A Machine Learning Approach for Source Code Similarity via Graph-focused Features

Giacomo Boldini[1 [0009−0006−4741−0033]], Alessio Diana[1 [0009−0001−4546−5915]], Vincenzo Arceri[1 [0000−0002−5150−0393]], Vincenzo Bonnici[1 [0000−0002−1637−7545]], and Roberto Bagnara[1 [0000−0002−6163−6278]]

Department of Mathematical, Physical and Computer Sciences, University of Parma, Parco Area delle Scienze, 53/A, 43124 Parma, Italy
{giacomo.boldini,alessio.diana}@studenti.unipr.it,
{vincenzo.arceri,vincenzo.bonnici,roberto.bagnara}@unipr.it

Abstract. Source code similarity aims at recognizing common characteristics between two different codes by means of their components. It plays a significant role in many activities regarding software development and analysis which have the potential of assisting software teams working on large codebases. Existing approaches aim at computing similarity between two codes by suitable representation of them which captures syntactic and semantic properties. However, they lack explainability and generalization for multiple languages comparison. Here, we present a preliminary result that attempts at providing a graph-focused representation of code by means of which clustering and classification of programs is possible while exposing explainability and generalizability characteristics.

Keywords: Code Similarity · machine learning · Graph-focused Features · Control-flow Graph

1 Introduction

Source code similarity aims at recognizing common characteristics between two different codes by means of their components. It plays a significant role in many activities regarding software development and analysis, which include plagiarism detection [27], malicious code detection and injection [6,3,4], clone recognition [14], bug identification [12], and code refactoring. In order to obtain precise and reliable code similarity tools, they cannot be based just on the code syntactic structure, but also semantics must be taken into account. For instance, a naive code similarity measure could be given by comparing the syntactic structure of the two compared program fragments (e.g., plain text, keywords, tokens, API calls). Nevertheless, such a trivial similarity technique does not sufficiently capture the semantics of the programs. In particular, two program fragments can have different syntactic structures but they implement the same functionality. On the opposite, they can have similar syntax but they differ in their behaviors. Thus, more sophisticated representations of the fragments composing a program, and measures based on such representations, are required.

Due to the wide range of features that can be taken into account in this context, it is possible to fine-tune the information embedded in such representations. Based on [24], code similarity is classically split into four main types of increasing complexity, such that each type subsumes the previous one: *Type I* represents a complete syntactic similarity, modulo blank characters, comments, and indentation; *Type II* represents a syntactic similarity, modulo identifiers renaming, literals, and types; *Type III* represents copied fragments with further modifications in statements; *Type IV* represents a functional similarity, i.e., semantic similarity.

At different levels, each type of technique aims at detecting if a program fragment is a *code clone* of another one. From here on, we refer to code clones as portions of code that share some level of similarity with other code segments. The type of clone is defined based on the level of similarity between the fragments.

Tools for identifying code clones are categorized based on the method used to represent source code and the technique for comparing them. The most common traditional categories include those based on text, token, tree, graph, metrics, and hybrid methods [7,24,25]. Recently, there has been a growing interest in applying Machine Learning (ML) approaches, specifically Deep Learning (DL) techniques, to identify code clones [15]. These methods are particularly useful for detecting *Type III* and *Type IV* clones, which are the most challenging to identify. They employ different neural network architectures (e.g., DNN, GNN, RvNN), source code representations (e.g., Abstract Syntax Tree, Control-flow Graph, Data Flow Graph, Program Dependence Graph, or combination of them) and embedding techniques, such as word2vec [18], code2vec [2], graph2vec [20], and RAE [13].

In this paper, we introduce novel techniques for tackling the code similarity problem. Our approach relies on both Control-flow Graph (CFG) [8] and the LLVM Intermediate Representation (LLVM-IR) concepts for representing source code. Furthermore, we employ both supervised and unsupervised Machine Learning methodologies to conduct code similarity analysis.

It is worth noting that the proposed code representation does not focus solely on *Type III* or *Type IV* similarities. Instead, it captures both syntactic and semantic features of the code, allowing for the detection of various types of code clones, exhibiting similarities in terms of design and functionality. By combining these different approaches, the proposed methods could identify a broader range of code clones than traditional techniques that focus solely on either syntax or semantics. In particular, CFG encodes the semantics flow of a program of interest, i.e., how control flows through each code element. Hence, it does not *completely* capture the program semantics (e.g., it does not capture how values flow within the program). Nevertheless, such a representation does not completely fit for *Type IV* similarity, since complete semantic equivalence is required by its definition. Still, it subsumes *Type III* similarity, since it CFG fully encodes the syntax, besides describing the control flow. In addition, for each source language, catching *Type III* similarity is affected by how the language-specific constructs are mapped into LLVM-IR instructions.

Our proposed approach focuses on pursuing an explainable approach to code similarity analysis. Specifically, features used for code similarity can be remapped to fragments of original source code. This provides a more transparent and interpretable analysis: for example, it could be used during the study of the feature importance of the models to refer directly to portions of source code. Furthermore, this can enable a deeper understanding of the similarities and differences between code fragments.

