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

Abstract

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 improve 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, Source Code Similarity, Transformer Models, Clone Detection

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 [22]. 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, without a deeper understanding, these methods may yield incomplete or misleading results, impacting the effectiveness of code maintenance and optimization efforts within increasingly large and complex projects.

To face these problems, transformer models like GraphCodeBERT [8] have undergone pre-training on extensive datasets of source code. These models provide a promising solution for a more effective exploration of the semantic structure of 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 and functionality behind different code fragments, even when those fragments appear different on the surface.

However, the complexity of these models poses a challenge, as their vast architectures and encoded multi-dimensional information make them difficult for humans to interpret [19]. It is, therefore, crucial to explore methods that improve our understanding of these models [4]. 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 efficient and 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 emphasizes the importance of the findings.

2. State-of-the-Art

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 [3] and similarity detection [16, 9, 10, 11]. 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 [5]. Building on these advances, researchers applied similar techniques to programming languages [7].

CodeBERT [7], the seminal model 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 [17] and clone detection [2].

An extension of CodeBERT, GraphCodeBERT [8], includes additional structural information. Incorporating data flow graphs, representing the relationships between variables, allows for capturing deeper semantic details, 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 [21].

2.2. Tokenization and Embedding Techniques

Tokenization is a critical step in preparing code for machine learning models. Traditional tokenization methods used in text processing often struggle with programming languages due to their unique syntax and use of various symbols and operators. More recent models use specialized tokenization techniques better suited for code, allowing them to handle a range of programming languages effectively.

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 are pre-trained using large datasets

and encode valuable information about each token. In GraphCodeBERT, including data flow information enriches these embeddings, allowing for a representation that captures both the lexical and structural aspects of the code.

2.3. Code Similarity Detection

Code similarity detection is essential in software engineering for tasks like clone identification [23] and automated code review [12]. Earlier methods relied on syntactic matching or basic structural analysis, which often missed deeper semantic relationships between code fragments [15].

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 differences and similarities than syntax-based approaches can [13].

Advanced techniques also use attention mechanisms to focus on the most relevant parts of the code during similarity computation [14]. This helps the models assign importance to tokens based on their role in the overall semantic 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 [20]. 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 by emphasizing areas of greater similarity.

GraphCodeBERT's integration of structural information from 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 relationships between code elements is crucial.

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

Earlier approaches have face the interpretability challenge using less advanced models [1]. This work proposes a method that improves transparency in code similarity assessments by utilizing GraphCodeBERT to identify relationships between code fragments. This method can detect code clones and explains why two code pieces are considered 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. The methodology is divided into several key 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 an intuitive graphical representation of the semantic connections between tokens in different code fragments.

3.1. Pre-trained GraphCodeBERT Model

GraphCodeBERT has been specifically 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 $\mathcal{F} : \mathcal{T} \rightarrow \mathbb{R}^d$, 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. Formally:

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

3.2. Tokenization

Tokenization splits the source code into meaningful units called tokens (e.g., keywords, operators, or identifiers). These tokens are then transformed into high-dimensional vector embeddings that encode semantic information.

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 or operators that are crucial to the code's syntax and structure. The output sequence T serves as the input to the embedding layer.

3.3. Input Representation

Each token t_i is mapped to a 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:

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

3.4. Attention Mechanism

The attention mechanism in GraphCodeBERT calculates attention weights for each token. These weights indicate the significance 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.

3.5. Similarity Matrix

Given two code fragments c_1 and c_2, with their respective token sequences $T_1 = \{t_1^{(1)}, t_2^{(1)}, \dots, t_{n_1}^{(1)}\}$ and $T_2 = \{t_1^{(2)}, t_2^{(2)}, \dots, t_{n_2}^{(2)}\}$, 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.

3.6. Dimensionality Reduction and Visualization

In order to effectively visualize the high-dimensional embeddings generated by GraphCodeBERT, we use Principal Component Analysis (PCA) [18], t-distributed Stochastic Neighbor Embedding (t-SNE) [6], and Uniform Manifold Approximation and Projection (UMAP) [6]. Each of these methods serves a distinct purpose in the dimensionality reduction process.

PCA reduces the dimensionality while preserving the data structure in terms of variance. In fact, it can identify the principal components (linear combinations of the original variables) that capture the maximum variance in the data. PCA is defined as:

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

where $p < d$ is the reduced dimensionality, $\mathbf{V}$ represents the original data, and $\mathbf{W}_{\text{PCA}}$ contains the principal component vectors. PCA is a good method for initial exploration.

t-SNE focuses on preserving the local structure of the data, making it well-suited for visualizing clusters and relationships. 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 the joint probabilities of the high-dimensional and low-dimensional points:

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

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 and may struggle with huge datasets.

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.

This approach is formalized as an optimization problem where a topological structure represents the high-dimensional graph, and the low-dimensional embedding is optimized to preserve this structure.

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

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

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.

UMAP then optimizes the low-dimensional embedding Y by minimizing a cross-entropy loss between the high-dimensional graph and the low-dimensional representation:

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

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 advanced data visualization techniques. These methods improve the clarity of relationships between fragments and allow us to explore the data more deeply. 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 most influential tokens.

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:

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

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

3.7. Similarity Computation

In order to quantify the similarity between two code fragments, their respective token embeddings are 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 cosine similarity between these embeddings is computed as:

$$\text{cosine_sim}(\mathbf{e}_i^{(1)}, \mathbf{e}_j^{(2)}) = \frac{\mathbf{e}_i^{(1)} \cdot \mathbf{e}_j^{(2)}}{\|\mathbf{e}_i^{(1)}\| \|\mathbf{e}_j^{(2)}\|}$$

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.

3.8. Interpretation Strategy

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

- PCA and t-SNE are used to reduce the dimensionality of the embeddings, allowing for visual inspection of the embedding space. These methods reveal clusters in the data.
- UMAP is used to visualize high-dimensional data. UMAP preserves more of the global structure of the data compared to t-SNE and can often produce more meaningful clusters, making it particularly useful for exploring relationships and groupings within the embeddings.
- 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.

4. Use Case

We introduce a method to improve understanding of how similarity between classical sorting algorithms can be evaluated. Sorting algorithms are fundamental to computer science and the foundation for many systems. This study focuses on five well-known sorting algorithms:

- **Bubble Sort**: A simple comparison-based algorithm that repeatedly swaps adjacent elements if they are in the wrong order, slowly bubbling the largest elements to the end.
- **Selection Sort**: An algorithm that repeatedly selects the smallest element from the unsorted part of the list and swaps it with the first unsorted element.
- **Insertion Sort**: A sorting algorithm that builds the final sorted list one element at a time by inserting each element into its correct position among the previously sorted elements.
- **Merge Sort**: A divide-and-conquer algorithm that splits the list into smaller sublists, recursively sorts them, and then merges the sorted sublists.
- **Quick Sort**: A divide-and-conquer algorithm that selects a pivot element, partitions the list around the pivot, and recursively sorts the sublists.

