

What You See is What it Means!

Semantic Representation Learning of Code based on Visualization and Transfer Learning

Patrick Keller, Laura Plein, Tegawendé F. Bissyandé, Jacques Klein and Yves Le Traon

SnT, University of Luxembourg

firstname.lastname@uni.lu

ABSTRACT

Recent successes in training word embeddings for NLP tasks have encouraged a wave of research on representation learning for source code, which builds on similar NLP methods. The overall objective is then to produce code embeddings that capture the maximum of program semantics. State-of-the-art approaches invariably rely on a syntactic representation (i.e., raw lexical tokens, abstract syntax trees, or intermediate representation tokens) to generate embeddings, which are criticized in the literature as non-robust or non-generalizable. In this work, we investigate a novel embedding approach based on the intuition that source code has visual patterns of semantics. We further use these patterns to address the outstanding challenge of identifying semantic code clones. We propose the WYSiWiM (“*What You See Is What It Means*”) approach where visual representations of source code are fed into powerful pre-trained image classification neural networks from the field of computer vision to benefit from the practical advantages of transfer learning. We evaluate the proposed embedding approach on two variations of the task of semantic code clone identification: code clone detection (a binary classification problem), and code classification (a multi-classification problem). We show with experiments on the **BigCloneBench (Java)** and **Open Judge (C)** datasets that although simple, our WYSiWiM approach performs as effectively as state of the art approaches such as ASTNN or TBCNN. We further explore the influence of different steps in our approach, such as the choice of visual representations or the classification algorithm, to eventually discuss the promises and limitations of this research direction.

KEYWORDS

semantic clones, embeddings, visual representation

1 INTRODUCTION

Semantic code clone identification is a long-standing challenge in software engineering [31]. It has applications in diverse automation tasks, including bug and vulnerability detection, program repair and synthesis, etc. Until recently, semantic clones were reliably identified using dynamic approaches, such as DyCLINK [41], which compare execution traces to decide whether two code fragments behave similarly. Unfortunately, such approaches typically require high-coverage testing to guarantee accuracy. In consequence, they do not effectively scale, and are not usually practical since they require complete and executable code as input. Recent advances in neural networks have provided a new play field for researching

static approaches that attempt to learn semantic representations of code via source code embeddings.

Semantic representation learning of source code has attracted significant attention in the research community in the last couple of years [2, 3, 10, 23, 25, 28, 44–47]. Traditionally, the literature proposes approaches that process code directly or use a syntactic tree representation, where code is treated as sentences. Then, specific approaches inspired by techniques from the Natural Language Processing (NLP) field are used to yield embeddings of these “sentences”. Various works in this realm face robustness issues [9] since simplification of Abstract Syntax Trees (AST), to cope with implementation constraints, weaken the capability of neural network models to capture real and complex semantics [48]. To address these limitations, the state-of-the-art ASTNN [47] approach proposes to split each large AST into a sequence of small statement trees, and recursively encodes the statement trees to vectors by capturing the lexical and syntactical knowledge of statements. Although this approach shows promising results on benchmark samples, its reliance on lexical similarity eventually poses two challenges: (1) the model must be regularly trained on new datasets to allow the inner word2vec [33] model to capture new vocabulary; (2) the model could be misled by relying on lexical tokens, given that two different library methods with the same names may have different semantics implemented outside the code fragment.

In this paper we propose to investigate another representation learning direction for capturing semantics. In contrast to recent works which all focus on lexical and syntactical information to capture semantics, the intuition behind our approach is to mimic the way a human would instantly perceive code. The data is received through visual perception and forms an image in the head of the programmer. This image is then analyzed for structures the programmer has seen before, by applying his experience. From those recognized structures, the programmer may identify patterns of functionality implementations, which would help him to rapidly infer the semantics of the code fragment, leading to a general understanding of the code. We apply the same methodology to design the WYSiWiM (“*What You See Is What It Means*”) approach: instead of directly training a complex and opaque semantic representation or embeddings based on syntactical information in source code, we simply render source code into a visual representation that is natural to code. This step is supposed to model the human visual perception. After the visualization process, WYSiWiM performs two different procedures. First, the visual structures of the code are extracted. To that end, a pre-trained image classification neural network, i.e., a neural network that has been trained on other image classification datasets¹ is used to yield a vector of internal features which represent structural information of the input image (i.e., of

Conference’17, July 2017, Washington, DC, USA
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00
<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

¹The technique of “extracting” knowledge from other datasets is referred to the literature as *transfer learning*

the code). Optionally, the pre-trained network can be re-trained by adding samples of images representing code. Second, the feature vectors are used for learning to discriminate between samples implementing different semantics, just as a human developer would do. Eventually, we expect to leverage the produced classifiers for *code classification* (i.e., given a code fragment, predict its functionality label) and *clone detection* (i.e., given a pair of code fragments, decide whether they are semantic clones).

Our main contributions are as follows:

- We propose a novel approach to semantic representation learning of code based on visual representations of code fragments. The WYSIWIM approach is intuitively simple, and it builds on transfer learning to efficiently produce embeddings by exploiting powerful pre-trained image classification models from the field of computer vision.
- We apply the visual representation embeddings of WYSIWIM to variant tasks of semantic code clone identification. Experimental validations against the BCB and OJ datasets show that WYSIWIM is capable of keeping up with the state-of-the-art while providing significant potential for improvement.
- Finally, we provide an analysis of the influence of some implementation choices. Notably, we discuss the possibilities of visual representations of code and the challenges associated to duplicates in the clone benchmarks as part of threats to validity.

2 BACKGROUND & RELATED WORK

We provide in this section an overview of related work after defining essential concepts to facilitate the readers understanding of the WYSIWIM approach description.

2.1 Definitions

2.1.1 Code clone concepts. We use the following clone-related definitions for our approach in Section 3.

- **Code Fragment:** Also referred to as code snippet, it is a piece of software. Formally, a code fragment is a contiguous set of code lines, which represents the input unit for clone identification. In practice, a code fragment can be a small set of instructions, a whole code block, a whole method or even a whole class.
- **(Code) Clone Pair:** It is a pair of code fragments that are syntactically or semantically similar to each other.
- **Clone Class:** This refers to a set of code fragments where any pairwise combination of code fragments is a clone pair.
- **Syntactic clone pair:** It is a clone pair where the code fragments were deemed similar according to a specific syntactic similarity measure.
- **Semantic clone pair:** It is a clone pair where the code fragments implement the same functionality, respectively the same behaviour or “semantics”.
- **Candidate pair:** It refers to a pair of code fragments that may or may not constitute a clone pair. We use this terminology when we do not want to distinguish, or do not yet know, whether or not a pair of code fragments represents a clone pair.