The rest of the manuscript is organized as follows. In Section 2 we present the three main phases of the proposed approach to extract graph-based features from `C`/`C++` source code. In addition, both supervised and unsupervised Machine Learning methodologies used are explained here. In Section 3 we show some of the obtained results using both clustering and classification methods. In Section 4 we discuss all the pros and cons of using this methodology. Section 5 concludes the paper.

2 Methods

In this section, we describe the methods adopted in our solution for facing the code similarity problem. This study focuses on the code similarity of `C`/`C++` translation units.[1] Nevertheless, as we will discuss in Section 4, the proposed solution can be also adapted to other programming languages. Before going into details of our contribution, in the following, we describe the toolchain of our solution, from a high-level point of view.

The overall architecture is depicted in Figure 1. The process consists of three main steps. The first step corresponds to a preprocessing phase that prepares the input source code for further analysis. In particular, this phase translates the program to a lower-level language, namely LLVM-IR [26], an intermediate representation used by the LLVM compiler to represent the source code during all the compilation phases. LLVM-IR is usually employed by compilers to describe and store all the information retrieved from the source code in order to perform a more precise translation into the target language.

The output of this phase is a set of CFGs derived from the LLVM-IR representation of the program of interest, each corresponding to a function/method of the input program. They are the inputs of the second step, which manipulates and enriches the basic CFG structure in order to obtain the so-called Augmented Control-flow Graphs (A-CFGs). This phase also manages the construction of the Augmented Call Graph (A-CG), namely a single graph made of A-CFGs, where also calling relationships are tracked. From this graph structure, in the last phase,

[1] In `C`/`C++`, executable programs are obtained by linking together the code coming from a complete set of *translation units*. A *translation unit* is the portion of a program a compiler operates upon, and is constituted by a main file (typically with a `.c`/`.C`/`.cxx`/`.cpp` extension) along with all header files (typically with a `.h`/`.H`/`.hpp` extension) that the main file includes, directly or indirectly. A prerequisite of our approach is that the translation units to be analyzed are complete, so that the Clang compiler can process them without errors.

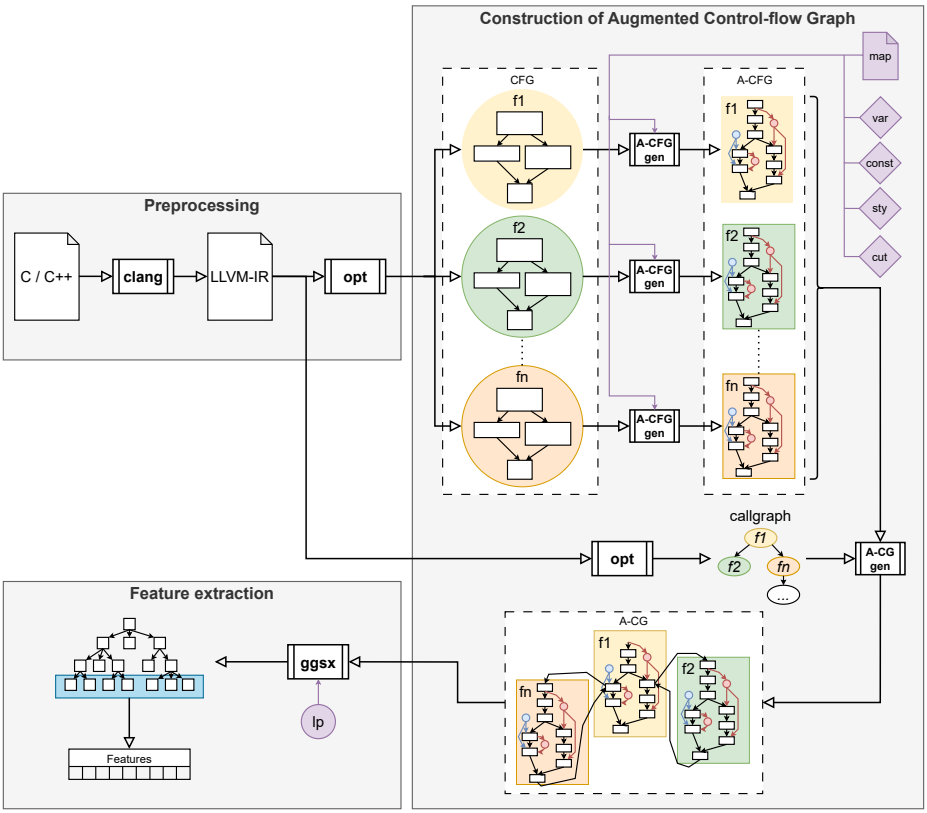


Fig. 1: Features extraction method split in its three main phases: preprocessing, construction of the A-CFGs/A-CG and feature extraction.

we rely on GraphGrepSX [5] for the graph features extraction. Source code is represented by the sets of features previously extracted, used by clustering and classification methods to reveal similarities between programs.

2.1 Preprocessing

The preprocessing stage involves two steps, both implemented by tools coming from the LLVM toolchain.

