Deep Learning Similarities from Different Representations of Source Code

Michele Tufano
College of William and Mary
Williamsburg, Virginia, USA
mtufano@cs.wm.edu

Cody Watson
College of William and Mary
Williamsburg, Virginia, USA
cawatson@cs.wm.edu

Gabriele Bavota
Università della Svizzera italiana (USI)
Lugano, Switzerland
gabriele.bavota@usi.ch

Massimiliano Di Penta
University of Sannio
Benevento, Italy
dipenta@unisannio.it

Martin White
College of William and Mary
Williamsburg, Virginia, USA
mgwhite@cs.wm.edu

Denys Poshyvanyk
College of William and Mary
Williamsburg, Virginia, USA
denys@cs.wm.edu

ABSTRACT

Assessing the similarity between code components plays a pivotal role in a number of Software Engineering (SE) tasks, such as clone detection, impact analysis, refactoring, *etc.* Code similarity is generally measured by relying on manually defined or hand-crafted features, *e.g.,* by analyzing the overlap among identifiers or comparing the Abstract Syntax Trees of two code components. These features represent a *best guess* at what SE researchers can utilize to exploit and reliably assess code similarity for a given task. Recent work has shown, when using a stream of identifiers to represent the code, that Deep Learning (DL) can effectively replace manual feature engineering for the task of clone detection. However, source code can be represented at different levels of abstraction: identifiers, Abstract Syntax Trees, Control Flow Graphs, and Bytecode. We conjecture that each code representation can provide a different, yet orthogonal view of the same code fragment, thus, enabling a more reliable detection of similarities in code. In this paper, we demonstrate how SE tasks can benefit from a DL-based approach, which can automatically learn code similarities from different representations.

CCS CONCEPTS

• Computing methodologies → Neural networks; **• Software and its engineering → Reusability**;

KEYWORDS

deep learning, code similarities, neural networks

ACM Reference Format:
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 *MSR '18: MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3196398.3196431

1 INTRODUCTION

Source code analysis is a rock-solid foundation of many Software Engineering (SE) tasks. Source code analysis methods depend on a number of different code representations (or models), which include, source code identifiers and comments, Abstract Syntax Trees (ASTs), Control-Flow Graphs (CFGs), data flow, bytecode, *etc.* These distinct representations of code provide different levels of abstraction, which create explicit and implicit relationships among source code elements.

The importance of a specific code representation is dependent upon the SE task. For example, identifiers and comments encode domain semantics and developers' design rationale, making them useful in the context of feature location [19, 20, 22, 58] and software (re)modularization [11–13]. Additionally, programs that have a well-defined syntax can be represented by ASTs, which, in turn, can be successfully used to capture programming patterns and similarities [6, 52, 54]. Since there are numerous SE tasks, such as static and dynamic program analysis, change history analysis, automated testing, verification, program transformation, clone detection *etc.*, it is important to rely on different available code representations so that different source code relationships can be efficiently identified. Yet, many existing solutions to these SE tasks are based on "hardcoded" algorithms rooted in the underlying properties of the specific code representation they use, which in order to work properly, also need to be adequately configured [56, 57, 71].

While SE researchers regularly use machine learning (ML) or Information Retrieval (IR) techniques to solve important SE tasks, many of these techniques rely on manually defined or hand-crafted features. These features then allow for ML-based SE applications. For example, distinct identifiers are typically used as features in concept location approaches [21], APIs (a reserved subset of identifiers) are used as features in approaches for identifying similar applications [47, 69], and AST pattern similarities are used to enable clone detection and refactoring [26, 33, 50]. However, it should be noted that all these features are selected via "*an art of intuitive feature engineering*" by SE researchers or domain (task) experts. While there are many successful and widely adopted ML-based approaches that use different code representations to support SE tasks, their performance varies depending on the underlying datasets [56, 57].

Improvements in both computational power and the amount of memory in modern computer architectures have enabled the development of new approaches to canonical ML-based tasks. The rise of deep learning (DL) has ushered in tremendous advances in different fields [43, 48]. These new approaches use representation learning [16] — a significant departure from traditional approaches — to automatically extract useful information from the disproportionate amount of unlabeled software data. The value in circumventing conventional feature engineering when designing ML-based approaches to SE tasks is two-fold. Firstly, learning transformations of the data drastically reduces the cost of modeling code, since software (and various code representations) store a lot of data to improve the effectiveness of learning. Secondly, performance is increased because generally, learning algorithms are more efficient than humans at discovering correlations in high dimensional spaces.

While the number of applications of DL to SE tasks is growing [4, 5, 15, 25, 29–32, 41, 42, 42, 51, 53, 70, 73], one recent example by White *et al.* [74] shows that DL can effectively replace manual feature engineering for the task of clone detection. Existing clone detection approaches leverage algorithms working on manually specified sets of features, and include, for example, text-based [23, 34–36, 59] string-based [7–9], token-based [37, 44, 60, 61], AST-based [14, 33, 39, 75], graph-based [18, 26, 38, 40, 45], or lexicon-based [46] approaches.

In this paper, we posit a fundamental question of whether an underlying code representation can be successfully used to automatically learn code similarities. In our approach, we employ the representations of identifiers, ASTs, CFGs, and Bytecode and use DL algorithms to automatically learn necessary features, which in turn, can be used to support SE tasks. Moreover, we also study whether combinations of the models trained on diverse code representations can yield more accurate and robust support for a SE task at hand. Our conjecture is that each code representation can provide an orthogonal view of the same code fragment, thus allowing a more reliable detection of similar fragments. Being able to learn similarities from diverse code representations can also be helpful in many practical settings, where some representations are simply not available or when some of the representations are compromised (*e.g.,* code obfuscation would prevent using identifier-based approaches).

While we conduct our experiments on a specific SE task — clone detection — our goal is not to develop an ultimate clone-detection approach, but rather show that effective DL-based solutions can be assembled from diverse representations of source code. Specifically, the noteworthy contributions of this work are as follows:

(1) *Deep Learning from different code representations.* We demonstrate that it is indeed possible to automatically learn code similarities from different representations, such as streams of identifiers, AST nodes, bytecode mnemonic opcodes, and CFGs. We evaluate how these representations can be used in a DL-based approach for clone detection. We argue that our results should be useful for any SE task that relies on analyzing code similarities.
(2) *Assembling a combined model.* We demonstrate that combined models can be automatically assembled to consider multiple representations for SE tasks (in our example, code clone detection). In fact, we show that the combined model achieves overall better results as compared to stand-alone code representations.
(3) *Inter-project similarities.* We also demonstrate that the proposed models can be effectively used to compute similarities, not only in the context of a single project, but also to analyze code similarities among different (diverse) software projects (*e.g.,* detecting clones or libraries across multiple projects).
(4) *Model reusability and transfer learning.* We demonstrate that we can learn multi-representation models on available software systems and then effectively apply these models for detecting code similarities on previously unseen systems.
(5) *Open science.* We release all the data used in our studies [2]. In particular, we include all the source code, datasets, inputs/outputs, logs, settings, analysis results, and manually validated data.

2 DEEP LEARNING IN SE

To the best of our knowledge, our work is the very first attempt to show that it is possible to automatically learn (via DL) simultaneously from different source code representations. There has been some preliminary work on using DL to replace manual feature engineering for the task of clone detection [74]. However, we are not aware of any other learning-based approach that operates on code representations other than identifiers and comments. In this section, we review some of the recent and representative papers that rely on DL in the context of SE tasks.

White *et al.* [74] used DL and, in particular, *representation learning* via recursive autoencoder, for the purpose of code clone detection. They represent each source code fragment as a stream of identifiers and literal types (from the AST leaf nodes).

Lam *et al.* [41] focuses on bug localization using both DL and IR techniques. Rather than manually defining features, the IR technique revised Vector Space Models (rVSM) collects these features which capture textual similarity. After that, a deep neural network learns how to relate terms in bug reports to tokens within source code.

Allamanis *et al.* [4] proposed an approach for suggesting meaningful class and method names. To accomplish this, identifiers in code are assigned a continuous vector, which considers the local context as well as long range dependencies by a neural log-bilinear context model. Then, identifiers which have similar vectors or embeddings will also appear to have similar contexts. However, in order to capture the global context of tokens, features are engineered, which requires configuration and manipulation when integrated with the model. To build upon their previous work, Allamanis *et al.* [5] used an Attentional Neural Network (ANN) combined with convolution on the input tokens for assigning descriptive method names. Their approach allows for automatic learning of translation-invariant features.