We provide a new perspective on how these algorithms relate to one another beyond their operational characteristics. The aim is to move beyond traditional analysis methods and focus on their code representations to identify aspects that may not be apparent through standard analysis.

Bubble Sort

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

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

Selection Sort

```python
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]
    return arr
```

Listing 2: An implementation of the Selection Sort algorithm in Python. Selection Sort is a straightforward comparison-based sorting algorithm that repeatedly selects the smallest element from the unsorted portion of the list.

Merge Sort

```python
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 in Python that recursively divides the array into two halves sorts each half and then merges them.

Quick Sort

```
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 in Python that selects a pivot partitions the array around the pivot and recursively sorts the subarrays.

Insertion Sort

```
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
    return arr
```

Listing 5: An implementation of the Insertion Sort algorithm in Python 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 1.

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 structural and functional relationships. Our goal is that our approach can help us understand these results in greater depth.

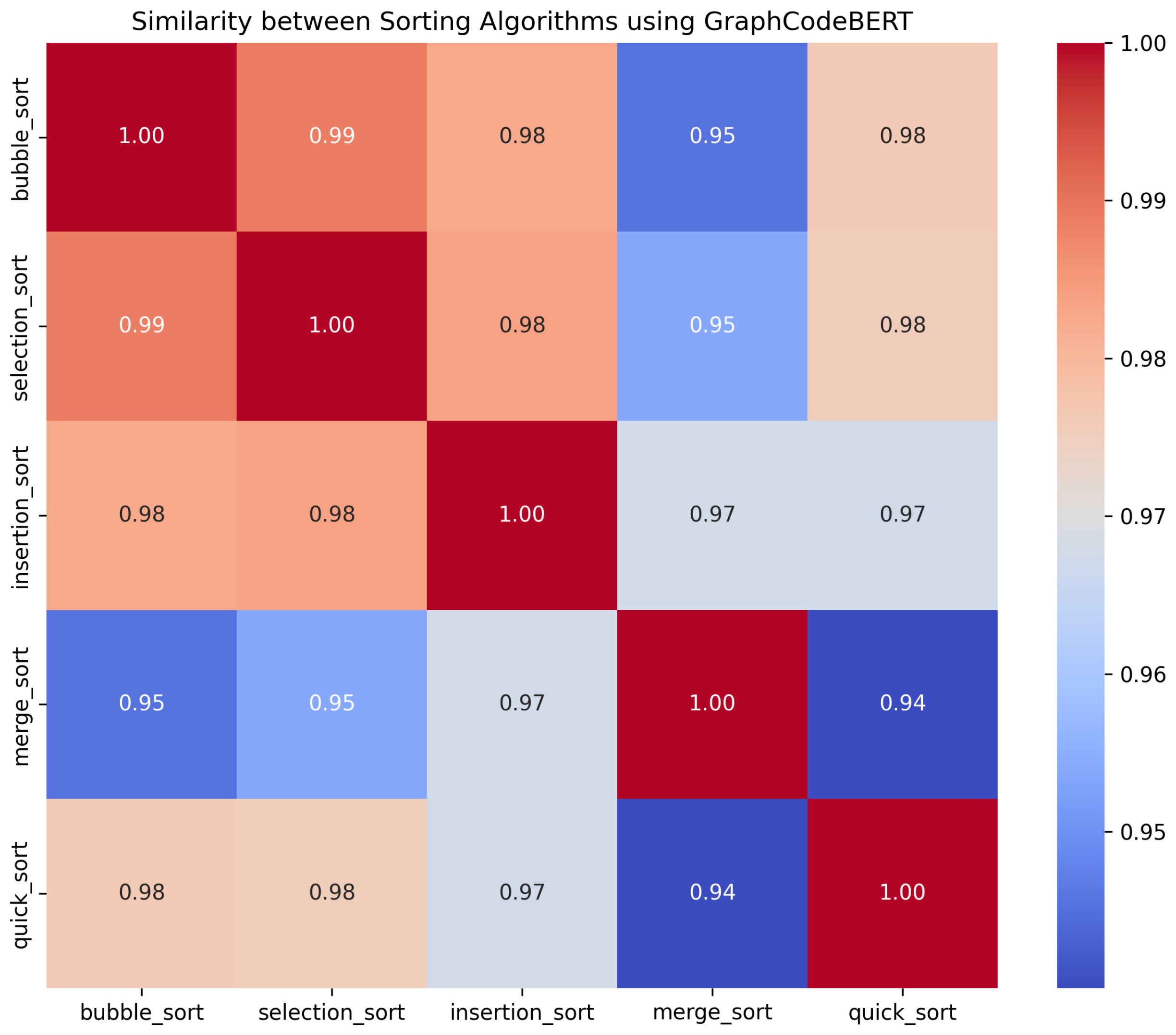


Figure 1: Visualization of the similarity relationships among various classical sorting algorithms using GraphCodeBERT. This heatmap shows how similar sorting algorithms are, based on their structure and behavior, using GraphCodeBERT. Algorithms like Bubble Sort and Insertion Sort, which follow similar step-by-step processes, show higher similarity scores. On the other hand, Quick Sort, which uses a more complex partitioning approach, shows lower similarity with Bubble Sort.

4.1. Pairwise Comparisons of Sorting Algorithms using PCA

We present pairwise comparisons of classical sorting algorithms by projecting their token embeddings into a two-dimensional space using PCA. This method 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 group or spread out in the 2D space.

Figures 2 and 3 illustrate these pairwise comparisons across the mentioned sorting algorithms, visually representing their token embeddings.

4.2. Pairwise Comparisons of Sorting Algorithms using t-SNE

We also present the pairwise comparisons between classical sorting algorithms by visualizing their token embeddings in a two-dimensional space using t-SNE. This technique is particularly effective for high-dimensional data reduction, allowing us to project the token embeddings into a 2D space while preserving the local structure of the data. Once again, each image visually represents the token embeddings for a specific pair of sorting algorithms, with different colors assigned to each algorithm for better differentiation. These visualizations offer a more detailed understanding of how the token embeddings for the algorithms relate to each other.

Figures 4 and 5 illustrate these pairwise comparisons, helping to reveal potential similarities or differences in the way the algorithms are represented through their token embeddings.

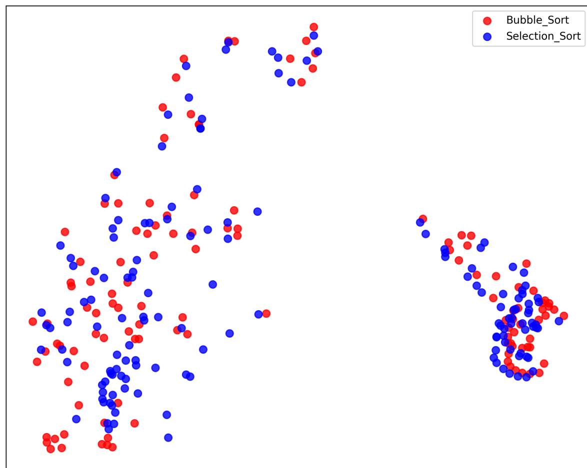
4.3. Pairwise Comparisons of Sorting Algorithms using UMAP

We also present pairwise comparisons between classical sorting algorithms by visualizing their token-level embeddings in a two-dimensional space using UMAP. This method allows for preserving the global and local structures of the high-dimensional token embeddings when projected in 2D. Once again, each image shows the distribution of token embeddings for a specific pair of sorting algorithms, with different colors distinguishing the algorithms. These visualizations provide a deeper look into how the token embeddings for each algorithm are organized and how their representations relate to one another.

