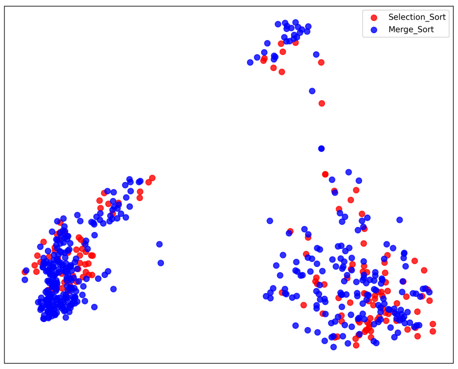


(d) Bubble Sort vs. Quick Sort

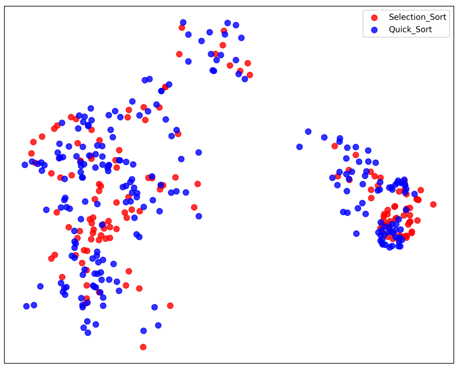


(e) Selection Sort vs. Insertion Sort

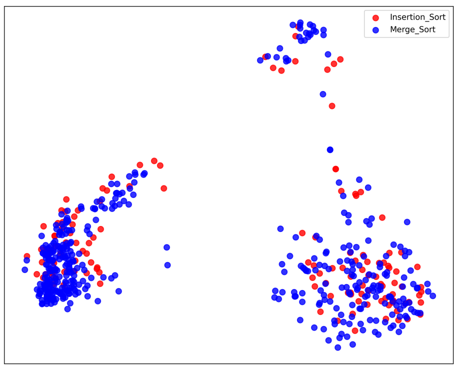
Fig. 3: Pairwise comparisons of classical sorting algorithms using PCA, showing the token embeddings in a 2D space (Part 1).