Gu *et al.* [29] used DL, to avoid feature engineering, in order to learn API usage scenarios given a natural language query. The approach encodes a query or annotation of the code into a fixed-length context vector. This vector helps to decode an API sequence which should correlate with the query. Therefore, once the model is trained, a natural language query will result in the correct API usage scenario for the context given in the query.

Wang *et al.* [70] used a Deep Belief Network (DBN) to learn semantic features from token vectors extracted from ASTs to perform defect prediction. Similar to our work, they learn a code representation and apply it to a SE task.

As in our case, features are learned automatically from the DL approach. The input to the DBN begins with source code, which is parsed, and a token vector is created. Then, the DBN produces a feature vector which can be analyzed for fault prediction.

All the aforementioned works are instances of DL approaches on source code applied to different SE tasks. Although our work is similar in that we use DL, our goal is not to develop an approach for a specific SE task. Rather, our intent is to empirically demonstrate that DL applied to SE can benefit from considering multiple representations of code. As the previous work has shown, DL can be applied to different representations and yield meaningful results to many SE tasks. However, we show that there is value in every code representation, and therefore all representations should be considered in order to identify all possible features from the data.

3 PROPOSED APPROACH

Our approach can be summarized as follows. First, code fragments are selected at the different granularities we wish to analyze (*e.g.,* classes, methods). Next, for each selected fragment, we extract its four different representations (*i.e.,* identifiers, AST, bytecode, & CFG). Code fragments are embedded as continuous valued vectors, which are learned for each representation. In order to detect similar code fragments, distances are computed among the embeddings.

3.1 Code Fragments Selection

Given a compiled code base, formed by source code files and compiled classes, code fragments are selected at the desired level of granularity: classes or methods. We start by listing all the `.java` files in the code base. For each Java file, we build the AST rooted at the `CompilationUnit` of the file. To do this, we rely on the Eclipse JDT APIs. We use the Visitor design pattern to traverse the AST and select all the classes and methods corresponding to `TypeDeclaration` and `MethodDeclaration` nodes. We discard interfaces and abstract classes since their methods do not have a bytecode and CFG representation. While it is possible to extract the other two representations (identifiers and ASTs), we chose to discard them so that we only learned similarities from code which exhibits all four representations. In addition, we filter out fragments (*i.e.,* classes or methods) smaller than 10 statements. Similar thresholds have been used in previous work for minimal clone size [72]. Furthermore, smaller repetitive snippets are defined as micro-clones. Syntactic repetitiveness below this threshold has simply been considered uninteresting because it lacks sufficient content [10]. For each code fragment $c_i \in \{Classes \cup Methods\}$ we extract its AST node n_i and its fully qualified name (signature) s_i. Identifier and AST representations are extracted from an AST node n_i, while the bytecode and CFG representations are queried using the fully qualified name s_i.

3.2 Code Representation Extraction

We use the following representations of the code: (i) identifiers; (ii) ASTs; (iii) bytecode; and (iv) CFGs. Extraction and normalization are two prepossessing steps we perform for each representation. In this section we describe these two steps for each selected representation.

3.2.1 *Identifiers.*

In this representation a code fragment is expressed as a stream (sentence) of identifiers and constants from the code. A similar representation is used by White *et al.* [74].
Extraction. Given the code fragment c_i and its corresponding AST node n_i, we consider the sub-tree rooted at the node n_i. To extract the representation, we select the terminal nodes (leaves) of the sub-tree and print their values. The leaf nodes mostly correspond to the identifiers and constants used in the code.
Normalization. Given the stream of printed leaf nodes, we normalize the representation by replacing the constant values with their type (< `int` >, < `float` >, < `char` >, < `string` >).

3.2.2 *AST.*

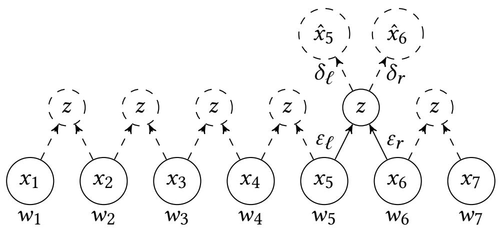
In this representation, a code fragment is expressed as a stream (sentence) of the node types that compose its AST.
Extraction. Similarly to what was described above, the sub-tree rooted at the node n_i is selected for a given code fragment c_i. Next, we perform a pre-order visit of the sub-tree printing, for each node encountered, its node type.
Normalization. We remove two AST node types: `SimpleName` and `QualifiedName`. These nodes refer to identifiers in the code, and were removed because: (i) they represent low-level nodes, which are less informative than high-level program construct nodes in the AST (*e.g.,* `VariableDeclarationFragment`, `MethodInvocation`); (ii) they account for ~46% of the AST nodes leading to a very large yet repetitive corpus; (iii) we target an AST representation able to capture orthogonal information as compared to the identifiers representation. The latter is formed for ~77% of terms belonging to `SimpleName`/`QualifiedName` nodes.

3.2.3 *CFG.*

In this representation a code is expressed as its CFG.
Extraction. To extract the CFG representation, we rely on Soot [3], a popular framework used by researchers and practitioners for Java code analysis. First, we extract the fully qualified name of the class from the signature s_i of the code fragment c_i. We use it to load the compiled class in Soot. For each method in the class, the CFG $G = (V, E)$ is extracted, where V is the set of vertices (*i.e.,* statements in the method) and E the set of directed edges (*i.e.,* the control flow between statements). In particular, the node represents the numerical ID of the statement as it appears in the method. Since the CFG is an intra-procedural representation, the method-level representation is a graph, while the class-level representation is a forest of graphs (CFGs of its methods).
Normalization. The CFG represents code fragments at a high-level of abstraction, therefore, no normalization is performed.

3.2.4 *Bytecode.*

In this representation, a code fragment is expressed as a stream (sentence) of bytecode mnemonic opcodes (*e.g.,* `iload`, `invokevirtual`) forming the compiled code.
Extraction. Let c_i be the code fragment and s_i its signature. If c_i is a method, we extract the fully qualified name of its class from s_i, otherwise s_i already represents the name of the class. Then, the bytecode representation is extracted using the command `javap – c – private < classname >` passing the fully qualified name of the compiled class. The output is parsed, allowing for the extraction of the class- or method-level representation.
Normalization. In the normalization step we remove the references to constants, keeping only their opcodes. For example, the



$\hat{x}_5$
$\hat{x}_6$
δ_ℓ
δ_r
z
ε_ℓ
ε_r
x_1 x_2 x_3 x_4 x_5 x_6 x_7
w_1 w_2 w_3 w_4 w_5 w_6 w_7

Figure 1: First iteration to encode a stream of terms

instruction `putfield#2` is normalized as `putfield`. We also separate the opcodes stream of each method with the special tag < `M` >.

3.3 Embedding Learning

For each code fragment $c \in \{Classes \cup Methods\}$, we extract its representations: r_{ident}, r_{AST}, r_{byte}, and r_{CFG}. We learn a single embedding (*i.e.,* vector) for each representation, obtaining: e_{ident}, e_{AST}, e_{byte}, and e_{CFG}. An embedding represents a code fragment in a multidimensional space where similarities among code fragments can be computed as distances.

We use two strategies to learn embeddings for the aforementioned representations. For identifier, AST and bytecode representations, we use a DL-based approach that relies on recursive autoencoders [63, 64]. For the CFG representation we use a Graph Embedding technique [55].

3.3.1 *DL Strategy.*

Let C be the set of all code fragments and R the corpus comprising the representations of the code fragments in a representation (r_{ident}, r_{AST}, r_{byte}). For each code fragment $c \in C$, its representation $r \in R$ is a sentence of the corpus R. The sentence r is a stream of words $r = w_1, w_2, \ldots, w_j$, where w_i is a term of the particular representation (*i.e.,* an identifier, AST node type or bytecode opcode). The corpus is associated with a vocabulary V containing the unique terms in the corpus.

To learn an embedding for a sentence r, we perform two steps. In the first stage, we learn an embedding for each term w_i (*i.e.,* word embeddings), which comprises the sentence. In the second stage, we recursively combine the word embeddings to learn an encoding for the entire sentence r. We now describe these two stages in detail.