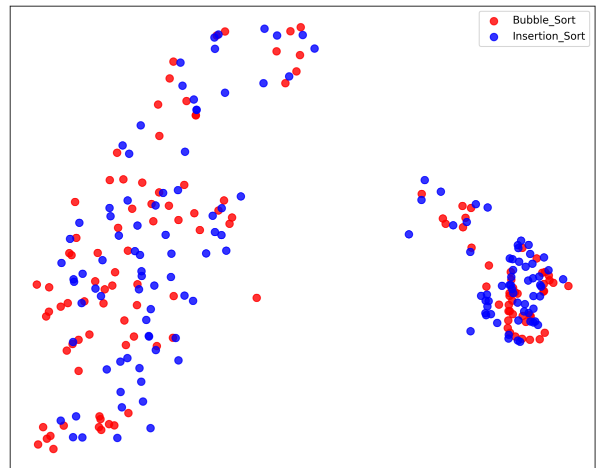
Figures 6 and 7 display these pairwise comparisons, shedding light on potential structural patterns or distinctions among the token embeddings of the algorithms.

4.4. Saliency Maps for Sorting Algorithms

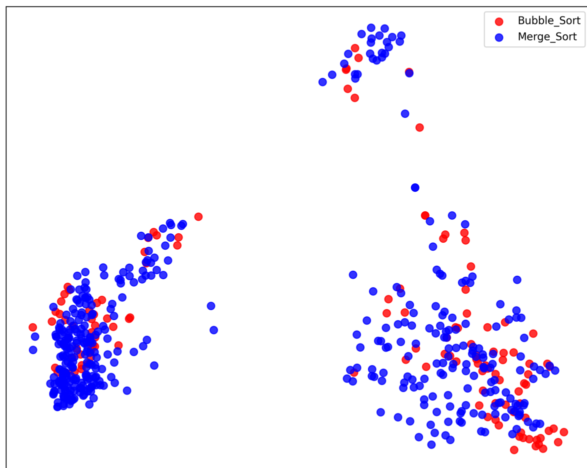
We also present the saliency maps generated for the pairwise comparisons of different sorting algorithms using the GraphCodeBERT model. Saliency maps provide a visual explanation of which parts of the input contributed most to the model's decisions, offering hints into how the model interprets the similarities between the algorithms. These maps allow us to observe which tokens or features the model focuses on when distinguishing between the sorting algorithms. Figure 8 illustrates the saliency maps for these comparisons, focusing on the critical areas of attention within the token embeddings.



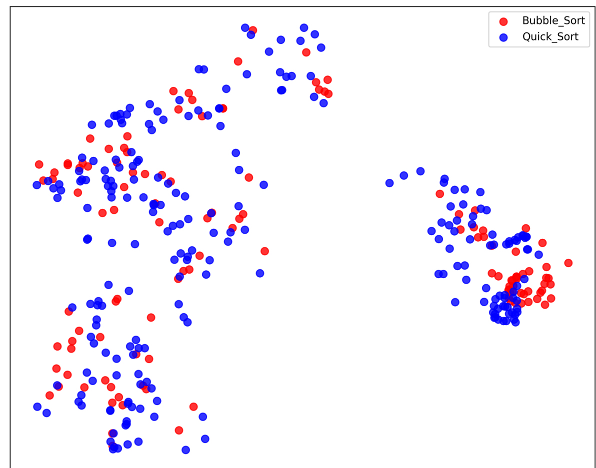
(a) Bubble Sort vs. Selection Sort



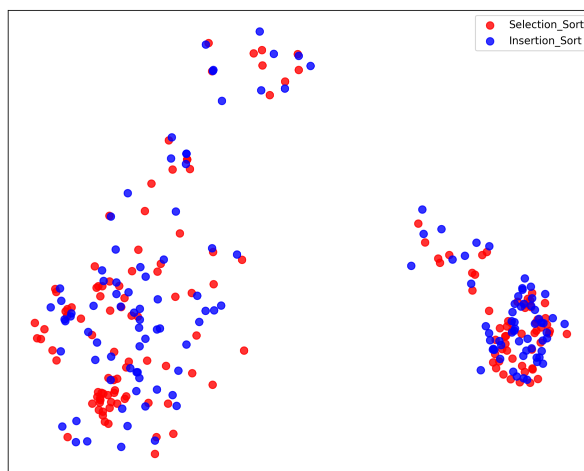
(b) Bubble Sort vs. Insertion Sort



(c) Bubble Sort vs. Merge Sort

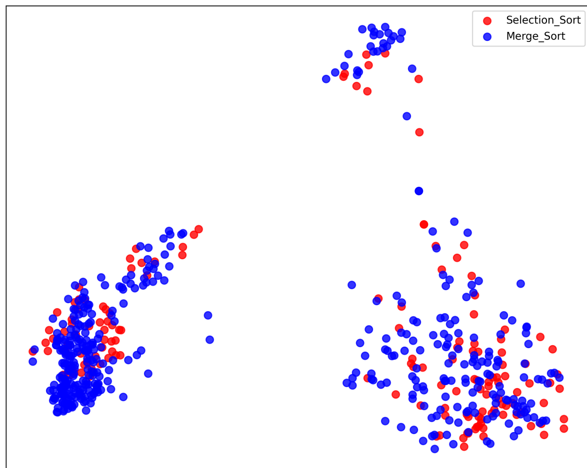


(d) Bubble Sort vs. Quick Sort

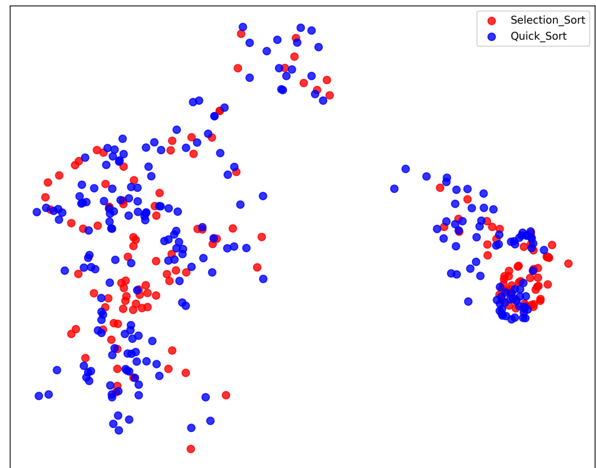


(e) Selection Sort vs. Insertion Sort

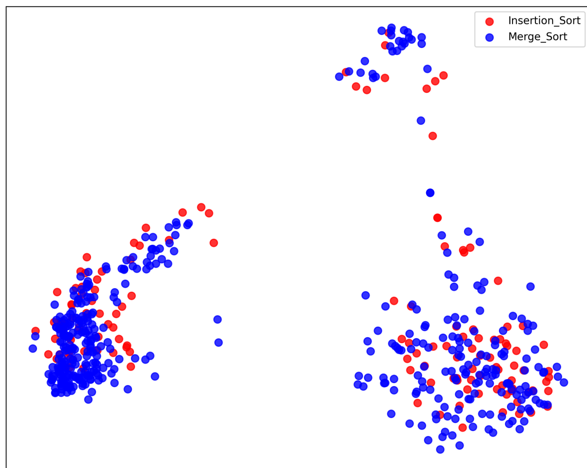
Figure 2: Pairwise comparisons of classical sorting algorithms using PCA, showing the token embeddings in a 2D space (Part 1).