The first step involves the compilation of a `C`/`C++` translation unit to its corresponding LLVM-IR representation. In order to do this, we rely on the Clang compiler. During the compilation process, Clang is instructed to compile to LLVM-IR language, instead of the default object code. Moreover, plain names (the ones appearing in the original source code) are preserved during this compilation phase. Additionally, Clang is run with all optimizations switched off, so as to ensure that the output code preserves the structure and functionalities

of the source code. These steps are crucial for enhancing the CFG and extracting features in the subsequent phases, making it easier to analyze and extract information for our purposes.

From the LLVM-IR code, the second phase manages the construction of CFGs. A CFG, in LLVM flavour, is a directed graph $G = (N, E)$ where nodes N are basic blocks, and edges $E \subseteq N \times N$ connecting basic blocks correspond to jumps in the control flow between the two linked basic blocks. Each basic block starts with a label (giving the basic block a symbol table entry), contains a list of instructions to be executed in sequence, and ends with a terminator instruction (such as a branch or function return) [26]. In order to generate CFGs, we rely on `opt`, which is a LLVM tool useful to perform optimization and analysis of LLVM-IR code. Finally, we rely on `opt` also to create the call graph [8] (CG), a graph that tracks calling relationships between functions in the program.

2.2 Construction of the Augmented Control-flow Graphs

In this phase, a new graph data structure, called Augmented Control-flow Graph (A-CFG) is built to enhance the information of a traditional CFG.

A A-CFG resembles a single-instruction CFG, where each basic block of the A-CFG corresponds to a single instruction. In order to obtain this, each CFG is analyzed individually. Each basic block of the original CFG is exploded, in the corresponding A-CFG, into a sequence of single LLVM-IR instructions, connected by sequential edges and preserving the original control flow. We call these nodes *instruction nodes*, and we call the edges connecting instructions nodes *flow edges*.

After this initial manipulation, further information is added to a A-CFG. In particular,[2] for each instruction node, variable updates (i.e., assignments) and reads, and used constants are retrieved, and for each variable and constant, a new node is added to the A-CFG. We refer to these nodes as *variable nodes*, which are also labeled with the variable type, and *constant nodes*, labeled with the constant value, respectively. Then, for each instruction node updating a variable x, an edge from the instruction node to the corresponding variable node x is created. Similarly, if the instruction node reads a variable x, an edge from the variable node x to the instruction node is added. The direction of the edge reflects whether the variable is written or read. Similarly, if an instruction node reads a constant, an edge connecting it to the corresponding constant node is added. We refer to these three types of edges as *data edges*.

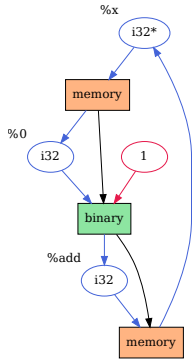
Instruction nodes labels are assigned through a surjective map $\mathsf{map} : O \to L$, where O denotes the set of LLVM-IR op-codes,[3] and L is an arbitrary set of labels. The *standard map* maps each specific LLVM-IR op-code (e.g., `add`, `store`) to one of the nine categories defined in the LLVM-IR language manual [26]:

[2] This phase is managed by using a custom LLVM-IR parser. The parser is generated using ANTLR [21] starting from the LLVM-IR 7.0.0 grammar.

[3] In LLVM-IR, op-code refers to the instruction code (or name) that specifies the operation to be performed by the instruction.

```
1  %0 = load i32, i32* %x, align 4
2  %add = add nsw i32 %0, 1
3  store i32 %add, i32* %x, align 4
```

(a) LLVM-IR



(b) part of A-CFG

Fig. 2: LLVM-IR code and portion of A-CFG for `C++` instruction `x = x + 1;`.

terminator, *binary*, *bitwise*, *vector*, *aggregate*, *memory*, *conversion*, *other* and *intrinsic*. For instance, the `add` op-code at line 2 in in Figure 2a, corresponds to the green node in the A-CFG reported in Figure 2b, where the standard map has mapped the op-code `add` to the label *binary*. Instead, the `load` and `store` op-codes at lines 1 and 3, respectively, are mapped in the orange nodes, where the standard map has mapped them to `memory`.

However, in general, a custom `map` function can be built acting on the set of labels L. Restricting or expanding L, starting from the set of the LLVM-IR instruction categories discussed above, one can fine-tune the level of detail regarding the information related to instruction nodes. This allows to meet the specific needs of the feature generation process, where nodes with the same label are indistinguishable.

Figure 2 shows how a part of a single basic block, reported in Figure 2a, is transformed into its corresponding A-CFG (Figure 2b) using the standard map discussed above. In particular, black edges represent flow edges, blue edges represent data edges concerning variables, and red edges represent data edges concerning constants.

It is also possible to specify whether to generate variable and constant elements, use simplified types, and remove unnecessary terminator nodes, setting the Boolean parameters `var`, `const`, `sty`, and `cut`, respectively. Such behavior is provided by specific parameters that can be activated/deactivated by the user.

The A-CFG structure has the granularity of a single function in a program, without tracking calling relationships between different A-CFGs. In order to represent an entire program, starting from the so-built A-CFGs, the idea is to create an Augmented Call Graph (A-CG), namely a new graph consisting of the union of the A-CFGs, linked together by following the function calls within it.