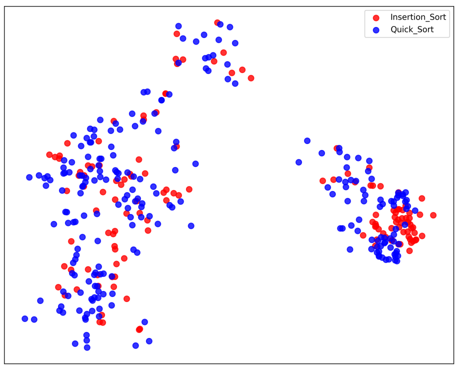
(a) Selection Sort vs. Merge Sort



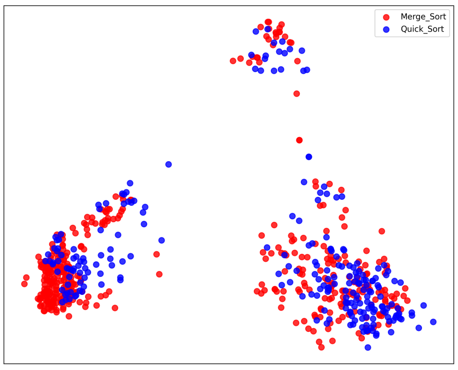
(b) Selection Sort vs. Quick Sort



(c) Insertion Sort vs. Merge Sort

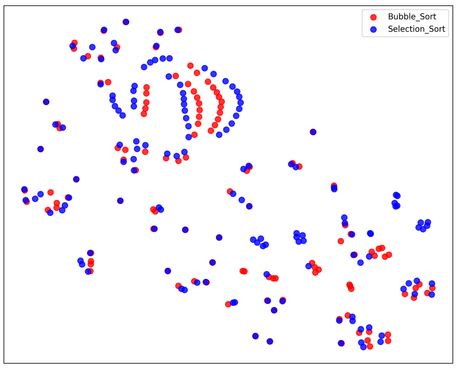


(d) Insertion Sort vs. Quick Sort

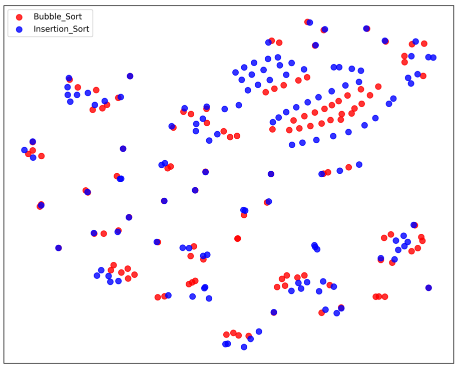


(e) Merge Sort vs. Quick Sort

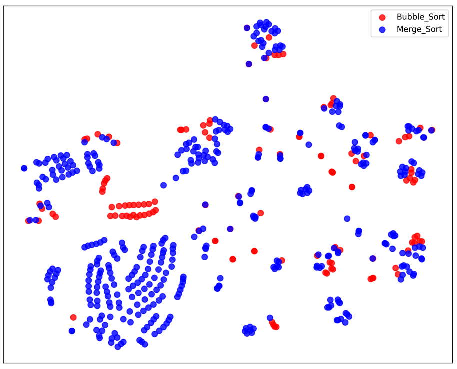
Fig. 4: Pairwise comparisons of classical sorting algorithms using PCA, showing the token embeddings in a 2D space (Part 2).