In the first stage we train a Recurrent Neural Network (RtNN) on the corpus R, where the size of the hidden units is set to n, which corresponds to the embedding size [49]. The model, trained on the corpus R, generates a continuous valued vector, called an embedding, for each word $w_i \in V$.

The second stage involves training a Recursive Autoencoder [63, 64] to encode arbitrarily long streams of embeddings. Fig. 1 shows the recursive learning procedure. Consider the sentence $r \in R$ formed by seven terms $\{w_1, \ldots, w_7\}$. The first step maps the stream of terms to a stream of n–dimensional embeddings $\{x_1, \ldots, x_7\}$. In the example in Fig. 1, there are six pairs of adjacent terms (*i.e.,* $[x_i; x_{i+1}]$). Each pair of adjacent terms $[x_\ell; x_r]$ is εncoded by performing the following steps: (i) the two n-dimensional embeddings, corresponding to the two terms, are concatenated in a single $2n$-dimensional vector $x = [x_\ell; x_r] \in \mathbb{R}^{2n}$; (ii) x is multiplied by a matrix $\varepsilon = [\varepsilon_\ell, \varepsilon_r] \in \mathbb{R}^{n \times 2n}$; (iii) a βias vector $\beta_z \in \mathbb{R}^n$ is added to the result of the multiplication; (iv) the result is passed to a nonlinear vector $f$unction f: $z = f(\varepsilon x + \beta_z)$.

The result z is an n-dimensional embedding that represents an encoding for the stream of two terms, corresponding to x. In the example in Fig. 1, x_ℓ and x_r correspond to the embeddings x_5 and x_6 respectively, which in turn, correspond to the terms w_5 and w_6. In this step the autoencoder performs dimensionality reduction. In order to assess how good z encodes the pair $[x_\ell; x_r]$, the autoencoder tries to reconstruct the original input x from z in the δecoding phase. z is δecoded by multiplying it by a matrix $\delta = [\delta_\ell; \delta_r] \in \mathbb{R}^{2n \times n}$ and adding a βias vector $\beta_y \in \mathbb{R}^{2n}$: $y = \delta z + \beta_y$.

The output $y = [\hat{x}_\ell; \hat{x}_r] \in \mathbb{R}^{2n}$ is referred to as the model's *reconstruction* of the input. This model $\theta = \{\varepsilon, \delta, \beta_z, \beta_y\}$ is called an *autoencoder*, and training the model involves measuring the Error between the original input vector x and the reconstruction y:

$$E(x; \theta) = ||x_\ell - \hat{x}_\ell||_2^2 + ||x_r - \hat{x}_r||_2^2 \tag{1}$$

The model is trained by minimizing Eq. (1). Training the model to encode streams with more than two terms requires recursively applying the autoencoder. The recursion can be performed following predefined recursion trees or by using optimization techniques, such as the greedy procedure defined by Socher *et al.* [62]. The procedure works as follows: in the first iteration, each pair of adjacent terms are encoded (Fig. 1). The pair whose encoding yields the lowest reconstruction error (Eq. (1)) is the pair selected for encoding at the current iteration. For example, in Fig. 1, each pair of adjacent terms are encoded (*e.g.,* dashed lines) and the pair of terms w_5 and w_6 is selected to be encoded first. As a result, in the next iteration, x_5 and x_6 are replaced by z and the procedure repeats. Upon deriving an encoding for the entire stream, the backpropagation through structure algorithm [27] computes partial derivatives of the (global) error function w.r.t. θ. Then, the error signal is optimized using standard methods.

3.3.2 *Graph Embedding Strategy.*

To generate the embeddings for CFG representations we employ the Graph Embedding Technique HOPE [55] (High-Order Proximity preserved Embedding). We rely on this technique for two main reasons: (i) it has been shown to achieve good results in graph reconstruction [28]; (ii) differently from other techniques (*e.g.,* SDNE), HOPE embeds directed graphs (as needed in the case of Control Flow Graphs). Given a graph $G = (V, E)$, HOPE generates an embedding for each node in the graph. Next, a single embedding is generated for the whole graph performing mean pooling on the node's embeddings. HOPE works by observing a critical property of directed graphs known as asymmetric transitivity. This property helps to preserve the structure of directed graphs by identifying correlations of directed edges. For example, assume three distinct, directed paths from v_1 to v_5 . Each of these paths increases the correlation probability that there exists a directed edge from v_1 to v_5 , however, the lack of directed paths from v_5 to v_1 decreases the probability of there being a direct edge from v_5 to v_1.

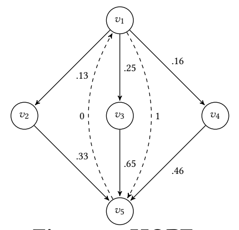


Figure 2: HOPE

HOPE preserves asymmetric transitivity by implementing the highly correlated metric known as the Katz proximity. This metric gives weights to the asymmetric transitivity pathways, such that they can be captured through the embedding vectors. HOPE learns two embedding vectors; the source vector and the target vector for each vertex. These vectors are then assigned values based upon the weights of the edges and their nature (source or target). For example, consider the graph in Fig. 2. Each of the solid lined edges are directed edges and the dotted edges capture the asymmetric transitivity of the graph. The vector created for v_1 as the target vector will be 0 since no paths lead to v_1 as their target. However, the value assigned to v_5 target vector will be much higher since many paths end with v_5 as their target. HOPE creates and learns the embeddings of the graph via Singular Value Decomposition for each vertex. Then, through mean pooling on the nodes embeddings, a single embedding for the entire graph can be generated.

3.4 Detecting Similarities

Let E be the set of embeddings learned for all code fragments by a particular representation (*i.e.,* E_{ident}, E_{AST}, E_{byte}, E_{CFG}). We compute pairwise Euclidean distances between each and every pair of embeddings $e_i, e_j \in E$. The set of distances D are normalized between $[0, 1]$ and a threshold t is applied to the distances in order to detect similar code fragments.

3.5 Combined Models

Each of the four models we built is trained on a single representation and identifies a specific set of similar code fragments. Such models can be combined using *Ensemble Learning*, an ML paradigm where multiple learners are trained to solve the same problem. In contrast to ordinary machine learning approaches, which try to learn *one* hypothesis from training data, ensemble methods try to construct a *set* of hypotheses and combine them [76].

A simple class of Ensemble Learning techniques are the algebraic combiners, where distances computed by several, single-representation models are combined through an algebraic expression, such as minimum, maximum, sum, mean, product, median, *etc.* For example, a weighted average sum of the distances computed from each representation can be computed. Formally, given two code fragments a and b, and their n representations, one can compute a dissimilarity score ds as follows:

$$ds(a,b) = \frac{1}{n}\sum_{i=1}^{n} w_i d_i(a,b)$$

Where $d_i(a,b)$ is the distance between a and b computed using the i-th representation, and w_i is the weight assigned to the i-th representation. Weights can be set based on the importance of each representation and the types of similarities we wish to detect.

Single-representation models can also be treated as experts and combined using voting-based methods. Each single-representation model expresses its own vote about the similarity of two code fragments (*i.e.,* are similar *vs* are not similar) and these decisions are combined in a single label. Different strategies can be used depending on the goal: (i) (weighted) majority of voting; (ii) at least one vote (max recall); (iii) all votes needed (max precision).

The aforementioned strategies are non-trainable combiners. However, given a set of instances for which the oracle is available, ensemble learning techniques can be employed to perform training. Random Forest is one of the most effective ensemble learning method for classification, which relies on decision tree learning and bagging [17]. Bagging (*i.e.,* bootstrap aggregating), is a technique that generates bootstrapped replicas of the training data. In particular, different training data subsets are randomly drawn, with replacement from the entire training dataset. Random Forest, in addition to classic bagging on the training instances, also selects a random subset of the features (feature bagging). Each training data subset is used to train a different deep decision tree. For a new instance, Random Forest averages the predictions by each decision tree, effectively reducing the overfitting on the training sets.

In the following, we show the effectiveness of combining similarities learned from different representations training two models in the context of clone detection (*i.e.,* a model classifying clone candidates as *true* or *false* positives) and classification (*i.e.,* a model that other than discerning clone candidates in *true* or *false* positives, also classifies the true positives in their own clone type).

Similarities from different code representations can be useful not only to detect clones, but also to classify them into types. For example, clone pairs with very high AST similarity, but low identifiers similarity are likely to be Type-II clones (possibly parameterized clones, where all the identifiers have been systematically renamed).