We also recall for the reader the following well-accepted definitions of clone types [8, 39, 42]

- **Type-1:** Identical code fragments, except for differences in white-space, layout, and comments.
- **Type-2:** Identical code fragments, except for differences in identifier names and literal values, in addition to Type-1 clone differences. They are also called *parameterized* or *renamed* clones.
- **Type-3:** Syntactically similar code fragments that differ at the statement level. The fragments have statements added, modified and/or removed with respect to each other, in addition to Type-1 and Type-2 clone differences. They are also called *gapped* or *near-miss* clones.
- **Type-4:** Syntactically dissimilar code fragments that implement the same functionality. They are also known as *functional* or *semantic* clones. In practice, Type-4 clones are often identified as Type-3 clones with an upper-bound threshold on the syntactic similarity with respect to a specific similarity measure. It should be noted that Type-1 to Type-4 clones are generally considered mutually exclusive.

⇒ Semantic/functional clones are the primary target in this work.

2.1.2 Machine learning concepts. Since Section 3 develops a machine learning approach, we recall for the reader important concepts that we leverage. However, the inner details of these concepts are strictly out of the scope of this work.

Image Classification: Image classification is a well-studied problem in computer vision with several applications such as facial recognition in smart houses, object recognition for self-driving cars, or disease diagnostic in healthcare. The typical task consists in training a model to classify an image into a single or multiple predefined categories [20, 30]. Recent advances in deep neural networks have led to significant breakthroughs in image classification, where computers manage to match human-level accuracy under some conditions [16]. Convolutional Neural Networks (CNNs) is the most popular neural network model being used to address the image classification problem. The general idea behind CNNs is that a local understanding of an image is good enough. A convolution is then a weighted sum of the pixel values of the image, as a sliding window is moved across the whole image. Eventually, the CNNs extract low, middle and high-level features and classifiers in an end-to-end multi-layer fashion, and the number of stacked layers can enrich the “levels” of features. Simply explained, those image classification neural networks learn to recognize visual features from the images, such as structures and colorings.

However, CNNs have been shown to present a degradation problem when the deeper network starts to converge: with the network depth increasing, accuracy gets saturated and then degrades rapidly. Residual networks [16] (ResNets) have then been proposed to overcome this problem by explicitly letting deeper stacked layers to fit a residual mapping (instead of an underlying mapping as in CNNs). In this work, we will build on these tried and true models from the literature.

Transfer Learning: Transfer learning is a technique in machine learning which consists of transferring knowledge from a specific domain to another one. To give a real-world example to the concept, we could imagine that learning to play the piano can help a human to learn to play guitar later on. Even though the instruments are very different, the notes and the rhythms are the same, hence we

can transfer this knowledge from one task to the other and thus reduce the effort to learn.

2.2 Related work

Our work is related to various research directions in the literature, including code clone detection, computer vision, machine learning and software engineering benchmarking.

2.2.1 Code clone identification. Although code clone identification has been largely studied in the literature, relatively few techniques have explicitly targeted semantically similar code fragments. Most approaches indeed focus on textually, structurally or syntactically similar code fragments. The state-of-the-art techniques on static detection of code clones leverage various intermediate representations to compute code similarity. Token-based [6, 21, 27] representations are used in approaches that target syntactic similarity. AST-based [7, 17] representations are employed in approaches that detect similar but potentially structurally different code fragments. Finally, (program dependency) graph-based [23, 29] representations are used in detecting clones where statements may be intertwined with each other. Although similar code fragments identified by all these approaches usually have similar behavior, such static approaches still miss finding such fragments which have similar behavior even if their code is dissimilar [19].

To find similarly behaving code fragments, researchers have relied upon dynamic or concolic code similarity detection which consists in identifying programs that yield similar outputs for the same inputs [18, 22, 24, 26, 42]. Although these approaches can be very effective in finding semantic code clones, dynamic execution of code is not scalable and implies several limitations for practical usage (e.g., the need of exhaustive test cases to ensure confidence in behavioral equivalence).

Conceptually, the closest related work is by Ragkhitwetsagul et al. [38] who developed a syntactic code clone detection approach based on a visual representation of code. In order to visually represent the code, they pre-process the code by removing comments and normalizing the code formatting. The code is then rendered while applying a syntax highlighting, as done in an IDE, to create the code image. This image is then post-processed by applying various simple image transformations, such as blurring, to finally measure the resulting image similarity. The decision of whether or not two code snippets are clones is then based on the level of image similarity between the visual layout. The scope of such an approach is only limited to syntactic clones. Nevertheless, their experiments also show that the visual representation-based method can generally keep up with the state-of-the-art for syntactic clone detection. Although our approach shares the core concept of *visualizing code*, we have a different scope (semantic clones in our case) and we additionally augment this visual representation through transfer learning.

Recently, researchers have investigated leveraging advanced natural language processing and deep learning techniques to statically detect harder-to-detect clones (i.e., type-4 clones). Kim et al. [14] proposed the FaCoY code-to-code search engine where tokens from input code fragments are alternated by considering code fragments from related stackoverflow posts. This enables the search engine to identify syntactically dissimilar code fragments from the

search database. This work, however, is rather competitive to online code search engines than code clone detectors. With their OreO framework, Saini et al. [40] have proposed to use a combination of machine learning, information retrieval, and software metrics to deal with all clone types. They build a specific deep neural network with siamese architecture to address type-4 clones with relative success.

2.2.2 Semantic representation learning. Deep learning advances have been exploited for statically learning semantic representations of code. A prominent work in this direction is the Tree-based convolutional neural network (TBCNN) proposed by Mou et al. [35]. The authors proposed an effective embedding method for programming language processing, and introduced a large dataset of functional clones which is necessary to train and evaluate the task of code classification. More recently, Zhang et al. [47] set the new state-of-the-art representation learning approach with ASTNN, which was demonstrated to be more effective than TBCNN for code clone identification tasks. ASTNN is a semantic embedding method which splits a given code AST into a sequence of smaller statement subtrees and applies a word2vec [33, 34] embedding to those subtrees. This way ASTNN manages to capture both the lexical and syntactical information within code fragments. We consider both ASTNN and TBCNN as the state-of-the-art for semantic clone identification, and thus they will be used as references for benchmarking our WYSIWIM approach.

2.2.3 ResNets. Deep Residual Networks [16] is undoubtedly today one of the most regarded state-of-the-art techniques within the field of computer vision. This neural network architecture allows to create deeper neural networks for image classification while reducing the network complexity in comparison to other deep learning techniques. Experimental data confirmed that the strategy is effective and may lead to human-level accuracy for the task of image classification. Our approach builds on the success of these networks.

