future internet

MDPI

Article

Analysis of Program Representations Based on Abstract Syntax Trees and Higher-Order Markov Chains for Source Code Classification Task

Artyom V. Gorchakov *, Liliya A. Demidova * and Peter N. Sovietov *

Institute of Information Technologies, Federal State Budget Educational Institution of Higher Education, MIREA—Russian Technological University, 78, Vernadsky Avenue, 119454 Moscow, Russia

* Correspondence: gorchakov@mirea.ru (A.V.G.); liliya.demidova@rambler.ru (L.A.D.); sovetov@mirea.ru (P.N.S.)

Abstract: In this paper we consider the research and development of classifiers that are trained to predict the task solved by source code. Possible applications of such task detection algorithms include method name prediction, hardware–software partitioning, programming standard violation detection, and semantic code duplication search. We provide the comparative analysis of modern approaches to source code transformation into vector-based representations that extend the variety of classification and clustering algorithms that can be used for intelligent source code analysis. These approaches include word2vec, code2vec, first-order and second-order Markov chains constructed from abstract syntax trees (AST), histograms of assembly language instruction opcodes, and histograms of AST node types. The vectors obtained with the forementioned approaches are then used to train such classification algorithms as k-nearest neighbor (KNN), support vector machine (SVM), random forest (RF), and multilayer perceptron (MLP). The obtained results show that the use of program vectors based on first-order AST-based Markov chains with an RF-based classifier leads to the highest accuracy, precision, recall, and F1 score. Increasing the order of Markov chains considerably increases the dimensionality of a vector, without any improvements in classifier quality, so we assume that first-order Markov chains are best suitable for real world applications. Additionally, the experimental study shows that first-order AST-based Markov chains are least sensitive to the used classification algorithm.

Keywords: program classification; Markov chains; abstract syntax trees; task classification; static analysis; k-nearest neighbor; support vector machine; random forest; multilayer perceptron

Citation: Gorchakov, A.V.; Demidova, L.A.; Sovietov, P.N. Analysis of Program Representations Based on Abstract Syntax Trees and Higher-Order Markov Chains for Source Code Classification Task. *Future Internet* **2023**, *15*, 314. https://doi.org/10.3390/fi15090314

Academic Editor: Massimo Cafaro

Received: 23 August 2023
Revised: 14 September 2023
Accepted: 15 September 2023
Published: 18 September 2023

1. Introduction

The ongoing digitalization leads to the expansion of tasks facing software developers. Static analysis tools are widely used in industrial software development, with applications in code simplification via automatic refactoring [1,2], code smell detection [3], non-idiomatic code search and elimination [4], and others. Modern research in program analysis is focused on the incorporation of machine learning and data mining algorithms into intelligent source code analyzers. Applications of such tools include intelligent code completion [5], defect and vulnerability search in software systems [6,7], recommendation of identifier names in integrated development environments [8,9], feature identification [10], synthesis of program transformation rules [11], and concept mining [12].

In this paper we consider the task detection problem. Known applications of task detection algorithms include malicious software identification [13,14], detection of semantic code clones [15,16], and hardware–software partitioning [17]. The task detection problem can be informally defined similarly to the algorithm detection problem described in [18]: given a finite set of different task types, and a program which solves a unique task of one of the given types, find the type of the task that a program solves. In our current research, we

consider the publicly available dataset [19] containing source codes of 13,881 small Python programs solving unique programming exercises of 11 different types; the unique exercises were automatically generated by the Digital Teaching Assistant (DTA) system [20]. The solutions to the exercises were submitted to DTA by Python programming course students, and the students were allowed to use any algorithm to solve a task of a given type. The programs from the dataset were successfully checked and accepted by the DTA system, and the classifiers were trained to predict the task type solved by a program based on its source code.

Prior to the application of a classification algorithm to the task detection problem, a feature extraction technique has to be applied to the source code dataset, with the aim of extracting only those source code features that are relevant in the considered problem domain. Source code transformation into vector-based representations is a widely applied feature extraction technique [7,9,18,21] that makes it possible to statically analyze software using machine learning algorithms such as multilayer perceptron [22], complex neural networks such as convolutional neural networks (CNN) [23], support vector machines [24], and other algorithms that require vectors at their input.

In this study we compare different approaches to source code transformation into vector-based representations, such as word2vec [21,25], code2vec [9,26,27], histograms of CPython assembly language instruction opcodes [18,28], histograms of abstract syntax tree (AST) node types, inspired by [28], and AST-based Markov chain models [15,16]. Additionally, we propose a general algorithm for source code transformation into vectors based on higher-order Markov chains, and include AST-based Markov chains of order 1 and 2 into the comparative study.

We aim to find answers to the following research questions (RQs):

- RQ1: How does the size of the dataset and the selected approach to source code transformation into vector-based representation affect the quality of different classification algorithms in the task detection problem?
- RQ2: Which of the considered approaches to source code transformation into vector-based representation is least sensitive to the used classification algorithm in the task detection problem?

After obtaining vector-based representations of programs, we train and evaluate widely applied classification algorithms such as the k-nearest neighbor (KNN) algorithm [29,30], support vector machine (SVM) [24], random forest (RF) [31], and multilayer perceptron (MLP) [22], using five-fold cross-validation to verify classifier quality. We assess classifier quality using multiclass classification quality metrics such as accuracy, recall, precision, and F1-score [32].