Table 1: Project Statistics

Project	#Classes	#Methods	Total LOC
ant-1.8.2	1,608	12,641	127,507
antlr-3.4	381	3,863	47,443
argouml-0.34	1,408	8,465	105,806
hadoop-1.1.2	3,968	21,527	319,868
hibernate-4.2.0	7,119	46,054	431,693
jhotdraw-7.5.1	765	6,564	79,672
lucene-4.2.0	4,629	23,769	412,996
maven-3.0.5	837	5,765	65,685
pmd-4.2.5	872	5,490	60,739
tomcat-7.0.2	1,875	15,275	181,023

4 EXPERIMENTAL DESIGN

The *goal* of this study is to investigate whether source code similarity can be learned from different kinds of program representations, with the *purpose* of understanding whether one can combine different representations to obtain better results, or use alternative representations when some are not available. More specifically, the study aims at answering the following research questions (RQ):

- **RQ$_1$** How effective are different representations in detecting similar code fragments?
- **RQ$_2$** What is the complementarity of different representations?
- **RQ$_3$** How effective are combined multi-representation models?
- **RQ$_4$** Are DL-based models applicable for detecting clones among different projects?
- **RQ$_5$** Can trained DL-based models be reused on different, previously unseen projects?

4.1 Datasets Selection

We make use of two datasets: (i) *Projects*: a dataset of 10 Java projects; (ii) *Libraries*: a dataset comprising 46 Java libraries.

4.1.1 Projects. This dataset comprising 10 compiled Java projects extracted from the *Qualitas.class Corpus* [68]. We rely on this dataset because it is publicly available and the projects have already been compiled. This (i) avoids any potential problem/inconsistency in compiling projects, and (ii) ensures reproducibility. The selection of 10 projects aimed at obtaining a diverse dataset, in terms of application domain and code size. Table 1 reports statistics of the dataset that we use in all our research questions but RQ$_4$.

4.1.2 Libraries. This dataset comprising 46 different Apache commons libraries [1]. We selected all the Apache commons libraries, for which we were able to identify both binaries and source code of the latest available release. We downloaded the compressed files for binaries and source code. Within the binaries, we located the `jar` file, which represents the library, and extracted the `.class` files. The compressed source code files were simply decompressed. The list of considered libraries is available in our replication package. We use this dataset in RQ$_4$ for inter-project clone detection.

4.2 Experimental Procedure and Data Analysis

We discuss the experimental procedure and data analysis we followed to answer each research question.

4.2.1 RQ$_1$: How effective are different representations in detecting similar code fragments? Given the projects dataset $P = \{P_1, \ldots, P_{10}\}$ and the four code representations $R = \{R_1, R_2, R_3, R_4\}$, we extract for each code artifact $c \in \{Classes \cup Methods\}$ its four representations r_1, r_2, r_3, r_4. Then, for each project P_i we train the representation-specific model on the code representations at class-level. With the trained models, we subsequently generate the embeddings for both classes and methods in the project. Therefore, the code artifact represented as r_1, r_2, r_3, r_4 will be embedded as e_1, e_2, e_3, e_4, where e_i is the embedding of the i-th representation. We set the embedding size of Identifiers, AST and Bytecode to 300, and the CFG embedding size to 4. The latter is significantly smaller than the former because (i) CFGs are abstract representations that do not require large embeddings; (ii) in order to converge towards an embedding representation, HOPE (and internally SVD) needs the embedding size to be smaller than the minimum number of nodes or edges in the graph. Pairwise Euclidean distances are computed for each pair of classes and methods of each system P_i. The smaller the distance, the more similar the two code artifacts.

In the next step, for each code representation R_i, we query the clone candidates from the dataset P at class and method level. To query the clone candidates, we apply a threshold on the distances to discern what qualifies as clones. We use the same two thresholds at class- and method-level ($T_{class} = 1.00e{-}08$ and $T_{methods} = 1.00e{-}16$, similar to [74]) for each representation. While, ideally, these thresholds should be tuned for each project and representation, we chose the same thresholds to facilitate the comparison among the four representations. Once we obtain the two sets of candidates $CandClasses_i$ and $CandMethods_i$ for each representation R_i, we perform the union of the candidate sets of the same granularity among all the representations: $CandClasses = \bigcup_{i=1}^{4} CandClasses_i$ (the same applies for $CandMethods$).

For each candidate $c \in CandClasses$ (or $CandMethods$) we generate a tuple $t_c = \{b_1, b_2, b_3, b_4\}$ where $b_i = True$ iff $c \in CandClass_i$ (*i.e.,* if c is identified as a clone by R_i) and $b_i = False$ otherwise. t_c can assume $2^4 = 16$ possible values (*i.e.,* $t_c = \{FFFF, FFFT, \ldots, TTTT\}$). However, the combination $FFFF$ does not appear in our dataset since $CandClasses$ and $CandMethods$ are sets containing the union of the clones identified by all representations, thus ensuring the presence of at least one True value. Therefore, there are 15 unique classes of values for t_c. We use these sets to partition the candidates in $CandClasses$ and $CandMethods$. Next, from each candidate clones partition, we randomly draw a statistically significant sample with 95% confidence level and ±15% confidence interval. The rationale is that we want to evaluate candidates belonging to different levels of agreement among the representations. Three authors independently evaluated the clone candidate samples. The evaluators decided whether the candidates were true or false positives and, in the former case, the clone type. To support consistent evaluations, we adapted the taxonomy of editing scenarios designed by Svajlenko *et al.* [67] to model clone creation and to be general enough to apply to any granularity level. Given the manually labeled dataset from each of the three evaluators, we computed the two-judges agreement to obtain the final dataset (*e.g.,* a candidate clone was marked as a true positive if at least two of the three evaluators classified it as such). In order to statistically assess the evaluators' agreement, we compute the Fleiss' kappa [24]. In particular, we compute the agreement for True and False positives as well as the agreement in terms of Clone Types. On the final dataset, precision and recall were computed for each candidate clones partition (*e.g., TFFF, FTFF, etc.*) and, overall, for each representation in isolation. We quantitatively and qualitatively discuss TP/FP pairs for each representation, as well as the distribution of clone types.

4.2.2 RQ$_2$: What is the complementarity of different representations? To answer RQ$_2$ we further analyze the data obtained in RQ$_1$ to investigate the complementarity of the four representations. First, we compute the intersection and the difference of the true positive sets identified with each representation. Precisely, the intersection and difference of two representations R_i and R_j are defined as follows:

$$R_i \cap R_j = \frac{|TP_{R_i} \cap TP_{R_j}|}{|TP_{R_i} \cup TP_{R_j}|}\% \quad \text{and} \quad R_i \setminus R_j = \frac{|TP_{R_i} \setminus TP_{R_j}|}{|TP_{R_i} \cup TP_{R_j}|}\%$$

where TP_{R_i} represents true positive candidates identified by R$_i$. We compute the percentage of candidates exclusively identified by a single representation and missed by all the others:

$$EXC(R_i) = \frac{|TP_{R_i} \setminus \bigcup_{j \neq i} TP_{R_j}|}{|\bigcup_j TP_{R_j}|}\%$$

Then, to understand whether these code representations are orthogonal to each other, we collect all the distance values of each representation computed for each pair of code fragments (classes or methods) in the dataset. Given these distance values, we compute the Spearman Rank Correlation [65] between each pair of representations, to investigate the extent to which distances computed from different representations on the same pairs of artifacts correlate.

4.2.3 RQ$_3$: How effective are combined multi-representation models? To answer RQ$_3$ we evaluate the effectiveness of two combined models: *CloneDetector*, which classifies candidate clones as true/-false positives, and *CloneClassifier*; which provides information about the type of detected clone (Type-1, 2, 3 or 4). While we are aware that once a clone pair has been identified it is possible to determine its clone type by comparing the sequence of tokens, the purpose of this classification is to show the potential of the proposed approach to provide complete information about a clone pair. The two models are trained using the manually labeled dataset obtained in RQ_1, with the distances computed by each representation used as features and the manual label used as target for the prediction. To train the two models we rely on Random Forest employing the commonly used 10-fold cross-validation technique to partition training and test sets. We evaluate the effectiveness of the two models computing Precision, Recall and F-Measure for each class to predict (*e.g.,* clone *vs* not-clone for the *CloneDetector*).