2.2.4 Benchmarks. In the code clone identification literature, two main benchmarks are widely used.

- BigCloneBench (BCB), released by Svajlenko et al. [43], is the first big-data-curated benchmark of real clones and used to evaluate modern tools of detecting code clones. It contains 8 million clone pairs and is to the best of our knowledge the biggest publicly available Java code clone benchmarks. It was built by labeling pairs of code fragments from the IJaDataset-2.0 [4]. BigCloneBench maintainers have mined this dataset focusing on a specific set of functionalities.
- OpenJudge (OJ), released by Mou et al. [35], is another public dataset used to evaluate code-clone detection. It is mostly used in the literature for evaluating program classification approaches, although recent works [40, 47] have applied code clone detection approaches to it. The dataset consists of solutions submitted by students to 104 programming questions on OpenJudge², written in C. For each question, there are 500 corresponding solutions, each of which is verified to be correct by OpenJudge and are thus considered as clones.

²<http://poj.openjudge.cn/>

3 WYSIWIM

In this section, we will overview the design of WYSIWIM, providing details on the considered visualizations and the learning models.

3.1 Approach overview

The core of the WYSIWIM approach is about the production of embeddings for a given **code fragment**. The idea is to take a code fragment and produce a vector of real numbers so that we receive an actionable representation of the embedded semantic information. As illustrated in Figure 1, we consider a deep feature extractor which works by producing embeddings for image renderings of code fragments.

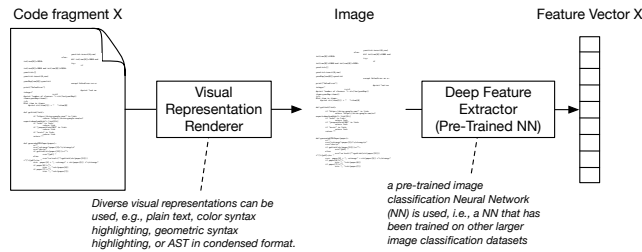


Figure 1: Deep feature extraction (a.k.a, visualization-based code embedding)

Building a deep feature extractor requires a training step based on a large dataset of images. During such a training, the neural networks learn suitable representations for the images within a feature space. Given that deep neural network architectures for image classification are known to capture a large number of structural features of images, we postulate that pre-trained models can be explored in a transfer learning scenario (cf. Section 3.2). Transferring the knowledge, embedded in those pre-trained models, allows us to extract visual features without the need of huge amounts of task-specific data to train the feature extractor.

Once the feature extractor is obtained, one can feed code rendered as images into it to collect the resulting feature vectors. Those can further be used to train simple binary classifiers that learn to apply the embedded semantic information. Simply put, the deep neural network is used to preprocess images so that they can be used to learn semantics by applying well-known classical machine learning algorithms.

Clone identification tasks. In this work we apply the WYSIWIM approach of visualization-based code semantics learning to the problem of clone identification, which is approached in two different ways: as a classical *code clone detection* problem and as a *code classification* problem.

- ★ In code classification, the goal is to predict the functionality implemented by a code fragment. In practice, we must learn to map the code fragment to one of a set of predefined semantic functionality labels (i.e. *clone classes*). It is thus a multi-class classification problem that takes a single code fragment as input and outputs a functionality label.

- ★ In clone detection, the goal is to directly decide if two code fragments are clones. It is thus a binary classification problem that

takes a pair of code fragments as input and outputs a Yes/No label on whether or not those fragments form a clone pair.

In principle, both tasks can be emulated by one another. On the one hand, the code classification task could be emulated by finding all clone pairs and building their transitive closure to generate the semantic clone classes. On the other hand, the clone detection task, could be emulated by directly comparing the code fragment labels. We have nevertheless opted in this work to build two separate workflows, both starting by first converting code fragments into their visual representations.

- ★ For code classification, the collected code “images” and their associated functionality labels are used to **fine-tune a pre-trained image classification network**. To that end, the **size of the output layer** of the pre-trained image classification network must be updated. Indeed the output layer nodes map to the classes that are seen during training. With new datasets, new classes appear.

- ★ For clone detection, the collected code “images” are **directly fed into a pre-trained image classification network in order to retrieve the corresponding embeddings** (which are numerical vectors representing the internal structural features within images). Obtained feature vectors are then used for training and testing a classical binary classifier.

3.2 Transfer learning from pre-trained models

In our approach, we transfer the embedded knowledge of the pre-trained image classification neural networks to our clone identification tasks (i.e., for both code classification and clone detection). The knowledge that is transferred in our case is *the ability to recognize visual patterns and structures from images*. Even though the data that those networks are trained on belong technically to a different domain, we expect that they still capture relevant structural knowledge that can be reused to extract the structural information from our specialized (code visualization) images. Thus, our hypothesis here is that, through the transfer learning, we can leverage powerful pre-trained networks which are able to effectively embed meaningful syntactic as well as semantic structures [16].

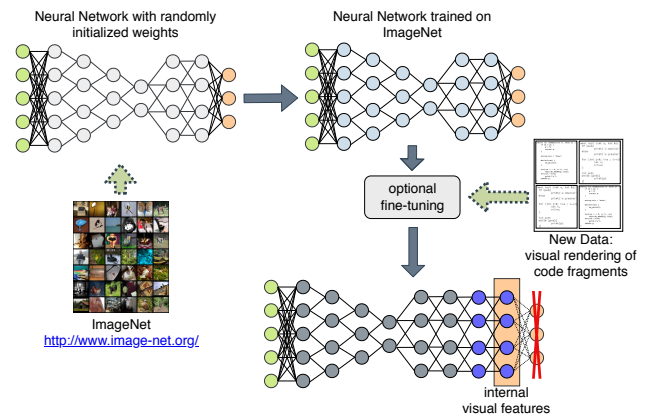


Figure 2: Principle of transfer learning applied to build our deep feature extractor for code