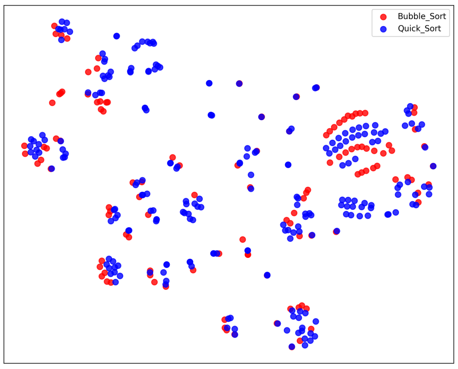
(a) Bubble Sort vs. Selection Sort



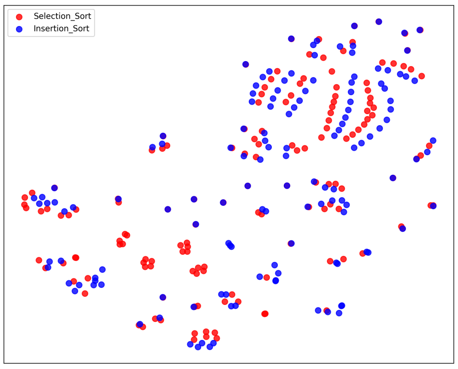
(b) Bubble Sort vs. Insertion Sort



(c) Bubble Sort vs. Merge Sort

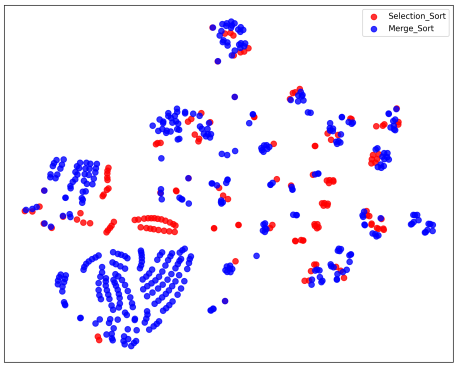


(d) Bubble Sort vs. Quick Sort

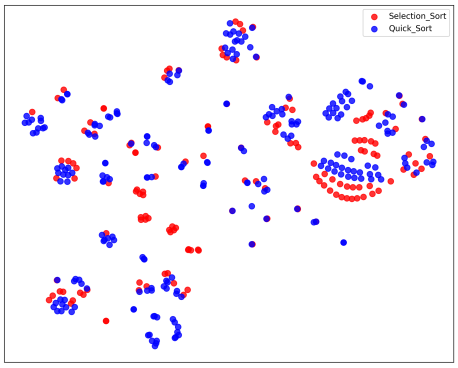


(e) Selection Sort vs. Insertion Sort

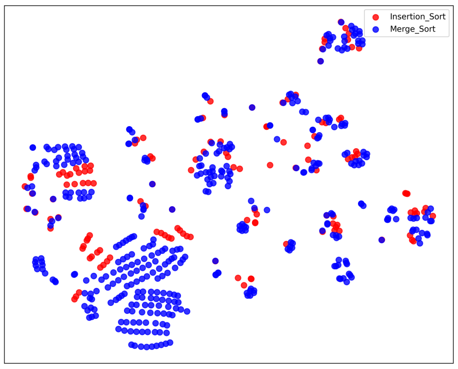
Fig. 5: Pairwise comparisons of classical sorting algorithms using t-SNE, showing the token embeddings in a 2D space (Part 1).



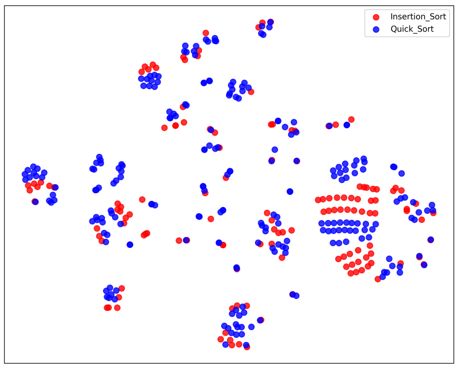
(a) Selection Sort vs. Merge Sort



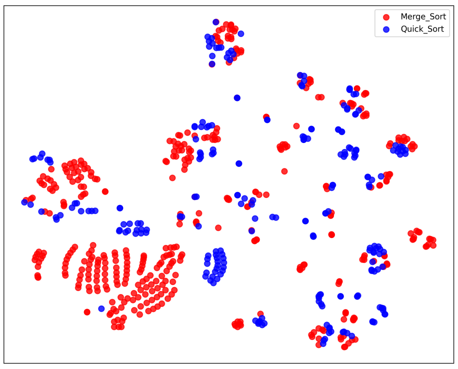
(b) Selection Sort vs. Quick Sort



(c) Insertion Sort vs. Merge Sort

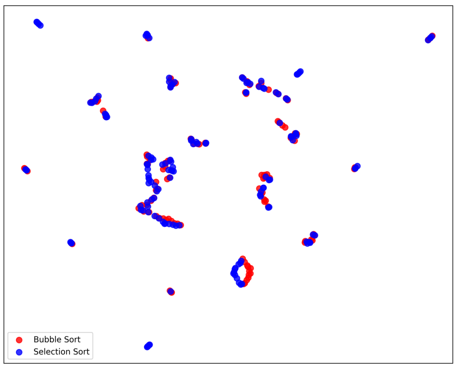


(d) Insertion Sort vs. Quick Sort

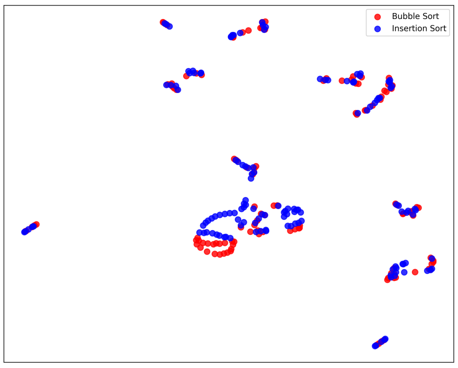


(e) Merge Sort vs. Quick Sort

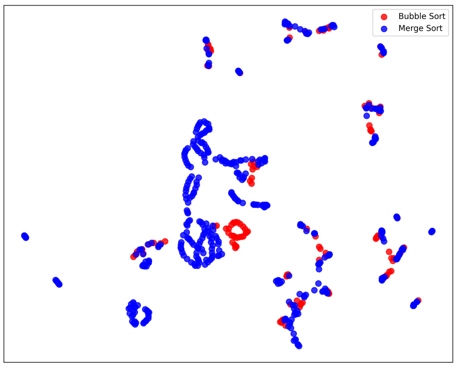
Fig. 6: Pairwise comparisons of classical sorting algorithms using t-SNE, showing the token embeddings in a 2D space (Part 2).