4.2.4 RQ_4: Are DL-based models applicable for detecting clones among different projects? The goal of RQ_4 is two-fold. On the one hand, we want to instantiate our approach in a realistic usage scenario in which only one code representation is available. On the other hand, we also want to show that the DL based model can be used to identify inter-project clones. Indeed, the latter is one of the major limitations of the work by White *et al.* [74], where given the potentially large vocabulary of identified-based corpora, the approach was evaluated only to detect intra-project clones.

We instantiate two usage scenarios both relying on the same *Library* dataset, and on training performed on the binaries (bytecode from `.class` files). In the first scenario, a software maintainer has to analyze the amount of duplicated code across projects belonging to their organization. We use the entire *Library* dataset and filter for inter-project clone candidates only (*i.e.,* candidates belonging to different projects). Clone candidates are evaluated by inspecting the corresponding Java files from the downloaded code.

In the second scenario, a software developer is using a *jar* file (*i.e.,* compiled library) in their project. The developer has no information about other libraries that could be redistributed with the *jar* file, since only compiled code is present. For provenance and/or licensing issues, the developer needs to address whether the *jar* file j imports/shadows any of the libraries in a given dataset L.

To perform this study we select `weaver-1.3` as the *jar* j and the remaining 45 libraries in the *Library* dataset as L. We identify similar classes between j and L. Then, to assess whether the identified classes have actually been imported in j from library $x \in L$, we downloaded the j's source code and analyzed the building files (`weaver-1.3` relies on Maven, thus we investigated the `.pom` files) to check whether the library x is imported as a dependency in j.

4.2.5 RQ_5: Can trained DL-based models be reused on different, previously unseen projects? One of the drawbacks of DL-based models is their long training time. This training time could be amortized if these models could be reused across different projects belonging to different domains. The major factor that hinders the reusability of such models is the possible variability in the vocabulary for new, unseen projects. For example, a language model and a recursive autoencoder trained on a given vocabulary would not be able to provide adequate results on a vocabulary containing terms not previously seen during the training phase. Such a vocabulary should be cleaned, either by stripping off the unknown terms, replacing them with special words (*e.g.,* < `unkw` >) or using smoothing techniques. These solutions negatively impact the performance of the models.

Table 2: Performances for different clone partitions

ID	Iden	AST	CFG	Byte	Precision % Methods	Precision % Classes
1	F	F	F	**T**	5	49
2	F	F	**T**	F	9	58
3	F	F	**T**	**T**	88	73
4	F	**T**	F	F	79	63
5	F	**T**	F	**T**	95	93
6	F	**T**	**T**	F	100	100
7	F	**T**	**T**	**T**	100	100
8	**T**	F	F	F	95	100
9	**T**	F	F	**T**	100	100
10	**T**	F	**T**	F	100	-
11	**T**	F	**T**	**T**	100	100
12	**T**	**T**	F	F	100	100
13	**T**	**T**	F	**T**	100	100
14	**T**	**T**	**T**	F	100	100
15	**T**	**T**	**T**	**T**	100	100

While in principle, with enough training data and available time, any of the aforementioned representation-specific models could be reused on a different dataset (*i.e.,* unseen project), in practice some representation-specific models are more suitable than others to be reused. In particular, models trained on representations with a limited or fixed vocabulary are easier to be reused on different projects. In our study, AST and Bytecode representations both have a fixed vocabulary, limited respectively by the number of different AST node types and bytecode opcodes.

To answer RQ_5 we perform a study aimed at evaluating the effectiveness of reusing AST models. We evaluate the effectiveness, showing that a reused model identifies a similar list of clone candidates as compared to the original model trained on the set projects. We select the AST representation which was trained on one of the largest projects in the dataset, *i.e.,* `lucene`. We use this AST model to generate the embeddings for the remaining nine projects in the dataset. Using the generated embeddings we compute the distances and query the clone candidates using the same class- and method-level thresholds used in RQ_1. Then, let L_R and L_O be the lists of candidates returned from the reused model and from the original model, we compute the percentage of candidates in L_R that are in L_O and *vice versa.*

We also show that the combined model *CloneDetector*, trained on clone candidates belonging to a single project (`hibernate`), can be effectively used to detect clones in the remaining nine systems.

5 RESULTS

RQ_1: How effective are different representations in detecting similar code fragments? Table 2 shows precision results for different candidate clone partitions, where each clone partition includes clones detected only by a given combination of representations. For example, the first partition $FFFT$ stands for the clone candidates identified by the Bytecode representation, but not by the other three representations. Note that for the partition with $ID = 10$ ($TFTF$) no class-level clones have been identified.

Table 2 shows that most of the partitions exhibit a good precision, with peaks of 100%. The notable exceptions are the partitions with ID 1 and 2, referring to clone candidates detected only by the Bytecode and by the CFG representations, respectively. We investigated such false positives and qualitatively discuss them later.

Table 3: Performances for different representations

Methods								
Representation	FP	TP	Type I	Type II	Type III	Type IV	Precision	Recall
Iden	1	201	151	15	35	0	100%	52%
AST	11	292	138	132	19	3	96%	75%
CFG	43	178	69	81	19	9	81%	46%
Byte	46	222	89	77	49	7	83%	57%
Classes								
Representation	FP	TP	Type I	Type II	Type III	Type IV	Precision	Recall
Iden	0	120	23	51	46	0	100%	40%
AST	18	188	18	121	44	5	91%	63%
CFG	24	120	7	65	41	7	83%	40%
Byte	34	217	23	115	77	2	86%	73%

Table 4: Complementarity Metrics

Methods											
Intersection %					Difference %					Exclusive %	
$R_1 \cap R_2$	Iden	AST	CFG	Byte	$R_1 \setminus R_2$	Iden	AST	CFG	Byte	R_i	$EXC(R_i)$
Iden		40	21	36	Iden		17	43	29	Iden	5% (21)
AST			42	44	AST	43		46	38	AST	9% (33)
CFG				36	CFG	36	12		24	CFG	1% (4)
Byte					Byte	35	18	39		Byte	1% (2)
Classes											
Intersection %					Difference %					Exclusive %	
$R_1 \cap R_2$	Iden	AST	CFG	Byte	$R_1 \setminus R_2$	Iden	AST	CFG	Byte	R_i	$EXC(R_i)$
Iden		33	14	42	Iden		19	43	8	Iden	3% (8)
AST			31	51	AST	48		49	19	AST	9% (26)
CFG				34	CFG	43	20		14	CFG	7% (21)
Byte					Byte	49	30	52		Byte	7% (21)

Table 3 shows the overall results for method- and class-level aggregated by representation. The table contains the raw count of True Positives (TP), False Positives (FP) and clone types identified by each representation. Estimated precision and recall is also reported in the last two columns of the tables. Note that the results of the single representation R_i (*e.g.,* Identifier) reported in Table 3 refer to aggregated candidates of Table 2 where the representation R_i is the True (*e.g.,* clone partitions ID 8-15).

The Identifier-based models achieve the best precision, both when working at method- and class-level. AST-based models provide the best balance between precision and recall. In terms of types of clones, Identifier-based models detect the highest number of Type I, AST-based models identify the highest number of Type II, Bytecode models identify the highest number of Type III, and CFG models detect the highest number of Type IV. This suggests that the four representations are complementary, and will be addressed further in RQ$_2$.

The data presented in this section is based on the agreement of three evaluators. The Fleiss' kappa shows substantial agreement in terms of TP/FP (93% for methods and 65% for classes) and a moderate/substantial agreement in terms of clone types classification (75% for methods and 57% for classes). In the following, we discuss in detail results achieved by models using different representations.

Identifiers. The identifiers-based model achieves the best precision both in method- and class-level results. However its estimated recall is respectively 52% and 40%, meaning that the model fails to detect a significant percentage of clones. While the recall could be increased by appropriately raising the threshold (*i.e.,* by adopting a more permissive threshold) at the expense of precision, we found that this would still not be enough to detect most of the clones detected by other representations. Indeed, we manually investigated several TP clone candidates identified by other representations and missed by the Identifier-based model. We found that the distances computed by the Identifier-based model for such clones were several orders of magnitude higher than a reasonable threshold (*e.g.,* the ones we use or the ones used by White *et al.* [74]). Moreover, a close inspection at the vocabulary of the candidates showed that they share a small percentage of identifiers, which in turn, makes them hard to detect with such a representation. Clearly, clone candidates where pervasive renaming or obfuscation has been performed, would be very difficult to detect with this model.