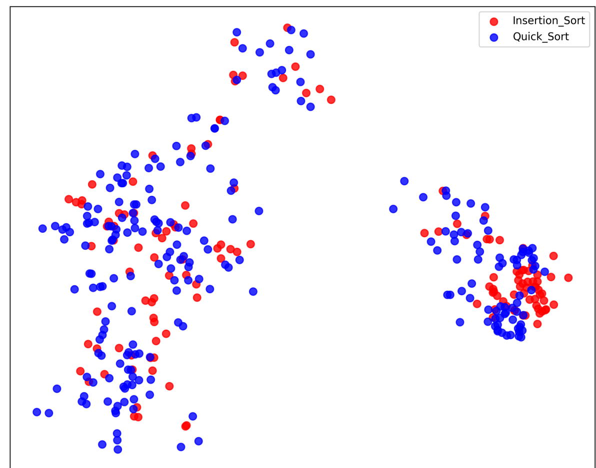
(a) Selection Sort vs. Merge Sort



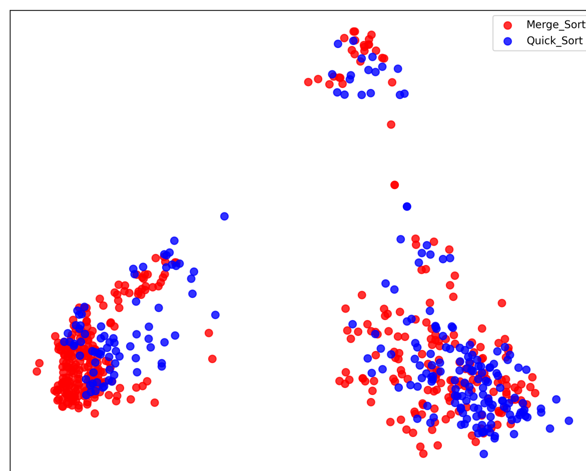
(b) Selection Sort vs. Quick Sort



(c) Insertion Sort vs. Merge Sort

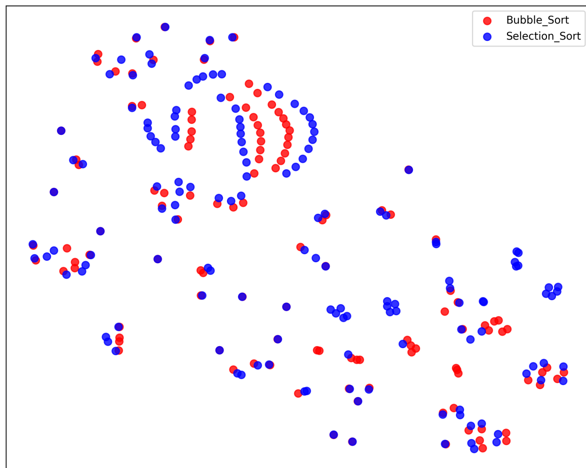


(d) Insertion Sort vs. Quick Sort

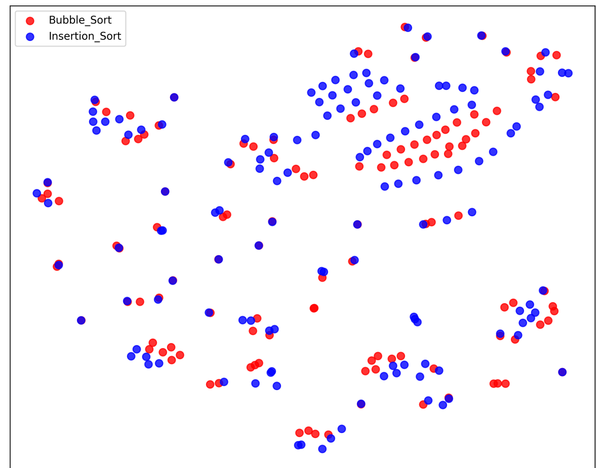


(e) Merge Sort vs. Quick Sort

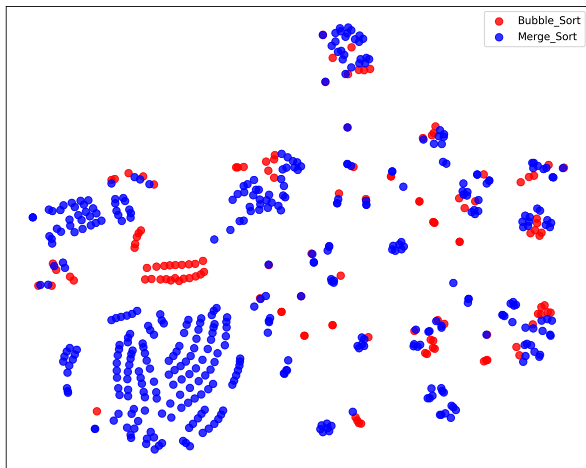
Figure 3: Pairwise comparisons of classical sorting algorithms using PCA, showing the token embeddings in a 2D space (Part 2).