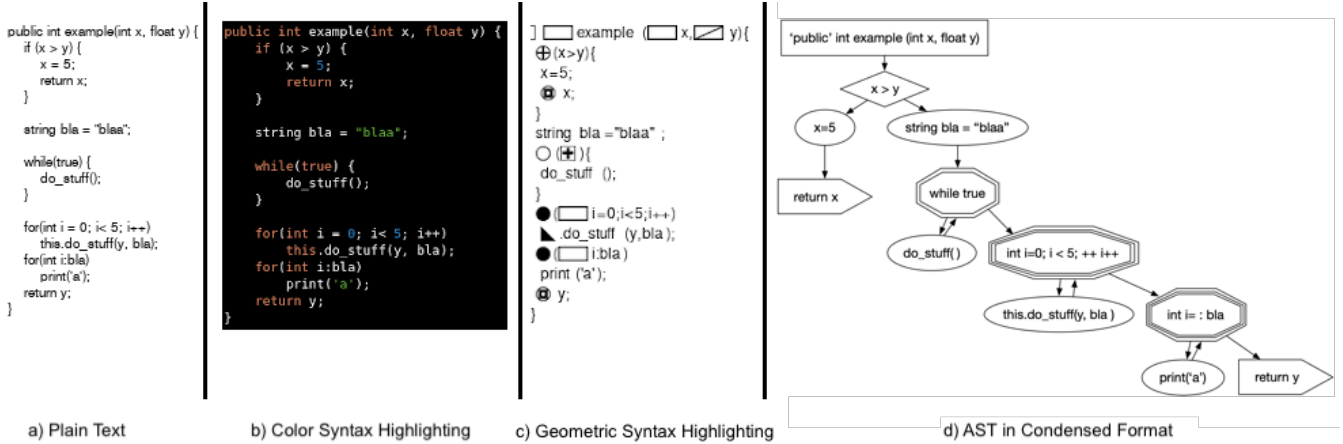


Figure 3: Variations of visual representations of code

Image classification neural networks consist of a multitude of convolutional layers that all learn different combinations and variations of the data contained in the previous layers. In addition, the networks have an input layer which accepts the input data and a fully connected output layer. This final layer is usually sized according to the number of possible labels and is in charge of deciding a label for the data coming from the previous layers. In our case, as depicted in Figure 2, we focus on retrieving the intermediate features that are accessible in the penultimate layer. Actually, these features could have been collected on any previous layers. For the sake of prototyping speed, we immediately accessed the readily-available features. Future work could investigate other layers.

It should be noted that transfer learning is gaining traction within the deep learning community, since several domains lack sufficient data for training [36]. Therefore, a fundamental motivation in the study of transfer learning is the fact that people can intelligently apply knowledge learned previously to solve new problems faster or with better solutions. For example, it has been shown possible to use the knowledge about notes and rhythm, which were learned for playing the piano, to learn guitar playing; applying the vocabulary learned in French to infer English words as they share a certain base; or in audio-visual correspondence tasks [5].

3.3 Visualization options

We explore in WYSIWIM four variations of code visualizations in order to assess the influence of the selected visual representation on the performance of WYSIWIM. We describe each visual representation by explaining its principle, detailing its implementation and arguing about its relevance.

- **PLAIN Text:** The first visual representation is straightforward. It consists of simply rendering the textual representation of the code as a black and white image without highlighting any language construct. The rendering is implemented using the *pillow*³ Python image drawing and manipulation library: source code text is rendered as-is, i.e., with the indentations used by the developer, while

applying a white background. PLAIN TEXT, illustrated in Figure 3(a), is considered as our baseline visual representation of code.

- **COLOR Syntax Highlighting:** A simple variation of the PLAIN TEXT visualization consists in rendering the code text while highlighting syntax with colors, similarly to what is done in programming environments. This rendering approach is implemented by first generating an html page to highlight the code using the *google code-prettify* javascript library. The web page is then saved as a PNG image using the *imgkit*⁴, a python wrapper for the Webkit web browser engine. As illustrated in Figure 3(b), this visual representation is expliciting code structures for human programmers. Therefore, we expect that color-based syntax highlighting can be relevant for semantic machine learning tasks.

- **GEOMETRIC Syntax Highlighting:** In the previous visualization option, emphasis is put on color. Yet, image classification neural networks are also known to capture shapes. We propose to build a rendering of code where language keywords are represented by specific geometric shapes (i.e., icons). The implementation is based on the tokenization of code fragments using the *javalang*⁵ python library. We preset the mapping of language keywords with specific icons. During rendering, the text tokens are then replaced by the associated icons. Overall, although this representation could be nonsensical for humans, we expect that it will support the learning algorithm in the same way colored syntax does for programmers visual perception of code.

- **AST in Condensed Format:** Finally, we consider a visual rendering of the abstract syntax trees. The implementation is based on the AST generated by the *javalang* python library and leveraging *graphviz* python bindings⁶. To render the resulting graph, we generate a "graphviz" graph model by traversing the AST and representing some subtrees (e.g., the "for" loop control) in a purely textual manner, while representing other elements as their actual tree structure. This helps to condense the AST since raw AST quickly explodes in depth and breadth even for small code fragments. In this representation we generated the graph such that the edges

⁴<https://pypi.org/project/imgkit/>

⁵<https://pypi.org/project/javalang/>

⁶<https://www.graphviz.org/>

³<https://pillow.readthedocs.io>

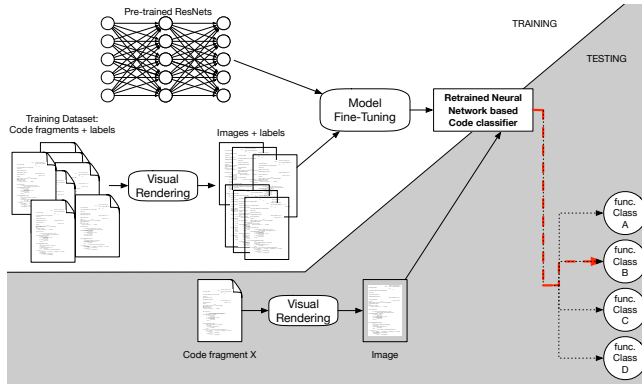


Figure 4: Illustration of the architecture for WxSiWiM's Code classification

represent the possible control flows inside the code in order to capture its sequential nature. Further, we apply some geometric shapes to specific types of nodes in order to augment the visual strength of specific code structures.

Overall, we try in WxSiWiM visualization options that emphasize on colors, shapes and structures, and compare against the baseline plain text rendering. Although the generation of visual renderings is stable (i.e., not a random process), it should be noted that the AST in condensed format is, by far, the slowest to compute, as it involves many complex steps.

Concretely, the output of the visualization rendering process is a single PNG image per visualization option and per code fragment. Each image may also be re-scaled to fit with the input requirements of the pre-trained image classification neural network.

3.4 Code classification architecture

Figure 4 provides a simple illustration of the overall architecture that we developed for code classification. We leverage neural networks (specifically, the powerful ResNets) that are pre-trained on the ImageNet dataset [13]. However, we perform a re-training step, which is actually aimed to fine-tune the neural networks, towards better learning to extract features that are semantically-relevant to different classes of code functionalities. To that end, we update the size of the output layer of the pre-trained image classification network so that the final size accounts also for the number of possible functionality labels in our code dataset. The re-trained (i.e., fine-tuned) network on the training dataset is then used as classifier to predict the labels of code fragments in the test set based on their visual renderings. Since we do not tune any hyper-parameters of the network, we do not need a validation set. Nevertheless, we ensure that the process of re-training is performed for an empirically determined number of epochs⁷. Details on how training and test sets are split are provided and discussed later in Section 4.2.