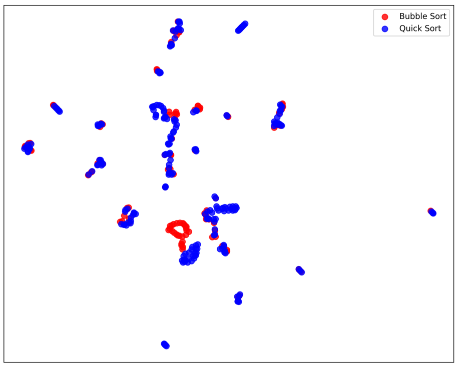
(a) Bubble Sort vs. Selection Sort



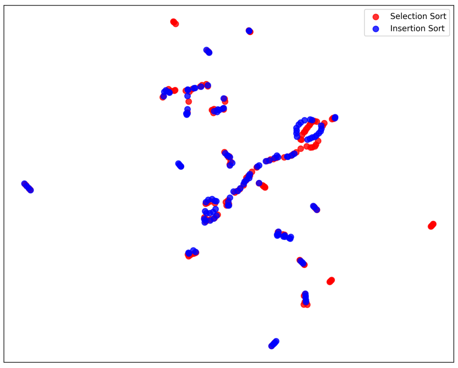
(b) Bubble Sort vs. Insertion Sort



(c) Bubble Sort vs. Merge Sort

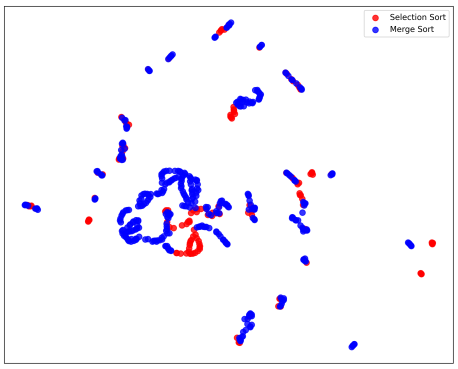


(d) Bubble Sort vs. Quick Sort

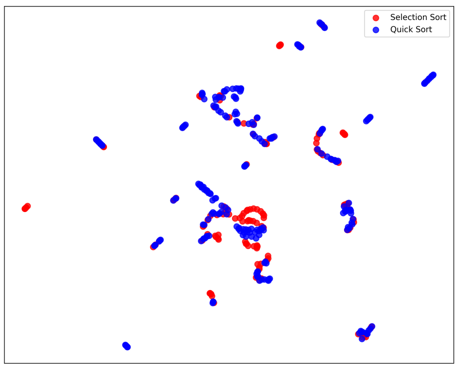


(e) Selection Sort vs. Insertion Sort

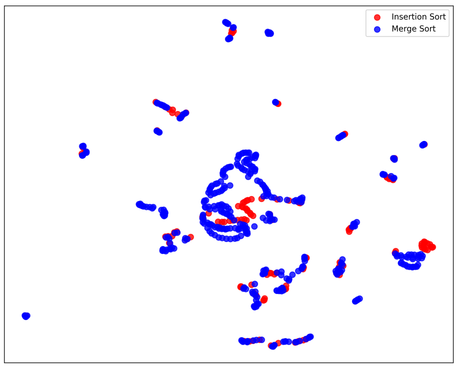
Fig. 7: Pairwise comparisons of classical sorting algorithms using UMAP, showing the token-level embeddings in a 2D space (Part 1).