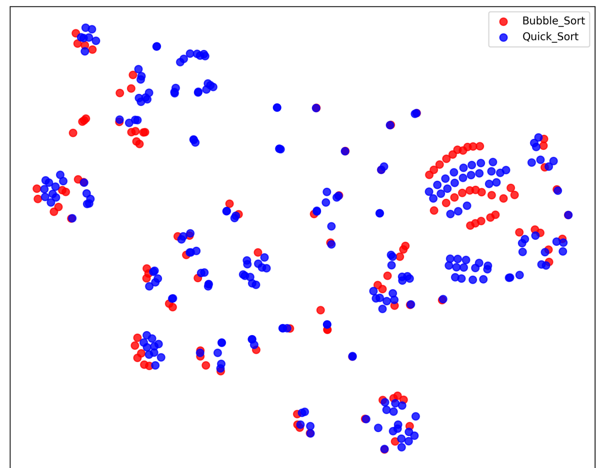
(a) Bubble Sort vs. Selection Sort



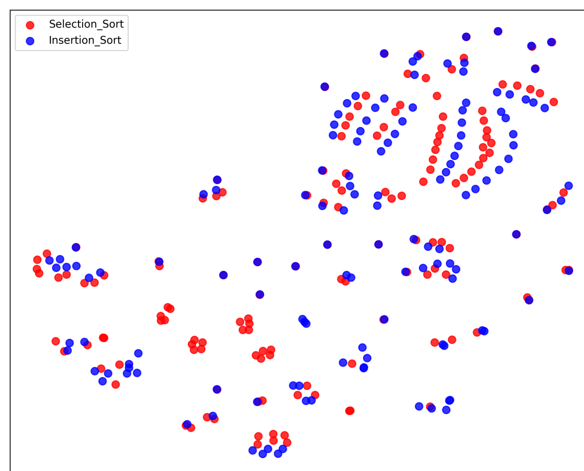
(b) Bubble Sort vs. Insertion Sort



(c) Bubble Sort vs. Merge Sort

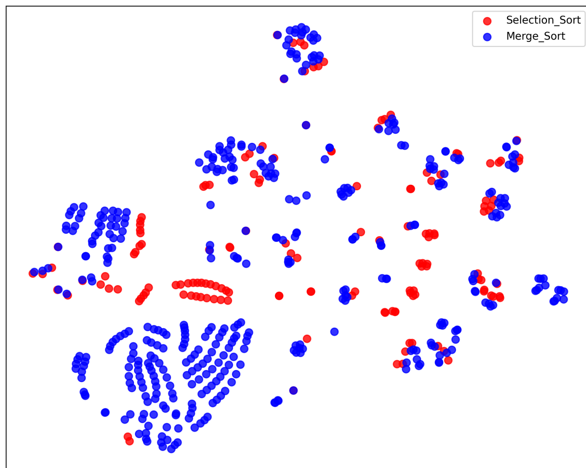


(d) Bubble Sort vs. Quick Sort

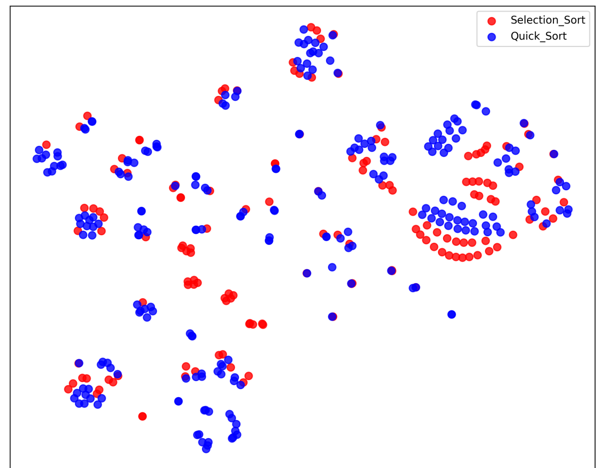


(e) Selection Sort vs. Insertion Sort

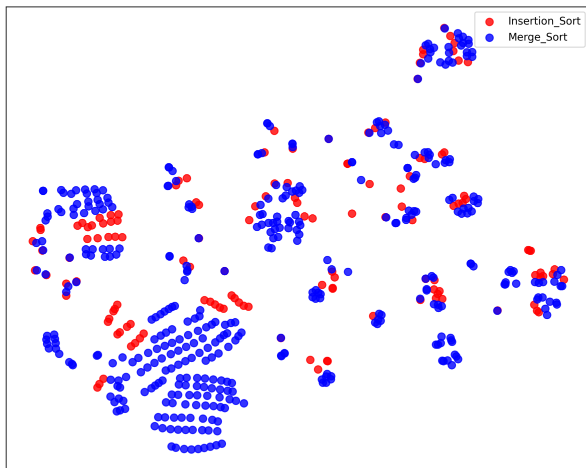
Figure 4: Pairwise comparisons of classical sorting algorithms using t-SNE, showing the token embeddings in a 2D space (Part 1).