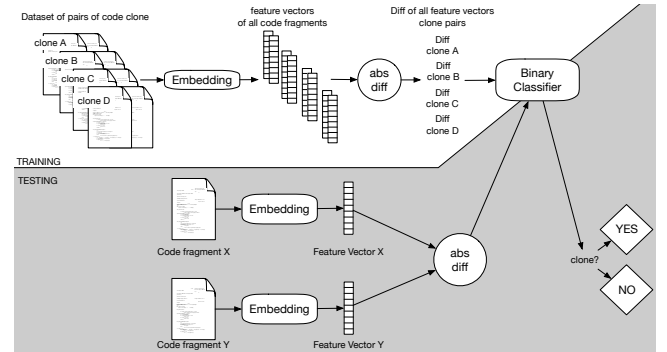


Figure 5: Illustration of the architecture for WxSiWiM's Clone detection

3.5 Clone detection architecture

Figure 5 illustrates WxSiWiM's architecture for clone detection. Similarly to the pipeline of code classification, we leverage a pre-trained neural network for visual classification to which we feed the images obtained from visual renderings of code snippets. In this case, however, our objective is to simply collect the embeddings produced during deep feature extraction. Thus, given that we do not need the network to learn about new classes in our new (code-related) image datasets, we propose to directly use the ResNets that were pre-trained on ImageNet datasets. We expect the embeddings to still be relevant for capturing structural features. The feature vectors (i.e., embeddings) are then used to train binary classifiers as in traditional machine learning. Concretely, to train the binary classifiers, the first step is to calculate the absolute difference between the feature vectors of the candidate pair vectors. Those difference vectors can then be used to train our binary classifiers.

Algorithms for binary classification. In code classification, we directly reuse the in-built capability of the neural networks to perform classification (i.e., using the *softmax* activation function at the last layer). Indeed, given that the input of the task is a single image representing the visualization rendering of a code fragment, the classical image classification neural network is suitable.

In clone detection, however, the input is a pair of code fragments (precisely, a pair of images taken from their visualization renderings). This means that the architecture of image classification networks is not readily applicable for this case as it always expects a single input only. For sake of simplicity and optimization, we decided to use the neural network to collect embeddings for individual images, and train our final classifier separately. This strategy allows us to experiment with different traditional classification algorithms. Our experiments provide results with Support Vector Machines [11], k-Nearest Neighbours [12] and a simple binary classification neural network [15].

⁷We ran a few experiments to check when the results stabilize.

4 EXPERIMENTAL SETUP

We enumerate the research questions, overview the datasets used in the experiments and discuss some important implementation details. We open-source the implementation of our prototype implementation of WYSIWIM and release all data related to the experiments recorded in this paper. The artifact web page is currently in an anonymous repository: <https://github.com/wysiwiw/wysiwiw>

4.1 Research Questions

RQ1: How does WYSIWIM perform in comparison with the state-of-the-art? We investigate the ability of our novel approach of semantics learning based on visual representation of code to keep up with the state-of-the-art for the tasks of code classification and clone detection.

RQ2: How does the visual representation influence the performance of WYSIWIM? Experiments for this research question are focused on the code clone detection task, where we try all the considered visual representations options and compare the performance differences.

RQ3: What is the impact of the classification algorithms on WYSIWIM? We investigate in this research question different supervised learning algorithms that can be leveraged to train the binary classifiers needed for the code clone detection architecture.

4.2 Selection of datasets

Code Classification: We assess the performance of WYSIWIM for the code classification task based on the **Open Judge (OJ) dataset** as introduced in [35]. This choice is motivated by the need to directly compare against the state-of-the-art (namely, TBCNN [35] and ASTNN [47]), which also run experiments on this dataset. This dataset contains 104 different functionalities and 500 samples per functionality. In order to achieve balanced datasets for training and testing, we apply a stratified sampling over the functionalities with a ratio of 4:1 (i.e., 80% of data for training and 20% for testing).

Code Clone Detection: The state-of-the-art for code clone detection being ASTNN [47], we reuse the dataset that they release in their experiment artifacts. This enables a direct and unbiased comparison. The split into training and testing sets is also predefined and applied as-is. This dataset consists of 20k Type-4 clone pairs and 20k non-clone pairs.

Nevertheless, we found that the ASTNN dataset is not balanced with respect to the number of clones per functionality. Thus we selected a custom subset of BigCloneBench (BCB) (cf. Section 2.2.4) for our further experiments. We focus on code fragments related to three functionalities (i.e., #7 - bubble-sort array #13 - shuffle array inplace and #44 - check for palindrome) which we consider the most suitable for our evaluation: these code fragments are dissimilar enough but concise; furthermore #7 and #13 code fragments deal all with arrays and yet semantically distant, offering an opportunity to properly assess the semantic clone detection approach. The dataset is constructed by randomly sampling 500 Type-4 clone pairs and 500 non-clone pairs per functionality. The ground truth information of clone/non-clone is based on the annotations provided in BigCloneBench.

4.3 Implementation

Our proof-of-concept implementation of WYSIWIM is written in Python using common frameworks and libraries. In particular, several Python libraries are leveraged for the code fragment processing towards producing visual renderings as images (cf. Section 3.3). We leverage the *pandas*⁸ library for data management. Further for the stratified splitting of the datasets, we use the dataset splitting method from the *scikit-learn*⁹ library.

Data Preprocessing. The pre-trained networks considered in our experiments have a limitation on the input image size being set to exactly 224x224 pixels. To fit with this requirement, we choose to simply re-scale the code visualizations to this size.

Data Augmentation. Usually in many machine learning applications, a data augmentation step is performed. In image classification in particular, one usually uses a set of random transformations to create many variations from the input data in order to artificially increase the size of the dataset. Those random transformations include rescaling, cropping, mirroring etc. For our approach however, we found that this is not beneficial since source code naturally does not appear up-side down or mirrored.

Code classification. The implementation of our code classification task is mainly based on PyTorch [37] and uses the pre-trained ResNet models provided by the PyTorch framework. In particular we use a ResNet18 and a ResNet50 to highlight the increase of performance when the number of layers is increased. Both are pre-trained on the ImageNet dataset [13].

Clone detection. For the implementation of the binary clone detection task, we use again the pre-trained ResNet50 model and drop the last layer in order to generate the raw feature vectors for our visualized code fragments. Those vectors are then converted into *numpy*¹⁰ arrays which facilitates the calculation of the absolute difference between vectors. For the final stage of learning and predicting, we use *pytorch* again to implement a simple binary classification network. SVM and k-NN algorithm implementations are taken from the *scikit-learn* library.