Bytecode & CFG. From Table 2, we can notice that candidates detected only by Bytecode or only by CFG models (partitions with ID 1 and 2) tend to be false positives. The precision for such partitions is 5% and 9% at method-level and 49% and 58% at class-level. However, when both Bytecode and CFG models detect clones not detected by the AST and Identifier-based models (partition with ID 3), they achieve reasonable levels of precision (88% and 73%).

Bytecode and CFG representations have a very high degree of abstraction. Within the CFG, a statement is simply represented by a node. Similarly, the bytecode is formed by low level instructions, which do not retain the lexical information present in the code (*e.g.,* a method call is represented as a `invokevirtual` or `invokestatic` opcode). Such a level of abstraction appears to be imprecise for fine granularities such as methods, where the code fragments might have similar structure but, at the same time, perform very different tasks. Better results are achieved at class-level, where false positives are mainly due to Java Beans, thus classes having a very similar structure, *i.e.,* class attributes with getters and setters acting on them performing similar low-level operations (storing/loading a variable, creating an object, *etc.*). While these classes are false positives in terms of code clones, they still represent a successful classification in terms of learning structural and low-level similarities from the code. The general lesson learned is that the use bytecode and CFG in a combined model yield an acceptable precision. Indeed, if bytecode is available (compiled code) CFGs can be extracted as well.

Table 3 also shows that Bytecode and CFG are the representations that detect the most Type IV clones in our dataset. Again, this is likely due to their high level of abstraction in representing the code.

AST. AST-based models seem to have the best overall balance between precision and recall. These models can identify similar code fragments, even when their set of identifiers (*i.e.,* vocabulary) is significantly different and share almost no lexical tokens. This has been confirmed by our manual investigation of the candidates. We report several examples of candidates identified exclusively by the AST model in our online appendix.

RQ$_2$: What is the complementarity of different representations? Table 4 reports the complementarity metrics between the different representations. In particular, on the left side of Table 4 we report the intersection of sets of true positive candidates detected by the four representations. For example, 40% of the true positives are detected by both the AST and Identifier models. The relatively small overlap among the candidate sets, at both method and class granularity (no more than 51%), suggests that these representations complement each other.

The middle section of Table 4 shows the difference in the sets of true positive candidates detected by the four representations. The reported values show the percentage of true positive clones

Table 5: Spearman's rank correlation

$\rho(R_1, R_2)$	Ident	AST	CFG	Byte
Ident		0.094	0.120	0.069
AST			0.157	0.031
CFG				0.046
Byte				

Table 6: Performance of the *CloneDetector*

	Methods			Classes		
	Precision %	Recall %	F-Measure %	Precision %	Recall %	F-Measure %
Clone	98	97	98	90	93	91
Not Clone	90	91	90	61	52	56
Weighted Avg.	96	96	96	85	86	85

detected by a certain representation and missed by the other. For example, 48% of the true positive clones identified by the AST model at class level are not identified by the Identifier-based model.

Finally, the left section of Table 4 shows the percentage and number of instances (in parenthesis) of true positive candidates identified by each representation and missed by all the others. From these results, we note that there are candidates exclusively identified by a single representation. Note that the number of instances and percentages are computed only on the manually validated sample.

Table 5 shows the Spearman's Rank correlation coefficient (ρ) for the different representations. While all the computed correlations are statistically significant, their low value suggest that distances computed with different representations do not correlate and provide different perspectives about the similarity of code pairs. Indeed, the "strongest" correlation we observe is between the AST and the CFG representations, which is still limited to 0.157 (basically, no correlation).

For example, the classes `MessageDestination` and `ContextEjb` were detected only by the AST representation. They both extend `ResourceBase` and offer the same features. These classes share only a few identifiers (therefore not detected by the Identifier model) and differ in some if-structures, making them more difficult to detect by CFG and Bytecode representations. This and other examples are available in our online appendix [2].

RQ$_3$: How effective are combined multi-representation models? Table 6 reports the results of the *CloneDetector* at class- and method-levels. When identifying clone instances, the model achieves 90% precision and 93% recall at class, and 98% precision 97% recall at method level. The lower performance at class-level could be due to the smaller dataset available (*i.e.,* 483 methods *vs* 362 classes). The overall classification performance (*i.e.,* the weighted average of the Clone/Not Clone categories) is also lower for the class-level (~85% F-Measure for classes, and ~96% for methods). We also trained the model by considering a subset of the representations. In our online appendix, we provide results considering all possible subsets of features. We found that single-representation models tend to have worse performance than the combined model, which was trained on all representations. However, the combinations with Identifiers+AST+{CFG *or* Bytecode} obtain results similar to the overall model trained on all representations.

Table 7 shows the results of the *CloneClassifier* at class- and method-levels. At class level, the *CloneClassifier* has overall F-Measure of 68%, with average precision of 67% and recall of 68%. At method level, the model obtains an overall F-Measure of 84%, with an average precision and recall of 84% across the different categories (*i.e.,* Not Clone and the four types of detected clones). As expected, the category with the lowest precision and recall is the Type IV clones for both method and class level, while other clone types achieve a precision ≥ 74% and a recall ≥ 75%. Single-representation models obtain significantly worse results in the classification task (with a bigger gap with respect to what observed for the *CloneDetector*). All the results are available in our online appendix.

Table 7: Performance of the *CloneClassifier*

	Methods			Classes		
	Precision %	Recall %	F-Measure %	Precision %	Recall %	F-Measure %
Not Clone	89	94	91	59	61	60
Type I	89	88	88	86	78	82
Type II	82	84	83	81	85	83
Type III	74	75	75	61	59	60
Type IV	67	18	29	00	00	00
Weighted Avg.	84	84	84	67	68	68

RQ$_4$: Are DL-based models applicable for detecting clones among different projects? We identified several groups of duplicate code across different Apache commons libraries in the dataset. The largest group of clones is between the libraries `lang3-3.6` and `text-1.1`. The duplicated code involves classes operating on Strings, for example: StringMatcher, StrBuilderWriter, StrTokenizer, StrSubstitutor *etc.*, for a total of 21 shared similar classes identified. We also identified another group of similar classes between `text-1.1` (package `diff`) and `collections4-4.1` (package `sequence`). In particular, the class StringsComparator in `text-1.1` appears to be a Type III clone of the class SequencesComparator in `collections4-4.1`. An example of Type II clone within these two libraries is instead the pair EditScript and ReplacementsFinder classes. These clones are classified as Type II, since only package information has been changed.

The libraries `math3-3.6.1` and `rng-1.0` contain two shared classes: ISAACRandom and Gamma/InternalGamma. ISACCRandom is a pseudo-random number generator. The classes share many similarities across the two libraries, however, they differ on a statement and structural level. As an example, both classes implement a method which sets a seed. In one class, the seed is set by an integer passed through an argument, while the other class sets the seed by using a combination of the current time and system hash code of the instance. Gamma and InternalGamma, named respectively to the library they belong to, are clones since parts of the Gamma class were used to develop InternalGamma. The developers only took the needed functionalities out of the Gamma class, stripped it of unnecessary code, and built InternalGamma. Therefore, many of the methods in Gamma either do not appear or are significantly smaller in InternalGamma. Despite InternalGamma being significantly smaller than Gamma, our tool was still able to detect similarity between the two classes.

The libraries `codec-1.9` and `net-3.6` share the same implementation for the class Base64, providing encoding/decoding features. Note that the classes mentioned in this study do not refer to imported libraries but to actual Java duplicated code, *i.e.,* the source code files are in both libraries.

We also identified false positives in this study, mainly due to small inner classes and enumerators. Enumerators have a very similar bytecode structure even if containing different constants. They are uninteresting with respect to the goal of this scenario.