The results of the conducted research indicate that AST-based and opcode-based vectors outperform token-based vectors on small-sized and medium-sized source code datasets. The use of histograms of assembly language instruction opcodes, AST-based histograms, and Markov chains leads to high classifier quality, despite the simplicity of these approaches to transforming source code into vectors. Representations based on first-order Markov chain models are least sensitive to the used classification algorithm, and an RF-based classifier used with Markov chains leads to the best classification quality. The obtained results are verified using null hypothesis significance testing.

The rest of the paper is structured as follows. Section 2 briefly reviews recent research in the field of program analysis in the context of the considered task classification problem, Section 3 briefly describes the task detection problem. Section 4 lists the approaches to source code transformation into vector-based representations studied in the current research. Section 5 describes the conducted numerical experiments and provides answers to RQ1 and RQ2. Finally, Section 6 presents the discussion regarding the results of the experimental studies and highlights future research areas.

2. Related Work

The algorithm detection problem considered in [18] is similar to the task detection problem studied in the current research. The key difference between the two problems is that different algorithms can be used to solve exactly the same task. Research in algorithm detection originates from [33], where the authors proposed the use of code metrics to characterize an implemented algorithm. These metrics included cyclomatic complexity, the number of variables and assignments used in the code, the number of unique operators and operands, and other metrics. The authors of [33] manually constructed a decision tree that analyzed the obtained values of code metrics in order to decide which algorithm was implemented in a given code snippet.

Modern static analyzers are often based on data mining and machine learning methods such as clustering algorithms [11,19] and classification algorithms [7,9,15,16,18] that automatically discover and learn patterns hidden in data. In [11], a clustering algorithm was used to infer syntactic program transformations from examples, with the aims of providing assistance with refactoring tasks and suggesting repairs to programming assignments in massive open online courses (MOOCs). In [19], a clustering algorithm was applied to the solutions of unique programming exercises with the aim of discovering the main algorithmic concepts used by students of a MOOC.

In [9,34], the method name prediction problem was considered, where the problem was reduced to the algorithm classification problem. Similar to [33], the approach described in [34] was based on code metrics, but used a KNN model, determining the k methods that were most similar to the given method with a known name. In [9], the methods were preliminary transformed into AST path contexts that were then fed into a neural network in order to learn program vectors; the authors of [9] also attempted to solve the method name prediction problem. In [7], a binary classifier was developed that allowed the automatic determination of whether the analyzed source code contains bugs. The developed defect prediction tool also transforms programs into vectors based on paths in their ASTs in a fashion similar to [9], and the considered defect prediction problem is similar to the algorithm detection problem. In [35], an AST-based CNN model was proposed for classifying programs by the implemented functionalities. In [18], the algorithm classification problem was extensively studied, and state-of-the-art classifiers and evaders were compared. The comparison included different approaches to source code transformation into vector-based representations, as well as approaches to code obfuscation and optimization.

Before applying a data mining method to either a clustering problem or a classification problem, a dataset containing sample source codes of programs has to be prepared, and relevant features have to be extracted from the dataset. As shown in [7,9,18,19], the transformation of programs into vector-based representations, which is also known in literature as *program embedding* or *source code embedding*, is a useful feature extraction technique that expands the variety of machine learning algorithms that can be applied to source code analysis [7,9,18,21,25].

Recent research introduced a number of different approaches to source code embedding. In [21], a token-based word2vec model and a multilayer perceptron were applied to the malware classification problem. The authors applied word2vec to bytes of the binary file and to instructions in the assembly file of a malware sample; the obtained vectors were then passed to a deep neural network, and the network was trained to classify malicious software. In [25], a number of natural language processing techniques such as word2vec, fastText, and BERT were applied to token-based representations of source code for vulnerability prediction, and a long short-term memory (LSTM) network was trained to detect different vulnerabilities by analyzing vector representations of programs. The vulnerability prediction problem reduces to a classification problem, and the vulnerabilities included structured query language (SQL) injections, cross-site scripting (XSS), and remote code execution. Embeddings based on word2vec outperformed those based on fastText and BERT in the SQL injection detection problem.

However, token-based feature extraction techniques do not take the hierarchical nature of program syntax into account, which can lead to the loss of useful information while transforming source code into vector-based representations, especially in problems similar to algorithm classification [18]. This is crucial, especially in cases when analyzed code snippets are obfuscated at the token level, in identifier name prediction [9], bug detection [7], code clone detection [15,16], and vulnerability detection [14] problems that can be reduced to the task classification problem.

In order to overcome the limitations of word2vec, the code2vec model [9] was proposed; this model is based on AST analysis. The code2vec model first transforms source codes into sets of paths between leaf nodes in AST, called path contexts, and then applies a neural network model with an attention mechanism to learn program vectors based on their path contexts. Since the first mention of the basic code2vec model [9] developed to solve the method name prediction problem for Java and C#, code2vec has been successfully applied to problems other than method name prediction [7,26,27]. In [26], code2vec was applied to the binary and multiclass classification of programs written in Scratch, a block-based programming environment. In [7], PathPair2Vec, an approach similar to code2vec, was applied to software defect prediction. Aiming to simplify the extraction of path contexts from ASTs and to provide code2vec support for programming languages other than C# or Java, which were the only languages supported by code2vec [9], JetBrains™ developed a tool called PathMiner [27], supporting Python, JavaScript, PHP, C, and C++ programming languages.