Specifically, for each instruction node n calling a function f, the following edges are added: one edge from n to the entry point instruction node of the A-CFG of the function f (it is unique) and one for each exit instruction node

(i.e., an instruction node terminating f) to n. This process is carried out by searching for calling instruction nodes and analyzing the original CG of the program, generated in the first phase.

2.3 Feature Extraction

The third and last phase consists of extracting all the features that will characterize the analyzed file during the execution of the Machine Learning models. The input of this phase is an augmented graph, both A-CFG and A-CG, created in the previous phase.

For a given graph G, the proposed method uses paths of G of bounded length as features. The indexing phase of GraphGrepSX [5] is used to build an index in a prefix-tree format for a graph database. The index is constructed by performing a depth-first search for each node n_j of the graph. During this phase, all paths of length lp or less are extracted, and each path is represented by the labels of its nodes. More formally, each path $(v_1, v_2, \ldots, v_{lp})$ is mapped into a sequence of labels $(l_1, l_2, \ldots, l_{lp})$. All the subpaths $(v_i, \ldots, v_j)$ for $1 \leq i \leq j \leq lp$ of a path $(v_1, v_2, \ldots, v_{lp})$ will be included in the global index, too. The number of times a path appears in the graph is also recorded.

In our proposal, a prefix-tree is built for a A-CG G. Then, only the paths of maximal length (i.e., equal to lp) are considered as features. Each feature is identified by a path p and its value corresponds to the number of occurrences of p in G. As a set of features describing the graph, we extract all the paths leading to the leaves at depth lp of the prefix tree.

2.4 Machine Learning

Let $\mathbb{F}^n$ be a set of features, with $\mathbb{F}^i$ being the i-th feature. Let $\mathbb{O} = \{o_1, o_2, \ldots, o_m\}$ be a set of vectors (or objects) such that $o_i \in \mathbb{F}^n$.

Let $d(o_i, o_j)$ be a real number representing the distance measure between objects o_i and o_j. Usually, distance function d is symmetric, $d(o_i, o_j) = d(o_j, o_i)$, positive separable, $d(o_i, o_j) = 0 \Leftrightarrow o_i = o_j$, and provides triangular inequality, $d(o_i, o_j) \leq d(o_i, o_k) + d(o_k, o_j)$. A clustering method is an unsupervised Machine Learning model which uses the distance function d between objects to organize data into groups, such that there is high similarity (low distance) within members in each group and low similarity across the groups. Each group of data represents a cluster. There are several approaches to clustering, one of which is agglomerative hierarchical clustering. This method works by iteratively merging the closest pair of data points or clusters until all data points belong to a single cluster. The distance between two clusters is computed using the distance function d between data points and the agglomeration method. The resulting cluster hierarchy, or dendrogram, can be cut at different levels to obtain different sets of clusters.

Let $\mathbb{C} = \{c_1, c_2, \ldots, c_t\}$ be a set of classes such that an surjective function $c : \mathbb{O} \mapsto \mathbb{C}$ is defined. Classification is a supervised Machine Learning model used to assign a class to a new object $o \notin \mathbb{O}$ by taking into example $\mathbb{O}$ and the

relation between the objects in $\mathbb{O}$ and their assigned classes, the method used to do this is defined by the model.

Let split $\mathbb{O}$ into two subsets $\mathbb{O}^T$ and $\mathbb{O}^V$ such that $\mathbb{O}^T \bigcap \mathbb{O}^V = \emptyset$, which are respectively called the training and the verification set. The training set is used to train a model, while the verification set is used for assessing the performance of the model in correctly assigning the classes to the objects in $\mathbb{O}^V$ without knowing their original/real class. It is essential that the balance between the classes of the original data set must be preserved in the training and verification sets.

Given a Machine Learning model trained on $\mathbb{O}^T$ for all the classes $\mathbb{C}$ and given an object $o \in \mathbb{O}^V$, let $\tilde{c}(o)$ be the class that the model assigns to o and $c(o)$ be the actual class for the object o. We distinguish the object in $\mathbb{O}^V$ into four sets, depending on the accordance between their real class i and the class that is assigned by the model, that are: true positives (TP_i), true negative (TN_i), false positives (FP_i) and false negatives (FN_i). They are defined as: $TP_i = ||\{o \in \mathbb{O}^V \mid c(o) = \tilde{c}(o) = i\}||, TN_i = ||\{o \in \mathbb{O}^V \mid c(o) \neq i \wedge \tilde{c}(o) \neq i\}||, FP_i = ||\{o \in \mathbb{O}^V \mid c(o) \neq i \wedge \tilde{c}(o) = i\}||, FN_i = ||\{o \in \mathbb{O}^V \mid c(o) = i \wedge \tilde{c}(o) \neq i\}||$. The accuracy of the model is defined as $(\sum_{i=1}^{t} ||TP_i||)/(|\mathbb{O}^V|)$. Precision and recall are defined for each class i as $|TP_i|/(|TP_i| + |FP_i|)$ and $|TP_i|/(|TP_i| + |FN_i|)$, respectively. The F1-score for each class i is defined as $(2|TP_i|)/(2|TP_i| + |FP_i| + |FN_i|)$. Because the problem is defined for multiple classes, we can compute a single value for these metrics (precision, recall, F1-score) by averaging over such classes. In particular, let $\mathbb{O}_i = \{o \in \mathbb{O}^V | c(o) = c_i\}$ a subset of verification set and M_i a performance metric (one of precision, recall, F1-score) computed on class c_i. The weighted average version of M uses class cardinality as weights and it is defined as $M = \frac{1}{|\mathbb{O}^V|} \sum_{c_i \in \mathbb{C}} |\mathbb{O}_i| \, M_i$.