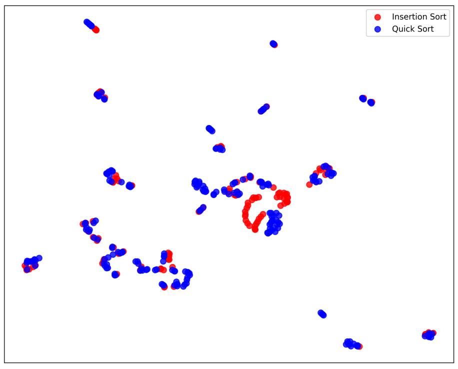
(a) Selection Sort vs. Merge Sort



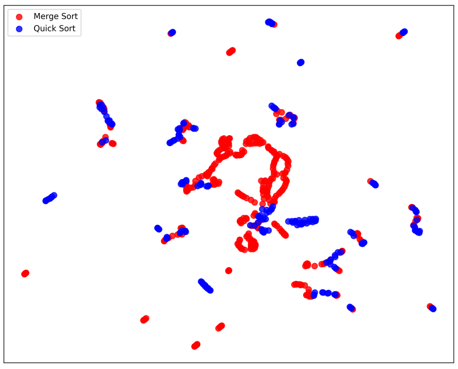
(b) Selection Sort vs. Quick Sort



(c) Insertion Sort vs. Merge Sort



(d) Insertion Sort vs. Quick Sort

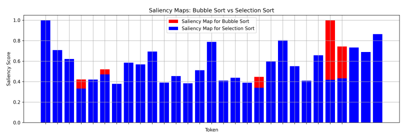


(e) Merge Sort vs. Quick Sort

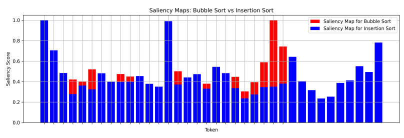
Fig. 8: Pairwise comparisons of classical sorting algorithms using UMAP, showing the token-level embeddings in a 2D space (Part 2).

4.2. *Saliency Maps for Sorting Algorithms*

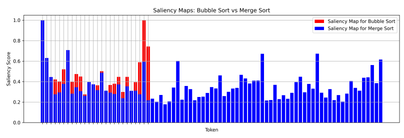
Saliency maps provide a visual explanation of which parts of the input contributed most to the model's decisions, offering hints into how the model interprets the similarities between the algorithms. These maps allow us to observe which tokens the model focuses on when distinguishing between the sorting algorithms. Figure 9 illustrates the saliency maps for these comparisons, focusing on the critical areas of attention within the token embeddings.



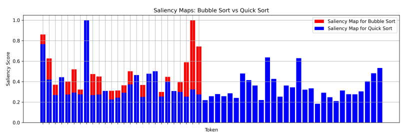
(a) Bubble Sort vs Selection Sort



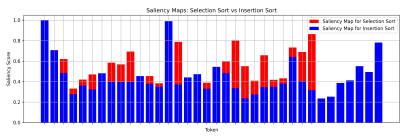
(b) Bubble Sort vs Insertion Sort



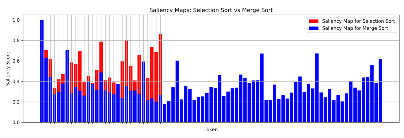
(c) Bubble Sort vs Merge Sort



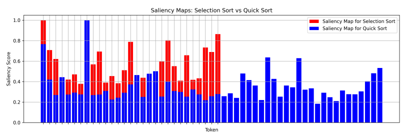
(d) Bubble Sort vs Quick Sort



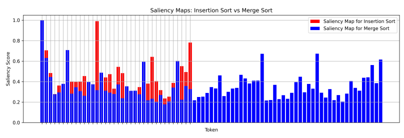
(e) Selection Sort vs Insertion Sort



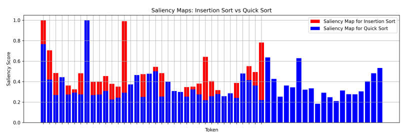
(f) Selection Sort vs Merge Sort



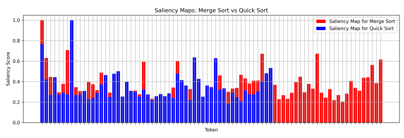
(g) Selection Sort vs Quick Sort



(h) Insertion Sort vs Merge Sort



(i) Insertion Sort vs Quick Sort



(j) Merge Sort vs Quick Sort

Fig. 9: Saliency maps for pairwise comparisons of different sorting algorithms.