However, as shown in [18,28], simple histograms of assembly language instruction opcodes are vector representations that are well suited for obfuscated programs and have the potential to outperform complex neural-network-based models in source code classification problems, as well as in problems that reduce to source code classification. Based on the findings reported in [18,28], we include histogram-based approaches to program embedding in the comparative study.

Another simple yet efficient approach to program embedding into vector space is based on Markov chains constructed from ASTs [15,16]. The recently proposed algorithm described in [15] builds an AST for a given program and then converts it into a Markov chain state transition graph, where vertices represent types of AST nodes, edges represent transitions between AST nodes, and weights of edges denote the probabilities of such transitions for the particular AST. Then, the adjacency matrix describing the Markov chain graph is transformed into a vector by concatenating all rows of the weighted adjacency matrix. The use of this approach obtained accuracy of up to 99% in source code classification tasks in [15]. In [19], this approach was used in source code clustering for identification of programs sharing similar high-level syntactic concepts. The code clone detector Amain described in [16] also regards the nodes of an AST as different states and builds a Markov chain model. The Markov chain weighted adjacency matrices are built for pairs of programs and compared using different distance measures, such as Euclidean distance, Manhattan distance, Chebyshev distance, and cosine distance. The feature vector containing calculated distances is then passed to a binary classifier trained to predict whether two programs are semantically similar. According to [16], the developed classifier achieved the highest accuracy on Google Code Jam and Big Clone Bench datasets when compared to state-of-the-art classifiers.

Research in algorithm detection introduced a number of data flow analysis-based approaches to algorithm detection. Such approaches include, for example, [36], where the authors convert code into the static single assignment (SSA) form and construct a system of recurrence equations (SRE) that is then used for equivalence testing of two code snippets. In our current study, we consider only lightweight token-based and AST-based program embeddings due to their simplicity. Despite the high quality of code representations that are based on Markov chains [15,16], the question remains open about whether increasing the order of a Markov chain will lead to the increase in classification quality. In this paper we describe the general algorithm to source code embedding based on Markov chains of

higher order, and compare simple first-order Markov chains with higher-order Markov chains, as well as with histogram-based and neural network-based vectors.

3. Task Detection Problem

A supervised learning problem is represented by a set of objects $\mathbb{X}$, a set of possible answers Y, and an unknown mapping $f : \mathbb{X} \to Y$. The values of the mapping f are known only for objects from a finite set $X \subset \mathbb{X}$, assuming $X = \{x_1, x_2, \ldots, x_s\}$, where s denotes object counts in X. In the supervised setting, we are given a training dataset U containing s examples, $U = \{(x_i, y_i) : x_i \in X,\ y_i \in Y, y_i = f(x_i)\}$ [37]. The task of a supervised learning algorithm is the construction of a mapping $a : \mathbb{X} \to Y$ based on the U set, the a mapping should approximate f not only for objects from X, for which the values of $f(x_i)$ are already known, but for all possible objects from $\mathbb{X}$.

If the set of all possible answers Y is given by $\{1, 2, \ldots, c\}$, assuming $c \in \mathbb{N}$, then the supervised learning problem is a classification into c non-overlapping classes problem. In this case, the learned mapping $a : \mathbb{X} \to Y$ that approximates f is called a *classifier*. In the classification problem, the $\mathbb{X}$ set is considered to be divided into c subsets $\mathbb{X}_1, \mathbb{X}_2, \ldots, \mathbb{X}_c$, and every subset $\mathbb{X}_j$ contains objects belonging to the j-th class, $j = (1, \ldots, c)$; in other words, $\mathbb{X}_j = \{x \in X : f(x) = j\}$. More formally, $\mathbb{X} = \bigcup_{j=1}^{c} \mathbb{X}_j$, c is the class count. If $c = 2$, the classification problem is called *binary*, and if $c > 2$, the problem is multiclass.

In the current study, the finite set of objects X is given by source codes of programs solving unique programming exercises of different types, and the finite set of answers Y contains all known task types. The considered dataset [19] contains solutions to tasks of 11 different types. Hence, the task detection problem is a multiclass classification problem, and the solution to such a problem is the construction of a classifier $a : \mathbb{X} \to Y$ that can determine the type of task from the Y set that a program solves. The training dataset is given by $\{(x_i, y_i) : x_i \in X,\ y_i \in Y\}$, where $x_i \in X$ denotes the i-th program source code, and $y_i \in Y$ denotes the type of task the i-th program solves. The classification algorithms considered in this study take vectors as their input, so in Section 4 we describe the approaches to the conversion of source codes into vector-based representations.

The dataset [19] contains 13,881 programs solving tasks of 11 types. The count of programs grouped by task type is shown in Figure 1, which illustrates the balance of classes in the considered dataset. Every unique exercise of a given type generated by DTA can be solved by a student using any algorithm they prefer. Table 1 lists the brief descriptions of task types.