A common approach for exploiting clustering results in a classification task is to assign a class label to each cluster based on the majority class of its constituent data points. Specifically, for each cluster, the class that has the highest number of objects within the cluster is selected as the assigned class label. If ground-truth labels are available, external evaluation measures can be used to evaluate clustering. One commonly used metric is the Rand Index (RI) [11]. It measures the agreement between two clusterings by comparing every pair of data points and counting the number of pairs that are grouped together or separately in both the predicted and ground-truth clusterings. It is defined as the ratio of the sum of agreements (pairs assigned to the same cluster in both clusterings) and disagreements (pairs assigned to different clusters in both clusterings) over the total number of pairs.

This work employs the SciPy library [28] for implementing clustering models, while the Scikit-learn [22] library is used for developing classification models and computing evaluation metrics.

3 Results

We experimentally evaluate our proposal by using it as a basis for clustering and classification models.

We consider as data set a subset of `C++` programs, included in the CodeNet project [23]. The CodeNet project provides a large collection of source files in several programming languages, such as `C`, `C++`, `Java` and `Python`, and for each of them there are extensive metadata and tools for accessing the dataset to select custom information. The samples come from online systems that allow users to submit a solution to a variety of programming problems, in the form of competitions or courses, ranging from elementary exercises to problems requiring the use of advanced algorithms. Each consists of a single file in which the test cases and printouts of the required results are included.

CodeNet includes several benchmark datasets, created specifically to train models and conduct code classification and similarity experiments. These datasets were obtained by filtering the original dataset in order to remove identical problems and nearly duplicate code samples, with the goal of obtaining better training and more accurate metrics [1]. In this work, we used a subset of C++1000 data set (available at https://developer.ibm.com/exchanges/data/all/project-codenet/) consisting of 1000 programming problems, each of them containing 500 `C++` programs (submissions).

With respect to the existing taxonomy, the proposed approach can be considered to be between type III and type IV. Thus, a direct comparison with state-of-the-art approaches is not suitable.

3.1 Clustering

In these experiments, 100 random submissions were considered for each of 10 randomly chosen problems of CodeNet dataset. For each (submission) program, the corresponding feature vector is generated using the following generation method's parameters: $lp = 3$, `const` = True, `var` = True, `sty` = True, `cut` = True. Also, four atomic categories for `store`, `load`, `phi`, and `call` instructions are added to the standard `map` function. Each clustering experiment is designed as follows: N random problems (from 10 available) are selected, whose corresponding submission's feature vectors are taken and merged into a single dataset; outliers are first removed from it using Isolation Forest (IF) [17] and then features are selected by using the Extra Tree Classifier (ETC) [9] supervised method (the ground-truth label are the problem to which each submission belongs); the distance matrix between objects is computed using the Boolean Jaccard index and then a hierarchical clustering algorithm from the SciPy library tries to find the clusters. For each experiment, four agglomeration methods were tried: *single*, *complete*, *average* and *ward* [19]. 10 experiments were conducted for each combination of N and for each agglomeration method.

Table 1 reports mean and standard deviation of Rand Index (RI) for all the clustering experiments, grouped by the the number of considered problems N and by the agglomeration method used in the clustering algorithm.

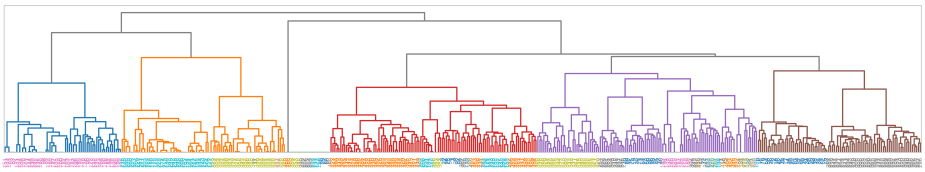


Fig. 3: Dendrogram of an experiment which considers $N = 6$ problems (about 600 submissions) and uses *ward* agglomeration method.

First of all, it shows that there is not actually a link between the variation in the problems considered (N) and the RI metric: in some agglomeration methods, there is a direct growth of both of them; in other methods, there is an inverse growth of them. Second, *ward* agglomeration generally performs better than all the other methods, despite N.

In addition, a dendrogram obtained from an experiment involving $N = 6$ problems (thus about 600 submissions) and using a *ward* agglomeration method is shown in Figure 3: each leaf in the tree represents a program; the color of the leaf represents the problem to which it belongs, while the color of the subtrees represent the clusters found by performing a cut to obtain at most N clusters.

It can be seen that the partitioning partially respects *Type IV* code similarity: some clusters are pure with regard to the problem they belong to, while others contain programs belonging to different problems. However, in the latter case, almost-pure subtrees are visible, suggesting that programs belonging to the same problem were joined in the same cluster in the early stages of the algorithm.