The results show that this method holds considerable potential for various applications. For example, automated code reviews help reviewers quickly identify redundancies, reducing maintenance costs. In refactoring, this method can assist developers in determining code segments that can be optimized. Additionally, in educational settings, this approach can aid in teaching by offering a deeper knowledge of the underlying principles of algorithms.

4.3. *Ablation Study*

To examine the role of various components in our approach, we have conducted an ablation study comparing configurations for analyzing the embeddings of the sorting algorithms. This analysis has included comparing token embeddings representing individual tokens in the code with pooled text embeddings, reducing the entire code snippet into a single vector. The study can be found in our GitHub repository[a], but some interesting facts are that token embeddings displayed distinct clusters corresponding to each algorithm. In contrast, pooled text embeddings showed a strong resemblance between Bubble Sort and Selection Sort due to their shared processing style, while Quick Sort and Merge Sort were more distinct owing to their different strategy.

5. Discussion

We have seen that our strategy can improve interpretability by presenting different approaches to visualize similarities between code fragments through the use of GraphCodeBERT. We have seen that this approach improves over traditional comparison approaches, providing deeper insights into the functional similarities between those fragments.

However, several areas deserve further investigation. One key area is the method's scalability when applied to larger projects. While our initial experiments focused on relatively small examples, real-world software projects involve thousands of lines of code. It could be interesting to assess how well the model performs under such conditions in terms of the quality of the generated outputs.

Another important aspect is the applicability across different programming languages. Although our experiments produced promising results in single-language comparisons, extending the evaluation to a broader range of languages would offer a more complete assessment of the method's usefulness.

Future work may also explore integrating this approach with other code analysis methods. For instance, combining our method with static analysis or code metrics tools could improve code maintainability. Additionally, incorporating feedback mechanisms where developers can refine the outputs could improve the usability of the results.

[a]`https://www.github.com/jorge-martinez-gil/graphcodebert-interpretability`

6. Conclusion

Our research has introduced a novel strategy for increasing the interpretability capabilities of the similarity between code fragments using GraphCodeBERT. The process is formalized through mathematical expressions, presenting a framework that captures and displays the relationships in an interpretable visual format. Our experimental results demonstrate the method's effectiveness in explaining similarities between code fragments.

This approach has implications for improving code understanding, guaranteeing code quality, and improving software development processes. Our method contributes to developing more maintainable software systems by simplifying tasks such as automated code review, refactoring, and plagiarism detection. Future research will aim to expand the applicability of this method, potentially integrating it into broader software engineering tools.

Additional future directions include the development of more language-agnostic models and improvements in the integration of structural information. Furthermore, exploring the combination of machine learning-based approaches with traditional program analysis techniques could lead to more accurate models. However, challenges remain in generalizing across different programming languages and coding styles.

Acknowledgments

The author thanks all the reviewers for their help in improving the manuscript. This research has been funded by the Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation, and Technology (BMK), the Federal Ministry for Digital and Economic Affairs (BMDW), and the State of Upper Austria in the frame of SCCH, a center in the COMET - Competence Centers for Excellent Technologies Programme managed by Austrian Research Promotion Agency FFG.

References

[1] Shamsa Abid, Xuemeng Cai, and Lingxiao Jiang. Interpreting codebert for semantic code clone detection. In *30th Asia-Pacific Software Engineering Conference, APSEC 2023, Seoul, Republic of Korea, December 4-7, 2023*, pages 229–238. IEEE, 2023.

[2] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. A systematic review on code clone detection. *IEEE access*, 7:86121–86144, 2019.