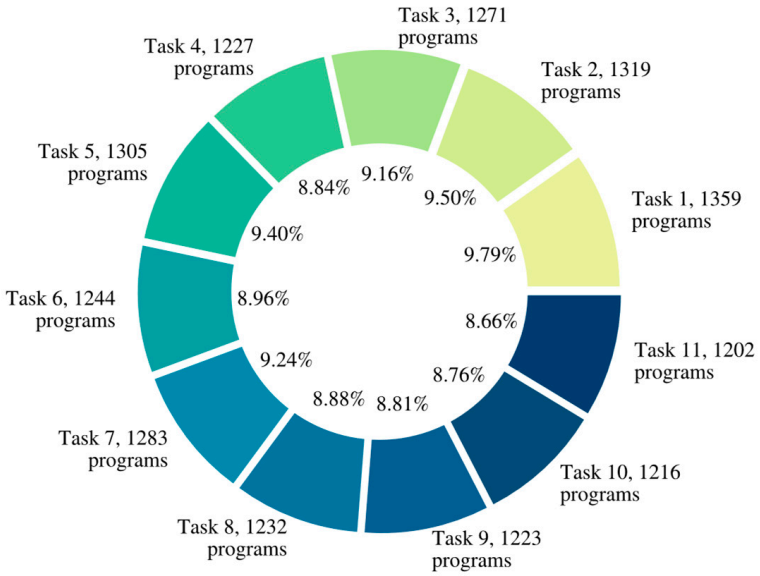


Figure 1. The count of programs grouped by task type.

Table 1. Descriptions of task type solutions which are provided in the [19] dataset.

Task	Description	Category
1.	Implement a mathematical function	Notation into code translation
2.	Implement a piecewise function formula	Notation into code translation
3.	Implement an iterative formula	Notation into code translation
4.	Implement a recurrent formula	Notation into code translation
5.	Implement a function that processes vectors	Notation into code translation
6.	Implement a function computing a decision tree	Notation into code translation
7.	Implement a bit field converter	Conversion between data formats
8.	Implement a text format parser	Conversion between data formats
9.	Implement a finite state machine as a class	Notation into code translation
10.	Implement tabular data transformation	Conversion between data formats
11.	Implement a binary format parser	Conversion between data formats

Examples of programs from the dataset in [19] that solve tasks of different types (see Table 1) using different algorithms are listed in Table 2. The table highlights the main difference between the task classification problem and the algorithm classification problem considered in [18,28]. According to Table 2, completely different algorithms can implement the solution to the same task.

Table 2. Examples of programs solving unique tasks of different types using different approaches.

Task	Approach 1	Approach 2
4.	**def main(n):** ** res1 = −0.31** ** res2 = −0.44** ** for i in range(1, n):** ** temp = res1 − res2 ** 3/38** ** res1 = res2** ** res2 = temp** ** return temp**	**def main(n):** ** if n == 0:** ** return 0.21** ** if n >= 1:** ** ans = main(n − 1) + 1** ** ans += math.atan(main(n − 1)) ** 3** ** return ans**
11.	**...** **def read_a(reader: BinaryReader):** ** a1 = reader.read_uint16()** ** a2 = read_array(** ** source=reader.source,** ** size=reader.read_uint32(),** ** address=reader.read_uint32(),** ** read=lambda reader: read_b(reader),** ** structure_size=50)** ** a3 = read_d(reader)** ** a4 = [reader.read_int32(), reader.read_int32()]** ** return dict(A1=a1, A2=a2, A3=a3, A4=a4)** **...**	**...** **def parse_c(buf, offs):** ** c1, offs = parse(buf, offs, 'float')** ** c2, offs = parse(buf, offs, 'float')** ** c3, offs = parse(buf, offs, 'uint16')** ** c4, offs = parse(buf, offs, 'uint8')** ** c5size, offs = parse(buf, offs, 'uint32')** ** c5offs, offs = parse(buf, offs, 'uint16')** ** c5 = []** ** for _ in range(c5size):** ** val, c5offs = parse(buf, c5offs, 'uint8')** ** c5.append(val)** ** return dict(C1=c1, C2=c2, C3=c3, C4=c4, C5=c5)** **...**

As shown in Table 2, the first approach to implementing a recurrent function (type 4 in Table 1) uses a loop, and the second approach uses a Python function that calls itself, implementing the recursion manually. The first approach to implementing a binary format parser (type 11 in Table 1) uses a class with a hidden state, and the second approach uses a Python function that returns a parsed value and an offset [19].

4. Approaches to Source Code Transformation into Vector-Based Representations

The transformation of source codes into vector-based representations is performed with the aim of extending the variety of classification algorithms that can be applied to solve a source code classification problem; algorithms such as SVM [24] or MLP [22] take vectors as input. The problem of source code embedding into m-dimensional vector space

reduces to the development of a mapping $g : \mathbb{X} \to \mathbb{R}^m$, where $\mathbb{X}$ denotes a set of source codes. Every k-th component of an i-th vector $\vec{v}_i \in \mathbb{R}^m$ associated with the i-th program $x_i \in \mathbb{X}$ encodes the k-th feature of a source code. The obtained vector $\vec{v}_i$ is passed to a classifier $a : \mathbb{R}^m \to Y$ which approximates the unknown target dependency, and Y denotes the set of all possible task types that a program possibly solves.