3.2 Classification

For the classification experiments, the A-CFG generation parameters for executing one experiment are fixed, which are $lp = 3$, `const` = True, `var` = False, `sty`

N	Single	Complete	Average	Ward
2	0.50 ± 0.00	0.50 ± 0.00	0.50 ± 0.00	0.89 ± 0.08
3	0.39 ± 0.02	0.39 ± 0.07	0.43 ± 0.12	0.68 ± 0.07
4	0.32 ± 0.02	0.53 ± 0.16	0.37 ± 0.09	0.71 ± 0.04
5	0.29 ± 0.03	0.63 ± 0.08	0.48 ± 0.14	0.76 ± 0.04
6	0.25 ± 0.02	0.66 ± 0.08	0.48 ± 0.11	0.81 ± 0.02
7	0.24 ± 0.02	0.73 ± 0.05	0.65 ± 0.07	0.81 ± 0.03
8	0.23 ± 0.01	0.79 ± 0.02	0.67 ± 0.07	0.83 ± 0.01
9	0.22 ± 0.02	0.81 ± 0.02	0.72 ± 0.05	0.85 ± 0.02
10	0.21 ± 0.01	0.83 ± 0.01	0.71 ± 0.04	0.87 ± 0.01

Table 1: Mean and standard deviation of RI measure regarding the clustering experiments. Results are grouped by the number of different problems solved by the input programs taken into account, N, and by agglomeration method.

	Accuracy	F1-score	Precision	Recall
KN	0.76 ± 0.03	0.76 ± 0.03	0.79 ± 0.03	0.76 ± 0.03
SVC (C=1)	0.80 ± 0.06	0.81 ± 0.06	0.83 ± 0.04	0.80 ± 0.06
NuSVC	0.77 ± 0.05	0.78 ± 0.05	0.81 ± 0.03	0.77 ± 0.05
DT	0.74 ± 0.05	0.74 ± 0.05	0.75 ± 0.05	0.74 ± 0.05
RF	0.86 ± 0.03	0.86 ± 0.03	0.87 ± 0.03	0.86 ± 0.03
MLP	0.84 ± 0.01	0.84 ± 0.01	0.85 ± 0.01	0.84 ± 0.01
GNB	0.59 ± 0.07	0.58 ± 0.07	0.69 ± 0.03	0.59 ± 0.07

Table 2: Results of the classification experiments by varying the type of employed classifier.

= False, `cut` = True and standard `map` function. As done in clustering experiments, here 10 random problems are used from the CodeNet dataset which are the classes during the classification. Every problem has around 100 elements. Table 2 shows mean and standard deviation of the accuracy, F1-score, precision and recall for all the classification methods tried: K-Neighbors (KN), Support Vector Classifier (SVC), Non linear Support Vector Classifier (NuSVC), Decision Tree (DT), Random Forest (RF), MultiLayer Perceptron (MLP), Gaussian Naive Bayes (GNB). All the metrics are computed by averaging the results of a 5-fold cross-validation process. F1-score, precision, and recall are computed as an average between the results of the single classes, weighted using the class cardinality. Almost all the methods obtain a relatively high score, but the best ones are Random Forest and MLP, with a value of accuracy higher than 0.84.

A new test was carried out by doing thousands of experiments with varying generation parameters. Figure 4 shows all methods' mean accuracy trend of all the experiments in regard to the values of the generation parameter analyzed (`var`, `const`, `cut`, `sty`), one in every plot. In particular, the growth of the value of lp is represented on the horizontal axis and the mean accuracy is represented on the vertical axis. The variables `const`, `sty`, and `cut` have almost the same mean accuracy for both the true and the false values of the parameter. On the other hand, the input parameter `var` presents a significant change in the mean accuracy. Not taking into account the elements related to variables generally leads to better results than using them. This can be explained by the fact that the classification methods focus on more important characteristics since the variables are omitted in the graph generation.

The time needed for the classification in relation to the value of the generation parameter lp is shown in Figure 5. Due to the 5-fold cross-validation process, each time corresponds to the sum of five training and five validation processes. Every line in the figure corresponds to the mean for every method for every parameter combination. The lighter color stripe represents 95% of the values around the mean value. The figure shows how MLP is the method with the highest time, below it, there is RF, and below them, we can find all the other methods. However, it can be seen that, for all the methods analyzed, the time increases when the value of lp increases.

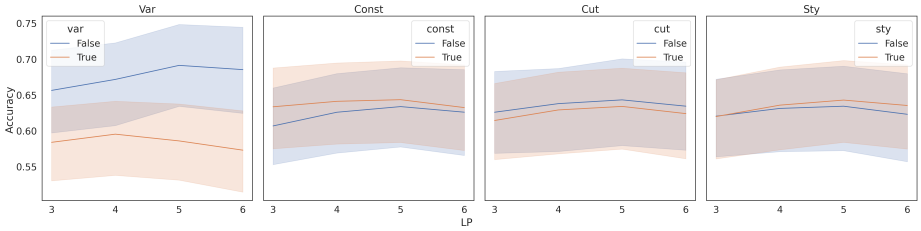


Fig. 4: Variation of the mean accuracy over different values of lp in respect to the change of the parameter values.