[3] Romain Brixtel, Mathieu Fontaine, Boris Lesner, Cyril Bazin, and Romain Robbes. Language-independent clone detection applied to plagiarism detection. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, pages 77–86. IEEE, 2010.

[4] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 213–222, 2009.

[5] Hila Chefer, Shir Gur, and Lior Wolf. Transformer interpretability beyond attention visualization. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*, pages 782–791. Computer Vision Foundation / IEEE, 2021.

[6] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[8] Andrew Draganov and Simon Dohn. Unexplainable explanations: Towards interpreting tsne and UMAP embeddings. *CoRR*, abs/2306.11898, 2023.

[9] Siyue Feng, Wenqi Suo, Yueming Wu, Deqing Zou, Yang Liu, and Hai Jin. Machine learning is all you need: A simple token-based approach for effective code clone detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics, 2020.

[11] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[12] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning*, pages 5110–5121, 2020.

[13] Itzik Malkiel, Dvir Ginzburg, Oren Barkan, Avi Caciularu, Jonathan Weill, and Noam Koenigstein. Interpreting bert-based text similarity via activation and saliency maps. In *Proceedings of the ACM Web Conference 2022*, pages 3259–3268, 2022.

[14] Jorge Martinez-Gil. A comprehensive review of stacking methods for semantic similarity measurement. *Machine Learning with Applications*, 10:100423, 2022.

[15] Jorge Martinez-Gil. A comparative study of ensemble techniques based on genetic programming: A case study in semantic similarity assessment. *Int. J. Softw. Eng. Knowl. Eng.*, 33(2):289–312, 2023.

[16] Jorge Martinez-Gil. Advanced detection of source code clones via an ensemble of unsupervised similarity measures. *CoRR*, abs/2405.02095, 2024.

[17] Jorge Martinez-Gil. Improving source code similarity detection through graphcodebert and integration of additional features. 2024.

[18] Jorge Martinez-Gil. Source code clone detection using unsupervised similarity measures. In Peter Bludau, Rudolf Ramler, Dietmar Winkler, and Johannes Bergsmann, editors, *Software Quality as a Foundation for Security - 16th International Conference on Software Quality, SWQD 2024, Vienna, Austria, April 23-25, 2024, Proceed-*

ings, volume 505 of *Lecture Notes in Business Information Processing*, pages 21–37. Springer, 2024.

[19] Ehsan Mashhadi and Hadi Hemmati. Applying codebert for automated program repair of java simple bugs. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*, pages 505–509. IEEE, 2021.

[20] Tomás Musil. Examining structure of word embeddings with PCA. In Kamil Ekstein, editor, *Text, Speech, and Dialogue - 22nd International Conference, TSD 2019, Ljubljana, Slovenia, September 11-13, 2019, Proceedings*, volume 11697 of *Lecture Notes in Computer Science*, pages 211–223. Springer, 2019.

[21] Daking Rai, Yilun Zhou, Shi Feng, Abulhair Saparov, and Ziyu Yao. A practical review of mechanistic interpretability for transformer-based language models. *CoRR*, abs/2407.02646, 2024.

[22] Mootez Saad and Tushar Sharma. Naturalness of attention: Revisiting attention in code language models. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 107–111, 2024.

[23] Daniel Smilkov, Nikhil Thorat, Charles Nicholson, Emily Reif, Fernanda B. Viégas, and Martin Wattenberg. Embedding projector: Interactive visualization and interpretation of embeddings. *CoRR*, abs/1611.05469, 2016.

[24] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 1433–1443, 2020.

[25] Jesse Vig. A multiscale visualization of attention in the transformer model. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 37–42, 2019.

[26] Rongcun Wang, Senlei Xu, Yuan Tian, Xingyu Ji, Xiaobing Sun, and Shujuan Jiang. SCL-CVD: supervised contrastive learning for code vulnerability detection via graphcodebert. *Comput. Secur.*, 145:103994, 2024.

[27] Huihui Wei and Ming Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *IJCAI*, pages 3034–3040, 2017.

[28] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 87–98, 2016.