4.1. word2vec

In the word2vec-based approach to program embedding into vector space, the source codes of programs from the dataset [19] (see Figure 1 and Table 2) are first transformed into sequences of Python programming language tokens using the **tokenize** module [38] that is available in the Python standard library.

In the next step, the continuous bag of words (CBOW) model [39] is trained to embed token sequences into vector space. We use the CBOW implementation available in the Gensim package [40]. Figure 2 summarizes the process of word2vec-based program embedding.

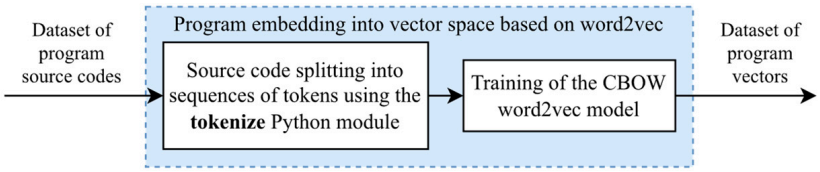


Figure 2. Program embedding into vector space based on the CBOW word2vec model.

4.2. code2vec

The code2vec-based approach to Python code transformation into vector-based representations involves the use of two components, namely, the PathMiner library [27] and the code2vec neural network with an attention mechanism [9]. Figure 3 summarizes the process of code2vec-based embedding of programs into vector space.

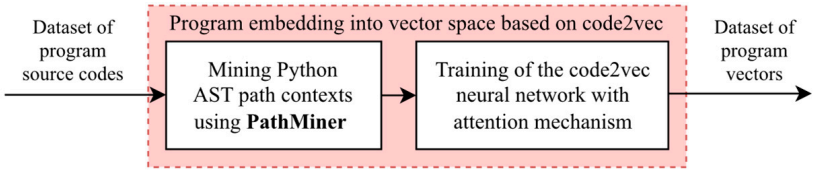
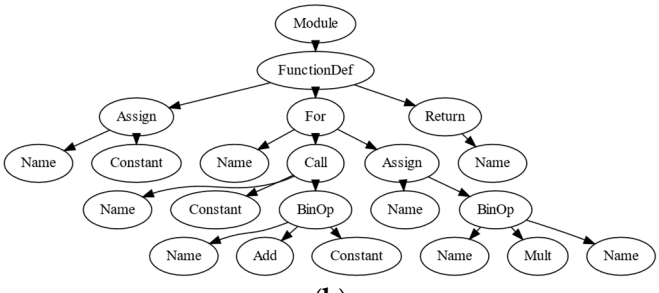


Figure 3. Program embedding into vector space based on code2vec and PathMiner.

As shown in Figure 3, the PathMiner library implements the transformation of source codes written in the Python programming language into sequences of path contexts extracted from an AST, as described in [9]. The object model of a sample AST constructed using the **ast** Python module [41] is shown in Figure 4a. Figure 4b shows the graphical representation of the sample AST (see Figure 4a) obtained with graphviz [42].

```
Module(body=[
  FunctionDef(name='main', args=arguments(args=[arg(arg='n')]), body=[
    Assign(targets=[Name(id='fl')], value=Constant(value=1)),
    For(target=Name(id='i'),
        iter=Call(func=Name(id='range'), args=[
          Constant(value=1),
          BinOp(left=Name(id='n'),
                op=Add(),
                right=Constant(value=1))]),
        body=[Assign(targets=[Name(id='fl')],
                     value=BinOp(left=Name(id='fl'),
                                 op=Mult(),
                                 right=Name(id='i')))]),
    Return(value=Name(id='fl'))])])
```

(**a**)



(**b**)

Figure 4. (**a**) The object model of a sample AST constructed using the **ast** module [41]. (**b**) The graphical representation of a sample AST obtained with the graphviz library [42].

In code2vec [9], a path context is a triplet (t_1, p, t_2), where t_1 and t_2 are operand tokens, leaf nodes of an AST (see Figure 4b), and p is a path in the AST connecting t_1 and t_2. Possible path contexts extracted from the AST shown in Figure 4a,b might be:

$$\text{(f, Name} \uparrow \text{Assign} \downarrow \text{Constant, 1)}, \tag{1}$$

$$\text{(f, Name} \uparrow \text{Return} \uparrow \text{FunctionDef} \downarrow \text{Assign} \downarrow \text{Name, f)}, \tag{2}$$

$$\text{(range, Name} \uparrow \text{Call} \uparrow \text{For} \uparrow \text{FunctionDef} \downarrow \text{Return} \downarrow \text{Name, f)}. \tag{3}$$

The original code2vec implementation [9] supports path context mining only for C# and Java programming languages, and PathMiner [27] provides support for Python path context mining using Another Tool for Language Recognition (ANTLR) [43] grammars. Paths are randomly sampled from an AST; maximum path length and width are hyper-parameters of PathMiner. The extracted tokens and paths are fed to the code2vec neural network which learns vector representations of source codes.