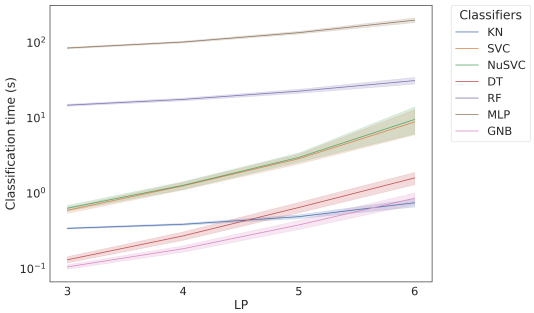


Fig. 5: Trend of log classification times for every classification method.

4 Discussion

The LLVM-IR code represents the actual starting point for the proposed features extraction method. For this reason, two considerations can be done. First, this methodology is easily extended to all those programs written in a language L for which an $L \rightarrow$ LLVM-IR compiler exists. This allows, secondly, a potential comparison between source codes written in different languages. To address this scenario, a more in-depth study on how language constructs are mapped to LLVM-IR is necessary.

The feature extraction phase proposed replaces classic embedding techniques used to describe the code graph representation in a lot of ML approaches seen before. In this case, a variable-length vector is created. It aims to lose as little information on the graph as possible. This set of features depends both on all the graph generation parameters and the path length lp used in feature extraction. In particular, the path length allows to describe the graph in different levels of detail, being more and more precise when the length of extracted path lp increases. Moreover, the map function allows the abstraction level of feature to be raised or lowered even further. This is because the labeling given by the function is reflected in the graph' instruction nodes, and thus in the extracted features. In this study, we analyzed small source code, however, it could be useful to use higher values of lp for analyzing bigger programs. To enable this

level of detail, extended versions of GraphGrepSX [10,16] will be needed for overcoming current running time and space limitations.

Differently from what happens in modern ML approaches to code similarity, our approach makes a step toward explainability. This is because the features created, during the extraction phase, can be remapped to a set of lines in the code fragment with a certain degree of approximation. By definition, each feature represents a number of identical paths within A-CFG/A-CG graphs, which mainly contain a succession of LLVM-IR instructions. For this reason, using compiler debugging information, it is possible to trace the set of lines of code that generated these successions of LLVM-IR instructions. This explainability feature gives the possibility to visualize which parts of the analyzed code fragments are more responsible for a high value of similarity between them.

5 Conclusions

Software complexity keeps growing, and the functions assigned to software are increasingly critical, in terms of safety and/or security. In addition, there is a general shortage of programmers, which are also characterized by a high turnover rate. While keeping track of all code in typical software projects is difficult in general, it is particularly difficult for developers joining the team at a later stage, and there are not enough senior developers to effectively mentor junior developers. As a result software projects frequently run late and/or enter production without a sufficient level of quality and maturity.

The techniques proposed in this paper for the identification of code similarities have the potential of assisting software teams working on larger codebases in a number of ways: identifying unwanted clones or code illegally copied in violation of open-source licenses; identifying vulnerabilities via similarity with code available in public vulnerability databases; assisting developers in performing tedious tasks (applying a learned recipe multiple times in multiple similar contexts); increasing developers' productivity by identifying regions of code that are amenable to the same treatment (reduction of context switches); capturing to some extent the knowledge of expert developers and project veterans and make it available to newcomers.

The proposed approach is based on a graph representation of the source code on top of which graph-focuses features are extracted for indexing it. In particular, atomic LLVM-IR instructions are ensembled in order to represent the control flow of the indexed program plus other suitable information aimed at better capturing the semantics of the code. Inspired by previous approaches in graph indexing techniques, our methodology extracts paths of fixed length from the formed graph and uses them as features of the indexed program. Because such paths correspond to specific portions of the input code, such features are used to characterize the indexed program as a whole, but they are also capable of identifying small portions of it. This aspect brings explainability to the overall machine learning approach that is applied for clustering programs and for classifying them by computing code similarity via such paths. An evaluation

of the proposed approach to an already existing data set of programs developed for solving different problems shows that it is possible to compute clusters of the programs that reflect the actual grouping of them with a feasible approximation. The evaluation also included supervised machine learning for classifying the programs according to the problem they are aimed at solving, showing promising results obtained by statistical evaluation of the performance. The proposed approach is not directly comparable to existing methodologies because it focuses on finding a type of source code similarity that is not currently recognized by any of the existing tools. However, the main advantages of it are the above-mentioned explainability and the fact that it works on LLVM-IR instructions. This restricts the similarity calculation to only those source codes that are free of compilation errors. but at the same time, it allows the approach to be potentially applied to any programming language that can be compiled to LLVM-IR. Thus, as a future work, we plan to evaluate its performance in clustering and classifying other programming languages, rather than `C`/`C++`, and in exploring a methodology for comparing programs developed in two different languages.

Acknowledgements This project has been partially founded by the University of Parma (Italy), project number MUR_DM737_B_MAFI_BONNICI. V. Bonnici is partially supported by INdAM-GNCS, project number CUP_E55F22000270001, and by the CINI InfoLife laboratory.