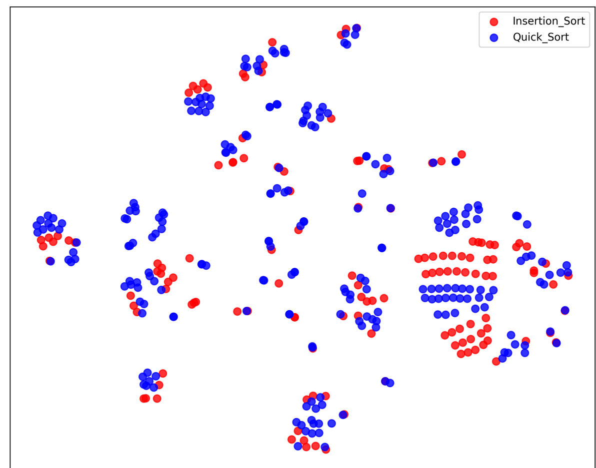
(a) Selection Sort vs. Merge Sort



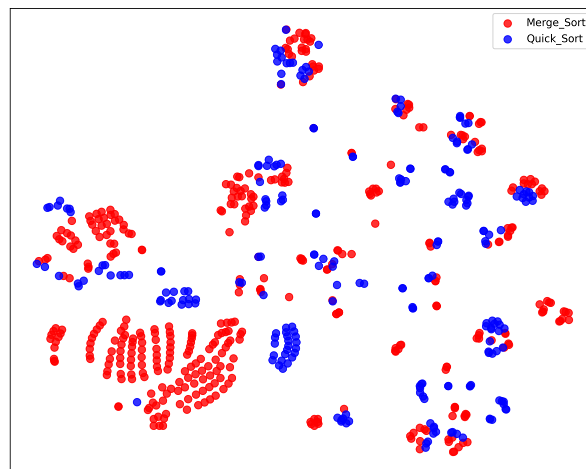
(b) Selection Sort vs. Quick Sort



(c) Insertion Sort vs. Merge Sort

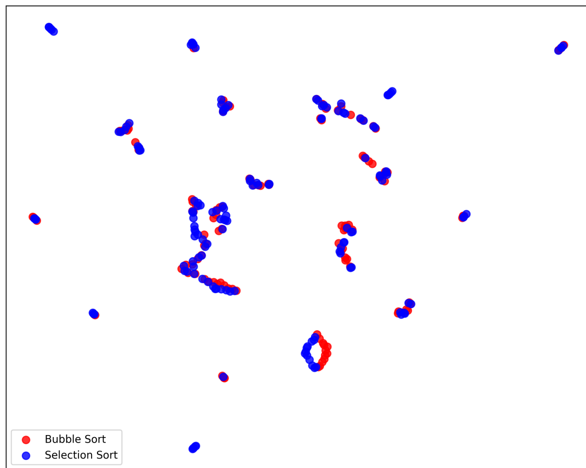


(d) Insertion Sort vs. Quick Sort

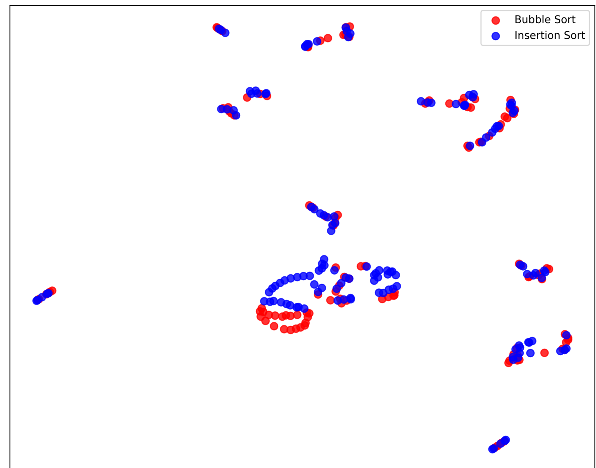


(e) Merge Sort vs. Quick Sort

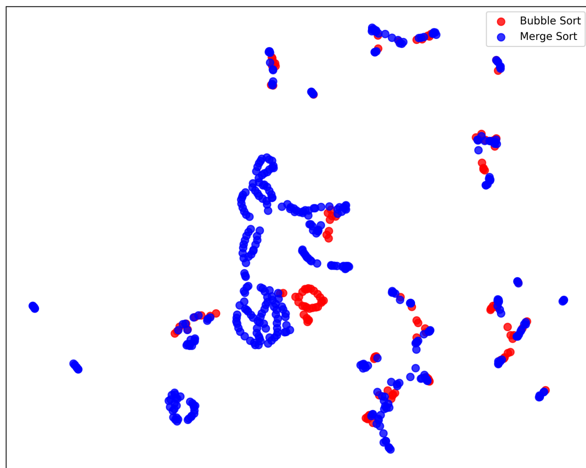
Figure 5: Pairwise comparisons of classical sorting algorithms using t-SNE, showing the token embeddings in a 2D space (Part 2).



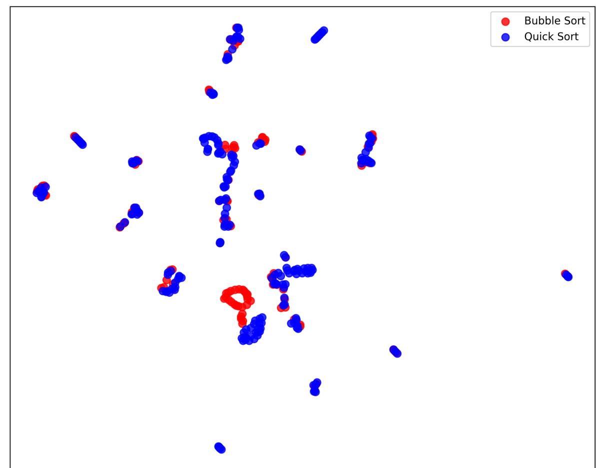
(a) Bubble Sort vs. Selection Sort



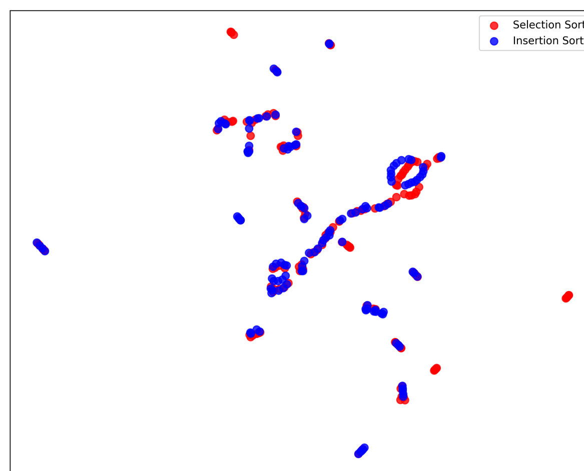
(b) Bubble Sort vs. Insertion Sort



(c) Bubble Sort vs. Merge Sort

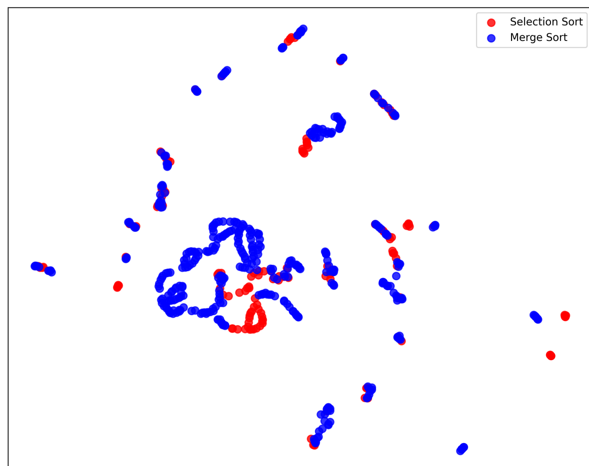


(d) Bubble Sort vs. Quick Sort

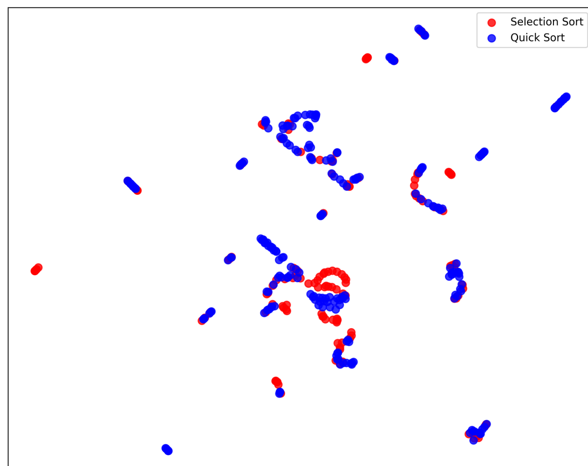


(e) Selection Sort vs. Insertion Sort

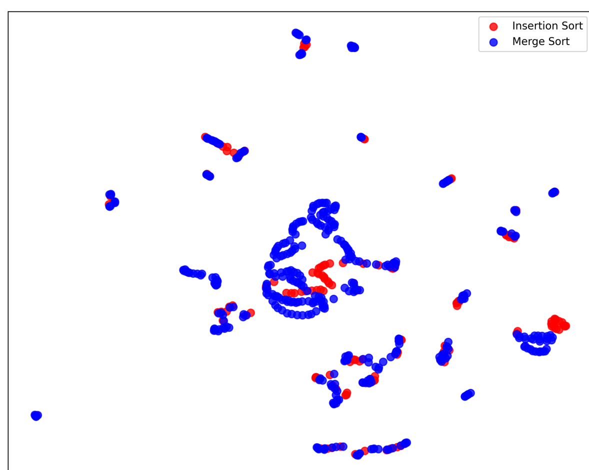
Figure 6: Pairwise comparisons of classical sorting algorithms using UMAP, showing the token-level embeddings in a 2D space (Part 1).