4.3. Markov Chains for Abstract Syntax Trees

AST-based Markov chain models are described in [15,16,19]. In [15,19], only first-order Markov chains are considered and used in classification and clustering tasks. In this study we describe a general algorithm which transforms an AST into a higher-order Markov chain. Our aim is to investigate if there is an increase in classifier quality when program vectors are based on Markov chains of an increased order.

A discrete-time first-order Markov chain is a sequence of dependent discrete random variables $x_0, x_1, x_2, \ldots, x_t$ if the probability distribution of x_{t+1} only depends on the value of the current variable x_t:

$$\mathrm{P}(x_{t+1}|x_t, x_{t-1}, x_{t-2}, \ldots, x_0) = \mathrm{P}(x_{t+1}|x_t), \tag{4}$$

where x_t and x_{t+1} are random variables following each other, t is the time moment.

A discrete-time n-th order Markov chain is a sequence of dependent discrete random variables $x_0, x_1, x_2, \ldots, x_t$ if the probability distribution of x_{t+1} depends on the values of the n preceding variables $x_t, x_{t-1}, x_{t-2}, \ldots, x_{t+1-n}$:

$$\mathrm{P}(x_{t+1}|x_t, x_{t-1}, x_{t-2}, \ldots, x_0) = \mathrm{P}(x_{t+1}|x_t, x_{t-1}, x_{t-2}, \ldots, x_{t+1-n}), \tag{5}$$

where x_t and x_{t+1} are random variables following each other, t is the time moment, and n is the order of the Markov chain.

The range of values of random variables $x_0, x_1, x_2, \ldots, x_t$ is called the state space of a Markov chain. The state space H is given by $\{h_1, h_2, \ldots, h_m\}$, where m is the count of all possible different states of a Markov chain, $m = |H|$. The transitions between every pair of states from H are given by the state transition matrix $\mathbf{P} \in \mathbb{R}^{m \times m}$. For a first-order Markov chain, the elements of a matrix $\mathbf{P}$ are given by:

$$p_{ij} = \mathrm{P}(x_{t+1} = h_j | x_t = h_i),\ p_{ij} \geq 0, \forall\, i : \sum_{j=1}^{m} p_{ij} = 1, \tag{6}$$

where x_t and x_{t+1} are random variables following each other, t is the time moment, $h_i \in H$ and $h_j \in H$, p_{ij} is the cell of the state transition matrix $\mathbf{P}$, denoting the probability of the transition from state h_i to state h_j, m is the count of all possible states, and the matrix $\mathbf{P}$ is the weighted adjacency matrix of the state transition graph of a Markov chain.

When building an AST-based Markov chain, the state space H contains types of all nodes that occur in the AST, and the $\mathbf{P}$ matrix contains the probabilities of the existence of links between every couple of AST node types.

In order to construct an n-th order Markov chain from an AST, we introduce Algorithm 1, which converts neighboring vertices in an AST into ordered sets of vertices; the count of vertices in such sets is equal to the order of a Markov chain. The result of the application of Algorithm 1 to the AST shown in Figure 4a,b is a new AST-based tree, where neighboring vertices are replaced with pairs of vertices (see Figure 5). The "▷" character denotes the comment symbol in Algorithm 1 and beyond.

Algorithm 1: Neighboring vertices merging in an AST

Input: $A = (V, E)$ ▷ AST with vertices represented as tuples.
1. $P = \varnothing$, $R = \varnothing$ ▷ a set of vertices and a set of edges of a new tree.
2. **For each** edge $(v_s, v_m) \in E$ **do:**
3. **For each** edge $(v'_m, v_d) \in E$ **do:**
4. $p_s = v_s \cup (v_{m,i})$, where $i = |v_m|$; $v_{m,i}$ is the last element of the v_m tuple.
5. $p_d = v_m \cup (v_{d,i})$, where $i = |v_d|$; $v_{d,i}$ is the last element of the v_d tuple.
6. $P \leftarrow P \cup \{p_s, p_d\}$.
7. $R \leftarrow R \cup \{(p_s, p_d)\}$.
8. **End loop.**
9. **End loop.**
10. **Return** $A^* = (P, R)$ ▷ AST-based tree with merged vertices.

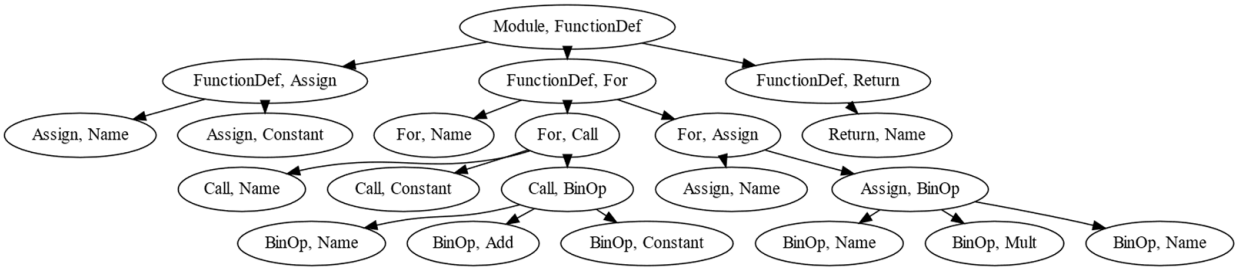


Figure 5. The result of the application of Algorithm 1 to the AST shown in Figure 4b.