5 RESULTS

We now present the experimental results in response to the research questions, and based on the experimental settings presented previously.

5.1 RQ1: [Performance of WYSIWIM]

5.1.1 Code classification. For performance comparison against the state-of-the-art for the task of code classification, we focus on the **accuracy** metric, which is used by the state-of-the-art ASTNN and TBCNN authors to report their performance (see. [35, 47]). As discussed previously, we also reuse the same OJ dataset that was used for ASTNN and TBCNN validation. We apply the PLAIN text visualization to render code.

Results: Table 1 provides the accuracy metrics of different approaches. WYSIWIM provides an accuracy of 89.7 and 86.4 percent

⁸<https://pandas.pydata.org/>

⁹<https://scikit-learn.org/>

¹⁰<https://www.numpy.org/>

for code classification on the OJ dataset with ResNet18 and ResNet50 respectively. These results suggest that we perform reasonably well in comparison to the state-of-the-art which are reported to yield accuracy scores of 94.0% and 98.2% for TBCNN and ASTNN respectively. Given the limitations that our implementation carries (cf. Section 6.2) and the potential for improvement (cf. Section 6.3), we view this performance of WYSiWiM as largely promising.

Method	Variation	Accuracy
TBCNN	-	94.0
ASTNN	-	98.2
WYSiWiM	with ResNet18 pre-trained model	86.4
WYSiWiM	with ResNet50 pre-trained model	89.7

Table 1: Accuracy comparisons for code classification.

5.1.2 Clone Detection. Experiments for Clone detection are done with the BigCloneBench which already have labels on pairs of clones and non-clones. For fair comparisons, we run ASTNN and WYSiWiM on the same samples of Type-4 clones that were used to evaluate ASTNN by Zhang et al. [47]. For this experiment, we present the results for our best configurations: the AST in condensed format as the visualization option and the neural network binary classification algorithm. We refer the reader to next experiments where we show that the visualization and algorithms have a limited impact on the performance of WYSiWiM. Finally, contrary to previous experiments, we do not compare against TBCNN since this approach has not been applied for clone detection.

Results: The results provided in Table 2 show that, overall, we perform similarly well as the state-of-the-art in terms of F-Measure. It is further noteworthy that WYSiWiM offers a better trade-off between precision and recall than ASTNN which present quasi-perfect precision but lower recall.

Method	F1 score	Precision	Recall
ASTNN	93.7	99.8	88.3
WYSiWiM	94.8	95.4	94.3

Table 2: Performance comparison for clone detection.

5.2 RQ2: [Visualization influence]

To examine the influence of visualization rendering options, we consider the clone detection task where WYSiWiM implements the binary neural network classifier for the final clone decision. The process is then performed for all previously-described visual representations options (cf. Section 3.3).

Results: The results depicted in figure 6 suggest that the AST in condensed format and the COLOR syntax highlighting visual representations yield the best results (which are further similar for these two representations).

On the one hand, it is noteworthy that the COLOR syntax highlighting improves over the PLAIN text visualization, hence confirming our initial intuition that colors can help to better capture semantics visually. On the other hand, although GEOMETRIC syntax highlighting performs slightly less well than others, it's relatively high performance indeed suggests that visual shapes are expressive enough to help learn semantics of code structures. In any case, we

also suspect that the performance degradation of GEOMETRIC syntax highlighting visualizations might emerge from a bad choice of the keyword substitution shapes. Finally, we note that, depending on the performance metric, any of the visualizations may perform better or worse than other visualizations.

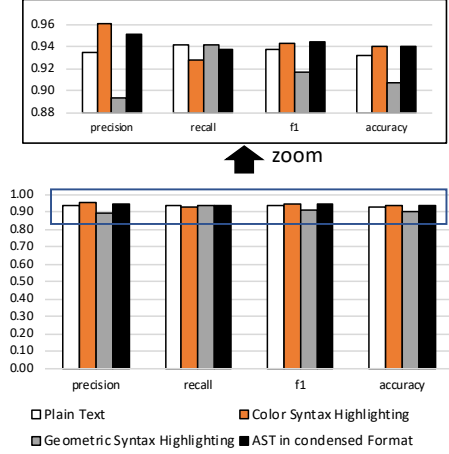


Figure 6: Influence of visual rendering schemes on clone detection performance.

5.3 RQ3: [Algorithm impact]

To run several experiments of clone detection while varying the classification algorithms, we leverage our main dataset sampled from BCB (cf. Section 4.2). We also fix the visualization option to the AST condensed format. The experiments are then performed to compare the variations of sensitivity of the WYSiWiM embeddings with respect to different algorithms. We use kNN¹¹, a simple NN¹², and SVM.

Results: Figure 7 presents the comparison results. It appears that the algorithm has a slight impact on recall scores between kNN and the Neural Network classifier, while they yield the same precision. In contrast the precision of SVM is lower by 8 percentage points. Nevertheless, all three algorithms offer reasonably good performance, which suggests that the embeddings produced by the pre-trained models are effective in terms of semantic representations.

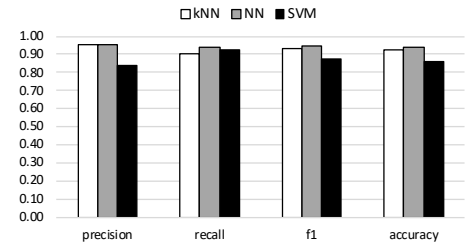


Figure 7: Impact of classification algorithms on clone detection performance

¹¹We use kNN with the default setting of scikit-learn where $k = 5$

¹²The NN is a simple fully-connected linear layer with bias, so in essence a linear combination

¹³we use the default Support Vector classifiers implementation in scikit-learn without tuning any parameters

6 DISCUSSION

Our experimental evaluation bears some threats to validity, while the approach itself has limitations that can be improved in future work.

6.1 Threats to Validity

Internal Validity - Dataset. Our dataset selections are limited in terms of size and diversity of functionalities. For the clone detection variant in particular, even though the number of clones is rather high, the number of code fragments that the clone and non-clones pairs are based on is still very small as the pairs are formed from those base code fragments by pairwise combination. This lack of diversity might negatively influence the generalizability of the evaluation results. Nevertheless, we mitigate this threat in the comparison experiment (RQ1) by using the same datasets as the state-of-the-art (i.e., ASTNN).

External Validity - Dataset. Even though the BigCloneBench is widely used throughout the literature, the judgment of whether or not a pair of code fragments form a clone remains biased and purely based on benchmark authors' intuition. Further, there is no single or precise notion of what semantic similarity is. Thus, the semantic boundaries of the functionality classes might not be consistent across all the represented functionalities. Finally, the program semantics of a code fragment might be obscured by the usage of external libraries that are not included within the dataset, in which case the decision task is technically unfeasible.