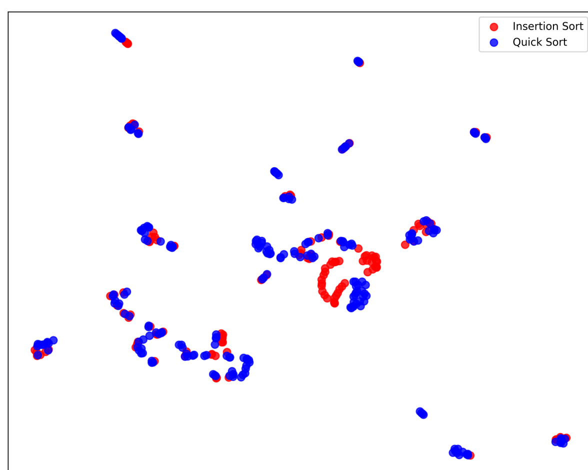
(a) Selection Sort vs. Merge Sort



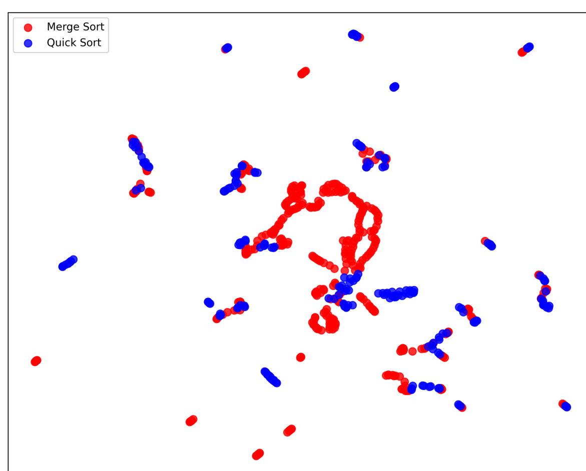
(b) Selection Sort vs. Quick Sort



(c) Insertion Sort vs. Merge Sort

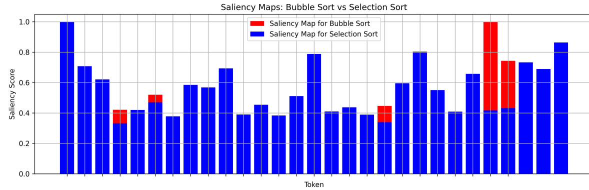


(d) Insertion Sort vs. Quick Sort

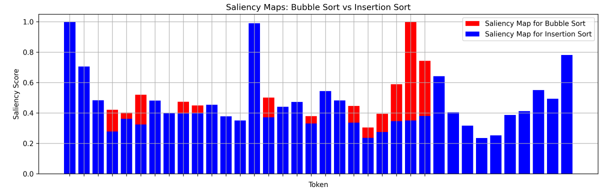


(e) Merge Sort vs. Quick Sort

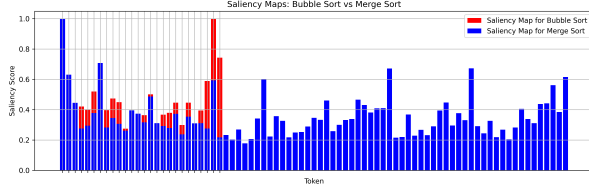
Figure 7: Pairwise comparisons of classical sorting algorithms using UMAP, showing the token-level embeddings in a 2D space (Part 2).



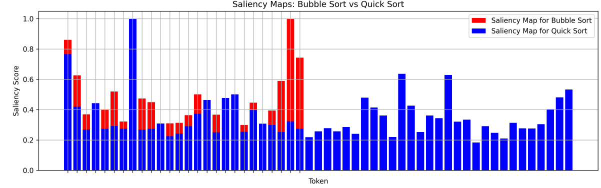
(a) Bubble Sort vs Selection Sort



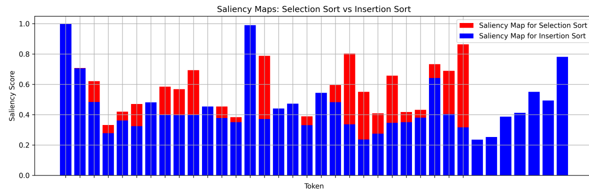
(b) Bubble Sort vs Insertion Sort



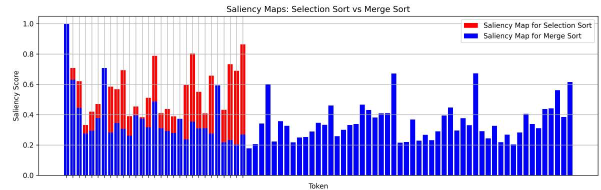
(c) Bubble Sort vs Merge Sort



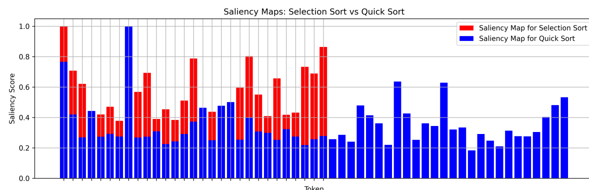
(d) Bubble Sort vs Quick Sort



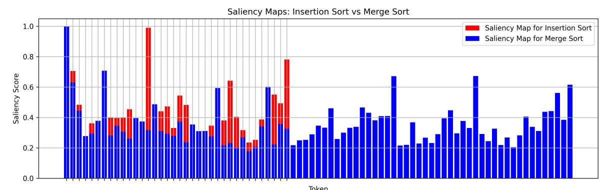
(e) Selection Sort vs Insertion Sort



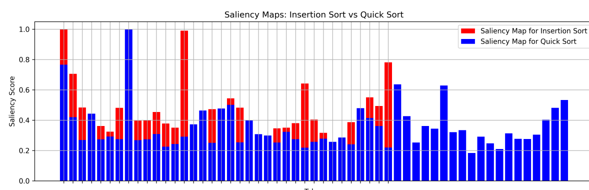
(f) Selection Sort vs Merge Sort



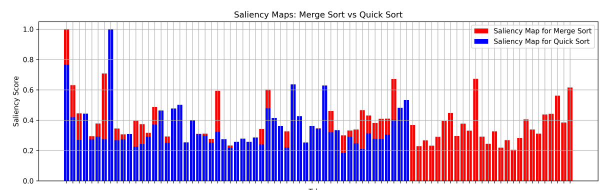
(g) Selection Sort vs Quick Sort



(h) Insertion Sort vs Merge Sort



(i) Insertion Sort vs Quick Sort



(j) Merge Sort vs Quick Sort

Figure 8: Saliency maps for pairwise comparisons of different sorting algorithms.

The results show that this method holds significant potential for various applications. For example, automated code reviews help reviewers quickly identify redundancies, ensuring consistency and reducing maintenance costs. In refactoring, this method can assist developers in identifying code segments that can be optimized. Additionally, in educational settings, this approach can aid in teaching by offering a deeper understanding of the underlying principles of algorithms. Furthermore, the ability to detect code similarities has promising applications in plagiarism detection, where it can identify functionally equivalent code that has been superficially altered to evade detection.