The construction of an n-th order Markov chain from an AST is performed according to Algorithm 2. First-order and second-order Markov chains constructed from the sample AST shown in Figure 4a,b according to Algorithm 2 are shown in Figure 6.

Algorithm 2: Construction of an n-th order Markov chain from an AST

Input: s ▷ program source code,
$n \in \mathbb{N} \wedge n \geq 1$ ▷ Markov chain order.
1. **Construct** an AST $A = (V, E)$ for program s using the **ast.parse** function [41].
2. **Delete** from A vertices belonging to set {**Load**, **Store**, **alias**, **arguments**, **arg**}.
3. **Repeat** n times:
4. **Set** A to a new tree computed according to Algorithm 1.
5. **End loop.**
6. **Define** the mapping $g : V \rightarrow T$ that maps a vertex $v \in V$ to its type.
7. $M = \varnothing$ ▷ a set of edges of an n-th order Markov chain.
8. $T = \{g(v) : v \in V\}$ ▷ a set of vertices filled with AST node types from V.
9. **For each** vertex type $t \in T$ **do:**
10. $V_d = \{v_d : (v_s, v_d) \in E \wedge g(v_s) = t\}$ ▷ multiset of descendant vertices of type t.
11. $T_d = \{g(v_d) : v_d \in V_d\}$ ▷ types of descendants of vertices of type t.
12. **For each** descendant vertex type $t_d \in T_d$ **do:**
13. $\omega = \frac{1}{|V_d|}|\{v_d : v_d \in V_d \wedge g(v_d) = t\}|$ ▷ normed descendant count for t_d.
14. $M \leftarrow M \cup \{(t, t_d, \omega)\}$ ▷ add a new edge with weight ω.
15. **End loop.**
16. **End loop.**
17. **Return** the state transition graph (T, M) of an n-th order Markov chain.

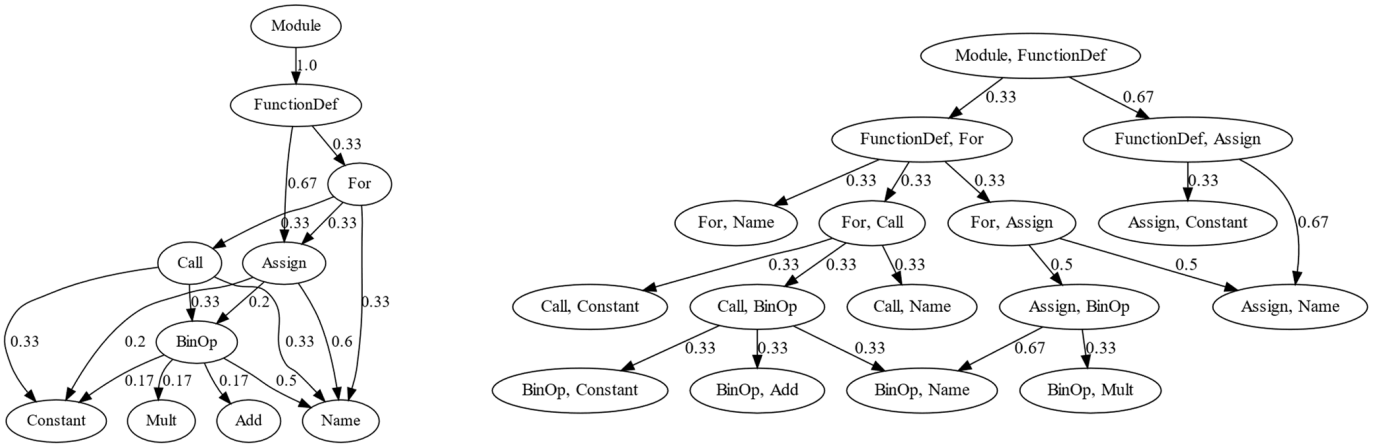


Figure 6. Markov chains of different order constructed from a sample AST shown in Figure 4a,b: (**a**) first-order AST-based Markov chain; (**b**) second-order AST-based Markov chain.

After building an *n*-th order Markov chain state transition graph from an AST using Algorithm 2, a weighted adjacency matrix is constructed based on the state space *H* containing all node types that occur in a vectorized dataset [19]. Figure 7 summarizes the process of Markov-chain-based embedding of programs into vector space.

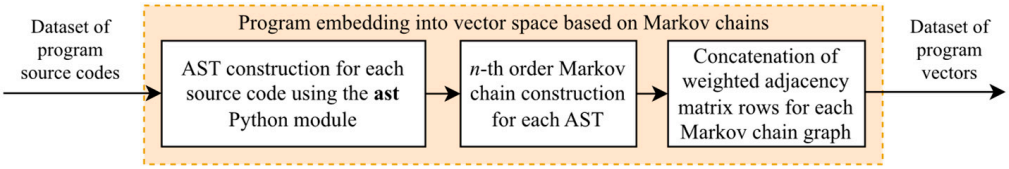


Figure 7. Program embedding into vector space based on *n*-th order Markov chains.