External Validity - Presence of clone duplicates in BCB. During development, we noticed the existence of conceptually duplicated clones in BigCloneBench. This fact showed up in the form of identical visual representations of code for different code fragment ids. It turned out that those fragments emerged from Type-1 clone pairs, which are technically the same code. When those both clone fragments are combined with another code fragment to form clone pairs, those clone pairs are conceptually duplicated. Although we cannot provide precise statistics on the extent of clone duplicates present in BigCloneBench, we can approximate, based on the code fragments that are used in Type-1 and Type-4 clone pairs, an upper-bound of approximately 30 percent clone duplicates. If we consider that Type-2 and Type-3 code snippets can also build clone duplicates, this estimation goes up to even 60 percent. In our case, during the development, we experienced drops of performance of about 10 percent, on small development examples. Hence we conclude that code clones should not be disregarded if precise and valid evaluations are desired. This conclusion is consistent with recent empirical results reported by Alamanis on the adverse effects of code duplication in machine learning models of code [1].

To explore the impact of code duplicates on the performance of WYSIWIM, we build a dataset (based on the same three functionalities and numbers of clones/non-clone pairs) where we do not use any clones that contain code fragments that are also used in Type-1 clone pairs. The results from figure 8 show that the avoidance of clone duplicates slightly degrades the overall results. This makes sense since the existence of clone duplicates makes the task easier and allows to achieve a higher score. This finding is further valid for both the best-performing algorithm (NN) and the worst-performing one (SVM).

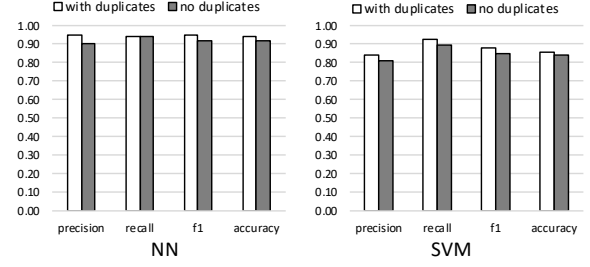


Figure 8: Influence of clone duplicates.

Construct Validity - Dataset. A recurrent construct validity issue in the machine learning literature is related to class imbalance. In clone detection, one must ensure that all functionalities are balanced in the dataset of clone and non-clone pairs. Some approaches may overfit to specific (and largely represented) classes. To check for this issue, we build a balanced dataset (with and without duplicates) and compared the performance of WYSIWIM clone detection on this dataset as well as the imbalanced dataset provided in ASTNN artifacts. Indeed, the ASTNN dataset is randomly sampled from the BCB (using a fixed random seed) and hence -more or less- keeps the unbalancing that is present in the BCB itself. Comparison results in Figure 3 with balanced and imbalanced (i.e., ASTNN dataset) suggest that WYSIWIM keeps its promises on performance.

Dataset	Accuracy	F1 score	Precision	Recall
ASTNN	91.8	94.8	95.4	94.3
balanced	94.1	94.5	95.2	93.8
balanced w/o duplicates	92.1	92.0	94.1	90.1

Table 3: Impact of class imbalance in the dataset of code clones

Construct Validity - Cross validation. We did not perform any cross validation on our approach, as our goal was to rather convey the concepts behind the approach rather than achieve high results. It is probable that the exact results vary to a certain extent on different splits of the dataset, especially since the different code fragments are probably not "semantically equally diverse" to each other, without further specifying what that could mean.

Conclusion Validity - Lack of definitions of semantic similarity. The software engineering community faces a crucial challenge for defining what semantic similarity means. Since we do not dare to explicitly define what semantic similarity means, we have to rely on the semantic value that is embedded in our dataset, respectively as it was implied by the creators of the BCB. In consequence, a specific selection of a subset of the dataset may even influence the overall semantics it carries. However, even when two approaches are applied on the same dataset, they might still view semantic similarity differently. These facts make it hard to evaluate and especially compare semantic approaches of any sort.

6.2 Limitations

Input size of ResNets. Image classification networks have technically and by construction a strong limitation on their input size. This is problematic as it introduces loss and distortion of our input

data. In consequence, we may completely loose the fine-grained lexical information that is contained in the visual representations of our code fragments.

Code fragment granularity. The approach as presented is mainly designed to work with method granularity code fragments. Image classification networks are designed to assign a single most suitable label to a whole single input. This is consistent with generally accepted good coding style rules, which claim that a single method should always implement a single functionality (known as the single responsibility principle) [32]. To enlarge the scope of the granularity, our core concept of code visualization could be leveraged to full programs by applying object localization instead to detect what functionalities a software is composed of. This principle could also explain why our approach works slightly less well on the OJ dataset, which consists of whole programs, while the BigCloneBench rather provides method level granularities.

Colors in visualizations. Our visualizations apply colors only very sparsely, in the case of the color syntax highlighting variant, or not at all for the other visual representations. The current implementation of WYSIWIM is thus not fully leveraging the potential of ResNet, which is designed to operate on all 3 color channels.

Traditional classification algorithms. For the clone detection task we apply very basic binary classification algorithms. These algorithms do probably not explore all semantics learned by the ResNet deep feature extractor.

Scope of the clone datasets. The datasets are not only a threat to validity but also a major limitation. Our hypothesis is that, due to the limited variety and size of the datasets available today, it is not possible yet to learn general semantic knowledge that can be applied to all possible data.

6.3 Lessons learned and Future work

As the current implementation of WYSIWIM represents only a proof-of-concept with limited goals, it offers a lot of potential for extensions and improvements. Furthermore, the general concept of visualizing code and learning on those visual representations could be interesting also to other software engineering tasks, or could be combined with existing approaches. Beyond our approach, we identified some general current limitations on the task of semantic code clone detection, such as the lack of suitable datasets and benchmarks but also the lack of more precise and actionable definitions of semantics or semantic similarity.

Mitigating Image classifier input limitation. As mentioned in the previous sub-section, a major limitation of our approach is the fixed input image size of the ResNet classifier. One potential way to mitigate this limitation could be to slice the image into multiple images of the required input size. Those slices could then be used to generate a larger feature vector, representing the whole image. This would allow to capture more fine-grained information as well. Of course, it might be necessary to apply also scaled versions of the images to capture large-scale structural information too.

Visualizations. As our visualizations showed, the use of colors can have a positive effect on the results. However, as our condensed

AST visualization yielded the best overall results, it might be interesting to further apply color coding on ASTs to make better use of the full potential of the image classification neural networks.