References

1. Allamanis, M.: The adverse effects of code duplication in machine learning models of code. In: Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. pp. 143–153 (2019). https://doi.org/10.1145/3359591.3359735
2. Alon, U., et al.: Code2vec: Learning distributed representations of code. Proc. ACM Program. Lang. **3**(POPL) (jan 2019). https://doi.org/10.1145/3290353
3. Arceri, V., Mastroeni, I.: Analyzing dynamic code: A sound abstract interpreter for *Evil* eval. ACM Trans. Priv. Secur. **24**(2), 10:1–10:38 (2021). https://doi.org/10.1145/3426470
4. Arceri, V., Olliaro, M., Cortesi, A., Mastroeni, I.: Completeness of abstract domains for string analysis of javascript programs. In: Hierons, R.M., Mosbah, M. (eds.) Theoretical Aspects of Computing - ICTAC 2019 - 16th International Colloquium, Hammamet, Tunisia, October 31 - November 4, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11884, pp. 255–272. Springer (2019). https://doi.org/10.1007/978-3-030-32505-3_15
5. Bonnici, V., et al.: Enhancing graph database indexing by suffix tree structure. In: Pattern Recognition in Bioinformatics: 5th IAPR International Conference, PRIB 2010, Nijmegen, The Netherlands, September 22-24, 2010. Proceedings 5. pp. 195–203. Springer (2010). https://doi.org/10.1007/978-3-642-16001-1_17
6. Dalla Preda, M., et al.: Abstract symbolic automata: Mixed syntactic/semantic similarity analysis of executables. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 329–341 (2015). https://doi.org/10.1145/2676726.2676986

7. Dhavleesh, R., et al.: Software clone detection: A systematic review. Information and Software Technology **55**(7), 1165–1199 (2013). https://doi.org/https://doi.org/10.1016/j.infsof.2013.01.008
8. Flemming, N., et al.: Principles of program analysis. Springer (2015). https://doi.org/10.1007/978-3-662-03811-6
9. Geurts, P., et al.: Extremely randomized trees. Mach. Learn. **63**(1), 3–42 (04 2006). https://doi.org/10.1007/s10994-006-6226-1
10. Giugno, R., et al.: Grapes: A software for parallel searching on biological graphs targeting multi-core architectures. PloS one **8**(10), e76911 (2013)
11. Hubert, L.J., Arabie, P.: Comparing partitions. Journal of Classification **2**, 193–218 (1985)
12. Jannik, P., et al.: Leveraging semantic signatures for bug search in binary programs. In: Proceedings of the 30th Annual Computer Security Applications Conference. pp. 406–415 (2014). https://doi.org/10.1145/2664243.2664269
13. Jie, Z., et al.: Fast code clone detection based on weighted recursive autoencoders. IEEE Access **7**, 125062–125078 (2019). https://doi.org/10.1109/ACCESS.2019.2938825
14. Krinke, J., Ragkhitwetsagul, C.: Code similarity in clone detection. In: Code Clone Analysis, pp. 135–150. Springer Singapore (2021). https://doi.org/10.1007/978-981-16-1927-4_10
15. Lei, M., et al.: Deep learning application on code clone detection: A review of current knowledge. Journal of Systems and Software **184**, 111141 (2022). https://doi.org/https://doi.org/10.1016/j.jss.2021.111141
16. Licheri, N., et al.: GRAPES-DD: exploiting decision diagrams for index-driven search in biological graph databases. BMC bioinformatics **22**, 1–24 (2021)
17. Liu, F.T., et al.: Isolation forest. In: 2008 Eighth IEEE International Conference on Data Mining. pp. 413–422 (2008). https://doi.org/10.1109/ICDM.2008.17
18. Mikolov, T., et al.: Efficient estimation of word representations in vector space. In: 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings (2013)
19. Müllner, D.: Modern hierarchical, agglomerative clustering algorithms (2011)
20. Narayanan, A., et al.: graph2vec: Learning distributed representations of graphs. CoRR **abs/1707.05005** (2017)
21. Parr, T.J., Quong, R.W.: Antlr: A predicated-ll(k) parser generator. Software: Practice and Experience **25**(7), 789–810 (1995). https://doi.org/https://doi.org/10.1002/spe.4380250705
22. Pedregosa, F., et al.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
23. Puri, R., et al.: Project codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. arXiv preprint arXiv:2105.12655 **1035** (2021)
24. Roy, C.K., Cordy, J.R.: A survey on software clone detection research. Queen's School of Computing TR **541**(115), 64–68 (2007)
25. Saini, N., et al.: Code clones: Detection and management. Procedia Computer Science **132**, 718–727 (2018). https://doi.org/https://doi.org/10.1016/j.procs.2018.05.080, international Conference on Computational Intelligence and Data Science
26. The LLVM Development Team: LLVMLanguage Reference Manual (Version 7.0.0) (2018)
27. Đurić, Z., Gašević, D.: A source code similarity system for plagiarism detection. The Computer Journal **56**(1), 70–86 (2013). https://doi.org/10.1093/comjnl/bxs018

28. Virtanen, P., et al.: SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods **17**, 261–272 (2020). https://doi.org/10.1038/s41592-019-0686-2