Table 8: Model Reusability

Project	Methods %		Classes %	
	$L_R \in L_O$	$L_O \in L_R$	$L_R \in L_O$	$L_O \in L_R$
ant-1.8.2	99	88	73	31
antlr-3.4	100	100	33	100
argouml-0.34	99	96	97	73
hadoop-1.1.2	99	95	95	74
hibernate-4.2.0	89	82	30	84
jhotdraw-7.5.1	99	98	82	77
maven-3.0.5	97	84	50	100
pmd-4.2.5	97	99	99	99
tomcat-7.0.2	98	97	87	69
Overall	97	93	58	90

Imported and shaded classes. We identified a large list of shared classes between the library j (`weaver-1.3`) and the following libraries in the dataset L: `collections4-4.1` (373 classes), `lang3-3.6` (79), and `io-2.5` (13). A closer inspection of the building files of `weaver-1.3` showed that the aforementioned libraries have been imported and shaded. That is, the dependencies have been included and relocated in a different package name in order to create a private copy that `weaver-1.3` bundles alongside its own code.

RQ$_5$: Can trained DL-based models be reused on different, previously unseen projects? Table 8 shows the percentage of candidates in L_R that are also in L_O and vice versa, both at method- and class-level. Generally, the list of candidates identified by the reused model and the original models tend to be similar. At method-level, we can see that 97% of the candidates identified by the reused model were also identified by the original model. Similarly, 93% of the candidates returned by the original model are identified by the reused model. At class-level we notice smaller percentages. This is mostly due to the fact that fewer clones are identified at the class-level. For example, for `antlr-3.4` the reused model identifies three candidates while the original model only identifies one. For `maven-3.0.5`, two candidates are identified by the reused model and only one by the original model. Still, 90% of the class-level candidates identified by the original models are detected by the reused model.

We also show that combined models can be reused on different systems. The *CloneDetector* model has been trained only on the data available for one project (`hibernate`) and tested on all the instances of the remaining projects. It achieved 98% precision and 92% recall at method-level and 99% precision and 95% recall at class-level.

6 THREATS TO VALIDITY

Construct validity. The main threat is related to how we assess the complementarity of the code representations. We support this claim by performing different analyses: (i) complementarity metrics; and (ii) correlation test.

Internal validity. This is related to possible subjectiveness when evaluating similarities of code fragments. To mitigate such threat, we employed three evaluators who independently checked the candidates. Then, we computed two-judge agreement on the evaluated candidates. We also qualitatively discuss false positives and borderline cases. Also, all our evaluations are publicly available [2].

External validity. The results obtained in our study using the selected datasets might not generalize to other projects. To mitigate this threat, we applied our approach in different contexts and used two different datasets; *Projects* and *Libraries*. For *Projects*, which is a subset of systems from the Qualitas.class corpus, we selected diverse systems in terms of size and domain, focusing on popular ones. All the *Libraries*, including all the *Apache commons* libraries, are publicly available, ensuring the replicability of the study. We did not utilize other clone-focused datasets (*e.g.,* such as BigCloneBench [66]) since in order to extract some representations (*i.e.,* CFG and Bytecode) we needed compiled code. Another threat in this category is related to the fact that we apply our approach on Java code only. While the representation extraction steps are implemented for Java, all the subsequent steps are completely language-agnostic because they rely on a corpus of sentences with arbitrary tokens. As a matter of fact, Recursive Autoencoders have been used in several contexts with different inputs such as natural language, source code, images. We do not compare our approach against code clone detection techniques since the focus of this paper is to show a general technique on how to learn similarities from multiple code representations, rather than building a code clone detector tool. Last, but not least, we focused on four code representations, but there may be others that are worthwhile to investigate (*e.g.,* data-flow or program dependency graphs).

7 CONCLUSION

In this paper, we show that code similarities can be learned from diverse representations of the code, such as Identifiers, ASTs, CFGs and bytecode. We evaluated the performance of each representation for detecting code clones and show that such representations are orthogonal and complementary to each other. We also show that our model is reusable, therefore, avoiding to retrain the DL approach so that it is project specific. This eliminates a large timesink, native to DL approaches, and broadens the applicability of our approach.

Moreover, combined models relying on multiple representations can be effective in code clone detection and classification. Additionally, we show that Bytecode and CFG representations can be used for library provenance and code maintainability. These findings speak to the vast amount of SE tasks which benefit from analyzing multiple representations of the code. We instantiated our approach in different use case scenarios and datasets.

Our approach inherently highlights the benefits of not only single code representations, but the combinations of these representations. This work also emphasizes the attributes that different code representations can accentuate. This allows for a more targeted choice by SE researchers of a code representation when applying representation-DL algorithms to a SE tasks. We believe that learning similarities from different representations of the code, without manually specifying features, has broad implications in SE tasks, and is not limited solely to clone detection.

8 ACKNOWLEDGMENT

This work was performed in part using computing facilities at the College of William and Mary which were provided by contributions from the National Science Foundation, the Commonwealth of Virginia Equipment Trust Fund, and the Office of Naval Research. This research was supported in part via NSF CAREER CCF-1253837 and CCF-1525902 grants. Gabriele Bavota gratefully acknowledges the financial support of the Swiss National Science Foundation for the JITRA projects (SNF Project No. 172479).

REFERENCES

[1] 2017. Apache Commons Project Distributions: https://archive.apache.org/dist/commons/. (2017).

[2] 2017. Online Appendix: https://sites.google.com/view/learningcodesimilarities, Source Code: https://github.com/micheletufano/AutoenCODE. (2017).

[3] 2017. Soot: https://github.com/Sable/soot. (2017).

[4] M. Allamanis, E. Barr, C. Bird, and C. Sutton. [n. d.]. Suggesting Accurate Method and Class Names *(FSE'15)*.

[5] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. 2091–2100. http://jmlr.org/proceedings/papers/v48/allamanis16.html

[6] Miltiadis Allamanis and Charles A. Sutton. 2014. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 472–483.

[7] B. Baker. [n. d.]. On Finding Duplication and Near-duplication in Large Software Systems *(WCRE'95)*.

[8] B. Baker. 1992. A program for identifying duplicated code. In *Computer Science and Statistics*.

[9] B. Baker. 1996. Parameterized Pattern Matching: Algorithms and Applications. *JCSS* 52, 1 (1996).

[10] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The Plastic Surgery Hypothesis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 306–317. https://doi.org/10.1145/2635868.2635898

[11] Gabriele Bavota, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. 2013. An Empirical Study on the Developers&#039; Perception of Software Coupling. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 692–701. http://dl.acm.org/citation.cfm?id=2486788.2486879

[12] Gabriele Bavota, Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea de Lucia. 2014. Improving Software Modularization via Automated Analysis of Latent Topics and Dependencies. *ACM Trans. Softw. Eng. Methodol.* 23, 1, Article 4 (Feb. 2014), 33 pages. https://doi.org/10.1145/2559935

[13] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. 2014. Methodbook: Recommending Move Method Refactorings via Relational Topic Models. *IEEE Trans. Softw. Eng.* 40, 7 (July 2014), 671–694. https://doi.org/10.1109/TSE.2013.60

[14] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. [n. d.]. Clone Detection Using Abstract Syntax Trees *(ICSM'98)*.

[15] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2016. Testing Advanced Driver Assistance Systems Using Multi-objective Search and Neural Networks. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/2970276.2970311

[16] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation Learning: A Review and New Perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.* 35, 8 (Aug. 2013), 1798–1828. https://doi.org/10.1109/TPAMI.2013.50

[17] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (01 Oct 2001), 5–32. https://doi.org/10.1023/A:1010933404324

[18] K. Chen, P. Liu, and Y. Zhang. [n. d.]. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets *(ICSE'14)*.

[19] Bogdan Dit, Latifa Guerrouj, Denys Poshyvanyk, and Giuliano Antoniol. 2011. Can Better Identifier Splitting Techniques Help Feature Location?. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension (ICPC '11)*. IEEE Computer Society, Washington, DC, USA, 11–20. https://doi.org/10.1109/ICPC.2011.47

[20] Bogdan Dit, Evan Moritz, Mario Linares-Vásquez, Denys Poshyvanyk, and Jane Cleland-Huang. 2015. Supporting and Accelerating Reproducible Empirical Research in Software Evolution and Maintenance Using TraceLab Component Library. *Empirical Softw. Engg.* 20, 5 (Oct. 2015), 1198–1236. https://doi.org/10.1007/s10664-014-9339-3

[21] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process* 25, 1 (2013), 53–95. https://doi.org/10.1002/smr.567

[22] Bogdan Dit, Meghan Revelle, and Denys Poshyvanyk. 2013. Integrating Information Retrieval, Execution and Link Analysis Algorithms to Improve Feature Location in Software. *Empirical Softw. Engg.* 18, 2 (April 2013), 277–309. https://doi.org/10.1007/s10664-011-9194-4

[23] S. Ducasse, M. Rieger, and S. Demeyer. [n. d.]. A Language Independent Approach for Detecting Duplicated Code *(ICSM'99)*.