Datasets and Benchmarks: A future work that is important beyond our approach is the development of datasets and benchmarks that are more suitable for semantic code clone detection and semantic approaches in general. This includes a high number of different functionalities and a high number of diverse code examples per functionality. Especially sets providing a multitude of more basic functionalities that do not depend on external libraries would be desirable. They would allow to learn models the way humans learn semantics of computing languages, by starting very small.

Data augmentation: Similarly to data augmentation done in image classification via generating variant images through rotation, cropping, etc., we could envision to apply a data augmentation, although at the meta level, such as mutating the code in semantically-equivalent ways in order to increase the size of our dataset.

Actionable definitions of semantics (similarity): Another very important future work would be to make efforts towards actionable definitions of semantics, or semantic similarity. A possible approach to this could be the definition of semantics through software tests. As software tests represent an executable variant of software specifications, they give a good notion of the requirements we put into our semantics. Of course, there are a few problematic aspects in this approach. One aspect is that each application may require different abstractions of a certain functionality. Another aspect is that the code snippets for a certain functionality would all have to use test suites.

7 CONCLUSIONS

We presented a novel direction to code semantics learning based on visualization and transfer learning. WYSIWIM exploits the power of pre-trained ResNets to extract deep features from visualization renderings of source code fragments. We apply this approach to two variants of clone identification, namely code classification and clone detection. Experimental results on BigCloneBench and the Open Judge datasets show that our approach performs reasonably well and can keep up with the state-of-the-art within the scope of our experimental settings (which we carefully design to be comparable to literature experiments). Our experiments reveal that visualizations of AST yield the best overall clone detection results. We complete the paper by enumerating a list of limitations, which, if resolved, may unleash a huge potential of WYSIWIM beyond clone identification tasks.

Availability: All experimental data as well as the source code of WYSIWIM is open sourced in an anonymous repository:

<https://github.com/wysiwm/wysiwm>

REFERENCES

- [1] Miltiadis Allamanis. 2018. The Adverse Effects of Code Duplication in Machine Learning Models of Code. *arXiv preprint arXiv:1812.06469* (2018).
- [2] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).
- [3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 40.

- [4] Ambient Software Evoluton Group. 2013. IJaDataset 2.0, <http://secold.org/projects/seclone>.
- [5] Relja Arandjelovic and Andrew Zisserman. 2017. Look, Listen and Learn. In *The IEEE International Conference on Computer Vision (ICCV)*.
- [6] B.S. Baker. 1992. A Program for Identifying Duplicated Code. In *Computing Science and Statistics: Proceedings of the 24th Symposium on the Interface*, Vol. 24. 49–57. Issue Mar.
- [7] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*. IEEE, 368–377.
- [8] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering* 33, 9 (2007), 577–591.
- [9] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoeffer. 2018. Neural code comprehension: a learnable representation of code semantics. In *Advances in Neural Information Processing Systems*. 3585–3597.
- [10] Zimin Chen and Martin Monperrus. 2019. A Literature Study of Embeddings on Source Code. *arXiv preprint arXiv:1904.03061* (2019).
- [11] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine learning* 20, 3 (1995), 273–297.
- [12] Thomas M Cover, Peter Hart, et al. 1967. Nearest neighbor pattern classification. *IEEE transactions on information theory* 13, 1 (1967), 21–27.
- [13] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li-Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.
- [14] FaCoY. 2017. <https://github.com/facoy/facoy>.
- [15] Simon Haykin. 1994. *Neural networks: a comprehensive foundation*. Prentice Hall PTR.
- [16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.
- [17] Lingxiao Jiang, Ghassan Mishserghi, Zhendong Su, and Stephane Glondou. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 96–105.
- [18] Lingxiao Jiang and Zhendong Su. 2009. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 81–92.
- [19] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. 2010. Code similarities beyond copy & paste. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. IEEE, 78–87.
- [20] Pooja Kamavisdar, Sonam Saluja, and Sonu Agrawal. 2013. A survey on image classification approaches and techniques. *International Journal of Advanced Research in Computer and Communication Engineering* 2, 1 (2013), 1005–1009.
- [21] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilingual token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.
- [22] H. Kim, Y. Jung, S. Kim, and K. Yi. 2011. MeCC: Memory comparison-based clone detector. In *Proceedings of the 33rd International Conference on Software Engineering*. IEEE, 301–310.
- [23] J. Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*. 301–309.
- [24] D. E. Krutz and E. Shihab. 2013. CCCD: Concolic code clone detection. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. 489–490.
- [25] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. Ccleaner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 249–260.
- [26] Sihan Li, Xusheng Xiao, Blake Bassett, Tao Xie, and Nikolai Tillmann. 2016. Measuring Code Behavioral Similarity for Programming and Software Engineering Education. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, New York, NY, USA, 501–510.
- [27] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: a tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*. USENIX Association, Berkeley, CA, USA, 20–20.
- [28] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).
- [29] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. 2006. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, NY, USA, 872–881.
- [30] Dengsheng Lu and Qihao Weng. 2007. A survey of image classification methods and techniques for improving classification performance. *International journal of Remote sensing* 28, 5 (2007), 823–870.
- [31] Andrian Marcus and Jonathan I Maletic. 2001. Identification of high-level concept clones in source code. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE, 107–114.
- [32] Robert C Martin. 2000. Design principles and design patterns. *Object Mentor* 1, 34 (2000), 597.
- [33] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
- [34] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.
- [35] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI Conference on Artificial Intelligence*.
- [36] Sinno Jialin Pan and Qiang Yang. 2009. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2009), 1345–1359.
- [37] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. 2017. Pytorch. *Computer software. Vers. 0.3.1* (2017).
- [38] Chaoyong Ragkhitwetsagul, Jens Krinke, and Bruno Marnette. 2018. A picture is worth a thousand words: Code clone detection based on image similarity. In *2018 IEEE 12th International Workshop on Software Clones (IWSC)*. IEEE, 44–50.
- [39] Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (2009), 470–495.
- [40] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. 2018. OreO: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 354–365.
- [41] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. 2016. Code relatives: detecting similarly behaving software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 702–714.
- [42] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. 2016. Code Relatives: Detecting Similarly Behaving Software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, 702–714.
- [43] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 476–480.
- [44] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep learning similarities from different representations of source code. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 542–553.
- [45] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code.. In *IJCAI*. 3034–3040.
- [46] Shuang Liu Lin Yang Sang Wei Yi Gao, Zan Wang and Yuanfang Cai. 2019. TECCD: A Tree Embedding Approach for Code Clone Detection. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE.
- [47] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 783–794.
- [48] Xiaodan Zhu, Parinaz Sobhani, and Hongyu Guo. 2015. Long short-term memory over recursive structures. In *International Conference on Machine Learning*. 1604–1612.