5. Discussion

Our strategy can improve interpretability by offering different approaches to visualize semantic similarities between code fragments through the attention mechanism of GraphCodeBERT. We have seen that this approach improves over traditional syntactic comparison approaches, providing deeper insights into the functional and logical similarities between code fragments.

However, several areas warrant further investigation. One key area is the method's scalability when applied to larger codebases. While our initial experiments focused on relatively small examples, real-world software projects involve thousands or even millions of lines of code. Assessing how well the model performs under such conditions, both in terms of computational efficiency and the quality of the generated outputs, will be crucial.

Another important aspect is the method's applicability across different programming languages. While our experiments showed promising results in single-language comparisons, expanding the range of languages and including those with other structures (e.g., functional versus object-oriented programming) would provide a more comprehensive evaluation 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 provide a more holistic view of code maintainability. Additionally, incorporating feedback mechanisms where developers can annotate or refine the outputs could improve the accuracy and usability of the results.

6. Conclusion

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

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

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 for code similarity detection. However, challenges remain in generalizing across different programming languages and coding styles and addressing these will be a crucial focus of ongoing research efforts.

Acknowledgments

The research reported in this paper 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.

References

[1] Abid, S., Cai, X., & Jiang, L. (2023). Interpreting codebert for semantic code clone detection. In *30th Asia-Pacific Software Engineering Conference, APSEC 2023, Seoul, Republic of Korea, December 4-7, 2023* (pp. 229–238). IEEE. doi:10.1109/APSEC60848.2023.00033.

[2] Ain, Q. U., Butt, W. H., Anwar, M. W., Azam, F., & Maqbool, B. (2019). A systematic review on code clone detection. *IEEE access*, *7*, 86121–86144. doi:10.1016/j.jss.2023.111796.

[3] Bruch, M., Monperrus, M., & Mezini, M. (2009). 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* (pp. 213–222).

[4] Chefer, H., Gur, S., & Wolf, L. (2021). Transformer interpretability beyond attention visualization. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021* (pp. 782–791). Computer Vision Foundation / IEEE. doi:10.1109/CVPR46437.2021.00084.

[5] Devlin, J., Chang, M., Lee, K., & Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. In J. Burstein, C. Doran, & T. Solorio (Eds.), *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)* (pp. 4171–4186). Association for Computational Linguistics. doi:10.18653/v1/n19-1423.

[6] Draganov, A., & Dohn, S. (2023). Unexplainable explanations: Towards interpreting tsne and UMAP embeddings. *CoRR*, *abs/2306.11898*. doi:10.48550/ARXIV.2306.11898.

[7] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages. In T. Cohn, Y. He, & Y. Liu (Eds.), *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020* (pp. 1536–1547). Association for Computational Linguistics volume EMNLP 2020 of *Findings of ACL*. doi:10.18653/V1/2020.FINDINGS-EMNLP.139.

[8] 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. B., Drain, D., Sundaresan, N., Yin, J., Jiang, D., & Zhou, M. (2021). 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.

[9] Martinez-Gil, J. (2014). An overview of textual semantic similarity measures based on web intelligence. *Artif. Intell. Rev.*, *42*, 935–943. URL: https://doi.org/10.1007/s10462-012-9349-8. doi:10.1007/S10462-012-9349-8.

[10] Martinez-Gil, J. (2019). Semantic similarity aggregators for very short textual expressions: a case study on landmarks and points of interest. *J. Intell. Inf. Syst.*, *53*, 361–380. URL: https://doi.org/10.1007/s10844-019-00561-0. doi:10.1007/S10844-019-00561-0.

[11] Martinez-Gil, J. (2022). A comprehensive review of stacking methods for semantic similarity measurement. *Machine Learning with Applications*, *10*, 100423. doi:10.1016/j.mlwa.2022.100423.

[12] Martinez-Gil, J. (2023). A comparative study of ensemble techniques based on genetic programming: A case study in semantic similarity assessment. *Int. J. Softw. Eng. Knowl. Eng.*, *33*, 289–312. doi:10.1142/S0218194022500772.

[13] Martinez-Gil, J. (2024). Advanced detection of source code clones via an ensemble of unsupervised similarity measures. *CoRR*, *abs/2405.02095*. URL: https://doi.org/10.48550/arXiv.2405.02095. doi:10.48550/ARXIV.2405.02095. arXiv:2405.02095.

[14] Martinez-Gil, J. (2024). Improving source code similarity detection through graphcodebert and integration of additional features. *CoRR*, *abs/2408.08903*. URL: https://doi.org/10.48550/arXiv.2408.08903. doi:10.48550/ARXIV.2408.08903. arXiv:2408.08903.

[15] Martinez-Gil, J. (2024). Source code clone detection using unsupervised similarity measures. In P. Bludau, R. Ramler, D. Winkler, & J. Bergsmann (Eds.), *Software Quality as a Foundation for Security - 16th International Conference on Software Quality, SWQD 2024, Vienna, Austria, April 23-25, 2024, Proceedings* (pp. 21–37). Springer volume 505 of *Lecture Notes in Business Information Processing*. doi:10.1007/978-3-031-56281-5_2.

[16] Martinez-Gil, J., & Aldana-Montes, J. F. (2013). Semantic similarity measurement using historical google search patterns. *Inf. Syst. Frontiers*, *15*, 399–410. URL: https://doi.org/10.1007/s10796-012-9404-7. doi:10.1007/S10796-012-9404-7.

[17] Mashhadi, E., & Hemmati, H. (2021). 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* (pp. 505–509). IEEE. doi:10.1109/MSR52588.2021.00063.

[18] Musil, T. (2019). Examining structure of word embeddings with PCA. In K. Ekstein (Ed.), *Text, Speech, and Dialogue - 22nd International Conference, TSD 2019, Ljubljana, Slovenia, September 11-13, 2019, Proceedings* (pp. 211–223). Springer volume 11697 of *Lecture Notes in Computer Science*. doi:10.1007/978-3-030-27947-9_18.

[19] Rai, D., Zhou, Y., Feng, S., Saparov, A., & Yao, Z. (2024). A practical review of mechanistic interpretability for transformer-based language models. *CoRR*, *abs/2407.02646*. doi:10.48550/ARXIV.2407.02646.

[20] Smilkov, D., Thorat, N., Nicholson, C., Reif, E., Viégas, F. B., & Wattenberg, M. (2016). Embedding projector: Interactive visualization and interpretation of embeddings. *CoRR*, *abs/1611.05469*. URL: http://arxiv.org/abs/1611.05469.

[21] Wang, R., Xu, S., Tian, Y., Ji, X., Sun, X., & Jiang, S. (2024). SCL-CVD: supervised contrastive learning for code vulnerability detection via graphcodebert. *Comput. Secur.*, *145*, 103994. doi:10.1016/J.COSE.2024.103994.

[22] Wei, H., & Li, M. (2017). Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *IJCAI* (pp. 3034–3040). doi:10.24963/IJCAI.2017/423.

[23] White, M., Tufano, M., Vendome, C., & Poshyvanyk, D. (2016). Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering* (pp. 87–98). doi:10.1145/2970276.2970326.