[24] J.L. Fleiss et al. 1971. Measuring nominal scale agreement among many raters. *Psychological Bulletin* 76, 5 (1971), 378–382.

[25] Wei Fu and Tim Menzies. 2017. Easy over hard: a case study on deep learning. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 49–60.

[26] M. Gabel, L. Jiang, and Z. Su. [n. d.]. Scalable Detection of Semantic Clones *(ICSE'08)*.

[27] C. Goller and A. Küchler. [n. d.]. Learning Task-Dependent Distributed Representations by Backpropagation Through Structure *(ICNN'96)*.

[28] Palash Goyal and Emilio Ferrara. 2017. Graph Embedding Techniques, Applications, and Performance: A Survey. *CoRR* abs/1705.02801 (2017).

[29] X. Gu, H. Zhang, D. Zhang, and S. Kim. [n. d.]. Deep API Learning *(FSE'16)*.

[30] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning.. In *AAAI*. 1345–1351.

[31] Vincent J. Hellendoorn and Premkumar T. Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 763–773.

[32] Xuan Huo, Ming Li, and Zhi-Hua Zhou. 2016. Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI'16)*. AAAI Press, 1606–1612. http://dl.acm.org/citation.cfm?id=3060832.3060845

[33] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. [n. d.]. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones *(ICSE'07)*.

[34] J. Johnson. [n. d.]. Identifying Redundancy in Source Code Using Fingerprints *(CASCON'93)*.

[35] J. Johnson. [n. d.]. Substring Matching for Clone Detection and Change Tracking *(ICSM'94)*.

[36] J. Johnson. [n. d.]. Visualizing Textual Redundancy in Legacy Source *(CASCON'94)*.

[37] T. Kamiya, S. Kusumoto, and K. Inoue. 2002. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *TSE* 28, 7 (2002).

[38] R. Komondoor and S. Horwitz. [n. d.]. Using Slicing to Identify Duplication in Source Code *(SAS'01)*.

[39] R. Koschke, R. Falke, and P. Frenzel. [n. d.]. Clone Detection Using Abstract Syntax Suffix Trees *(WCRE'06)*.

[40] J. Krinke. [n. d.]. Identifying Similar Code with Program Dependence Graphs *(WCRE'01)*.

[41] A. Lam, A. Nguyen, H. Nguyen, and T. Nguyen. [n. d.]. Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports *(ASE'15)*.

[42] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2017. Bug localization with combination of deep learning and information retrieval. In *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*. 218–229.

[43] Yann LeCun, Koray Kavukcuoglu, and Clément Farabet. 2010. Convolutional networks and applications in vision. In *International Symposium on Circuits and Systems (ISCAS 2010), May 30 - June 2, 2010, Paris, France*. 253–256.

[44] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. 2006. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *TSE* 32, 3 (2006).

[45] C. Liu, C. Chen, J. Han, and P. Yu. [n. d.]. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis *(KDD'06)*.

[46] Andrian Marcus and Jonathan I. Maletic. 2001. Identification of High-Level Concept Clones in Source Code. In *16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA*. 107–114.

[47] Collin McMillan, Mark Grechanik, and Denys Poshyvanyk. 2012. Detecting Similar Software Applications. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 364–374. http://dl.acm.org/citation.cfm?id=2337223.2337267

[48] T. Mikolov. 2012. *Statistical Language Models Based on Neural Networks*. Ph.D. Dissertation.

[49] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. [n. d.]. Distributed Representations of Words and Phrases and their Compositionality.

[50] Narcisa Andreea Milea, Lingxiao Jiang, and Siau-Cheng Khoo. 2014. Vector Abstraction and Concretization for Scalable Detection of Refactorings. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 86–97. https://doi.org/10.1145/2635868.2635926

[51] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI'16)*. AAAI Press, 1287–1293. http://dl.acm.org/citation.cfm?id=3015812.3016002

[52] Anh Tuan Nguyen and Tien N. Nguyen. 2015. Graph-based Statistical Language Model for Code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 858–868. http://dl.acm.org/citation.cfm?id=2818754.2818858

[53] Anh Tuan Nguyen and Tien N. Nguyen. 2017. Automatic categorization with deep neural network for open-source Java projects. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*. 164–166.

[54] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-based Mining of Multiple Object Usage Patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*. ACM, New York, NY, USA, 383–392. https://doi.org/10.1145/1595696.1595767

[55] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. 2016. Asymmetric Transitivity Preserving Graph Embedding. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. ACM, New York, NY, USA, 1105–1114. https://doi.org/10.1145/2939672.2939751

[56] Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. 2013. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 522–531.

[57] Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. 2016. Parameterizing and Assembling IR-Based Solutions for SE Tasks Using Genetic Algorithms. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. 314–325. https://doi.org/10.1109/SANER.2016.97

[58] Meghan Revelle, Bogdan Dit, and Denys Poshyvanyk. 2010. Using Data Fusion and Web Mining to Support Feature Location in Software. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension (ICPC '10)*. IEEE Computer Society, Washington, DC, USA, 14–23. https://doi.org/10.1109/ICPC.2010.10

[59] Chanchal Kumar Roy and James R. Cordy. 2008. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*. 172–181.

[60] Hitesh Sajnani and Cristina Lopes. 2013. A Parallel and Efficient Approach to Large Scale Clone Detection. In *Proceedings of the 7th International Workshop on Software Clones (IWSC '13)*. IEEE Press, Piscataway, NJ, USA, 46–52. http://dl.acm.org/citation.cfm?id=2662708.2662719

[61] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 1157–1168. https://doi.org/10.1145/2884781.2884877

[62] R. Socher, C. Lin, A. Ng, and C. Manning. [n. d.]. Parsing Natural Scenes and Natural Language with Recursive Neural Networks *(ICML'11)*.

[63] R. Socher, J. Pennington, E. Huang, A. Ng, and C. Manning. [n. d.]. Semi-supervised Recursive Autoencoders for Predicting Sentiment Distributions *(EMNLP'11)*.

[64] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. Manning, A. Ng, and C. Potts. [n. d.]. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank *(EMNLP'13)*.

[65] Student. 1921. An Experimental Determination of the Probable Error of Dr Spearman's Correlation Coefficients. *Biometrika* 13, 2/3 (1921), 263–282. http://www.jstor.org/stable/2331754

[66] J. Svajlenko and C. Roy. 2015. Evaluating Clone Detection Tools with BigCloneBench *(ICSME'15)*.

[67] Jeffrey Svajlenko, Chanchal K. Roy, and James R. Cordy. 2013. A mutation analysis based benchmarking framework for clone detectors. In *Proceeding of the 7th International Workshop on Software Clones, IWSC 2013, San Francisco, CA, USA, May 19, 2013*. 8–9.

[68] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S. Bigonha. 2013. Qualitas.class Corpus: A Compiled Version of the Qualitas Corpus. *Software Engineering Notes* 38, 5 (2013), 1–4.

[69] Mario Linares Vásquez, Andrew Holtzhauer, and Denys Poshyvanyk. 2016. On automatically detecting similar Android apps. In *24th IEEE International Conference on Program Comprehension, ICPC 2016, Austin, TX, USA, May 16-17, 2016*. 1–10. https://doi.org/10.1109/ICPC.2016.7503721

[70] S. Wang, T. Liu, and L. Tan. [n. d.]. Automatically Learning Semantic Features for Defect Prediction *(ICSE'16)*.

[71] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. 2013. Searching for better configurations: a rigorous approach to clone evaluation. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. 455–465.

[72] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. 2013. Searching for Better Configurations: A Rigorous Approach to Clone Evaluation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 455–465. https://doi.org/10.1145/2491411.2491420

[73] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*. 3034–3040. https://doi.org/10.24963/ijcai.2017/423

[74] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. [n. d.]. Deep Learning Code Fragments for Code Clone Detection *(ASE'16)*.

[75] W. Yang. 1991. Identifying Syntactic Differences Between Two Programs. *SPE* 21, 7 (1991).

[76] Zhi-Hua Zhou. 2009. *Ensemble Learning*. Springer US, Boston, MA, 270–273. https://doi.org/10.1007/978-0-387-73003-5_293