4.4. Histograms of Assembly Language Instruction Opcodes

According to [18,28], histograms of assembly language instruction opcodes tend to outperform complex neural network-based approaches to program transformation into vector-based representations in the algorithm classification task. According to [44,45], opcode frequency histograms also perform well in malware detection problems.

In Python, the bytecode is an implementation detail of the CPython interpreter. The **dis** module [46] available in the standard library allows disassembly of the CPython bytecode. An example of a simple Python function computing factorial is shown in Figure 8a; the corresponding CPython assembly language instructions, their labels, line numbers, and arguments are shown in Figure 8b. The AST describing the program in Figure 8a is shown in Figure 4b.

```
def main(n):
    f = 1
    for id in range(1, n + 1):
        f = f * id
    return f
```

(**a**)

```
1   0 RESUME       0
2   2 LOAD_CONST   1 (1)
    4 STORE_FAST   1 (f)
3   6 LOAD_GLOBAL  1 (NULL + range)
   18 LOAD_CONST   1 (1)
   20 LOAD_FAST    0 (n)
   22 LOAD_CONST   1 (1)
   24 BINARY_OP    0 (+)
   28 PRECALL      2
   32 CALL         2
   42 GET_ITER
>> 44 FOR_ITER      7 (to 60)
   46 STORE_FAST    2 (id)
4  48 LOAD_FAST     1 (f)
   50 LOAD_FAST     2 (id)
   52 BINARY_OP     5 (*)
   56 STORE_FAST    1 (f)
   58 JUMP_BACKWARD 8 (to 44)
5  60 LOAD_FAST     1 (f)
   62 RETURN_VALUE
```

(**b**)

Figure 8. (**a**) A simple Python program; (**b**) the CPython bytecode for the simple Python program.

The process of program embedding into vector space based on opcode histograms is described by Algorithm 3. The algorithm takes in an ordered set of programs and outputs an ordered set of vectors.

Algorithm 3: Source code embedding into vector space based on opcode histograms

Input: S ▷ a set of source codes to be converted into vector representations.
1. $I = \varnothing$ ▷ the ordered set of instruction type multisets.
2. $O = \varnothing$ ▷ the set of known instruction types.
3. **For each** source code $s \in S$ **do:**
4. **Construct** an AST a for program s using the **ast.parse** function [41].
5. **Build** a code object c from the AST a using the **compile** Python function.
6. **Disassemble** a multiset of instruction types i from c using **dis** [46].
7. $I \leftarrow I \cup \{i\}$ ▷ add the multiset of instruction types i to I.
8. $O \leftarrow O \cup i$ ▷ add discovered instruction types to the O set.
9. **End loop.**
10. $V = \varnothing$ ▷ the ordered set of vectors.
11. **For each** multiset of instruction types $i \in I$ **do:**
12. $\vec{v} = \varnothing$.
13. **For each** known instruction type $o \in O$ **do:**
14. $\omega = \frac{1}{|i|}|\{i_k : i_k \in i \wedge i_k = o\}|$ ▷ instruction type occurrence frequency.
15. $\vec{v} \leftarrow \vec{v} \cup \{\omega\}$ ▷ add a new component to the program vector.
16. **End loop.**
17. **End loop.**
18. **Return** the V set of source code vector-based representations.

The histograms of relative occurrence frequencies of CPython assembly language instructions obtained for sample programs listed in the first row of Table 2 are shown in Figure 9; these histograms were constructed using Algorithm 3. The first approach highlighted in magenta in Figure 9 uses a loop for recurrent formula computation (see the first row of Table 2), and the second approach highlighted in cyan uses a function that calls itself recursively.

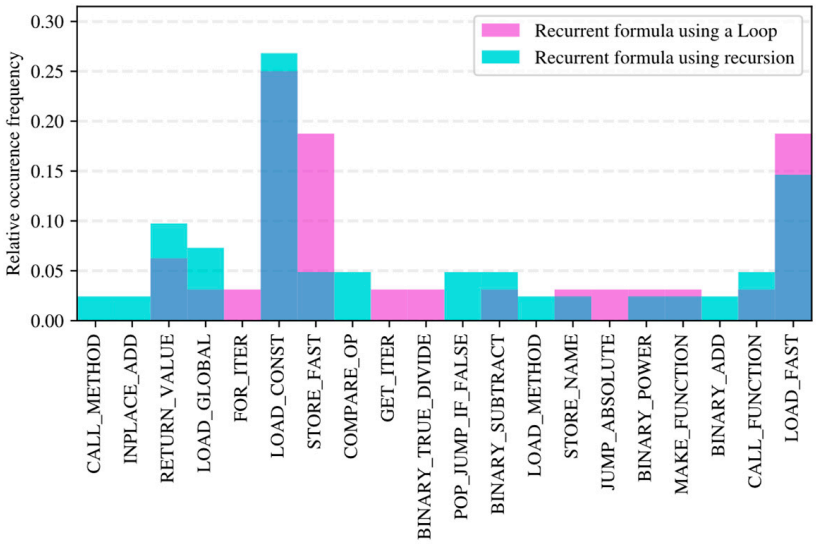


Figure 9. Histograms of CPython assembly language instructions obtained using Algorithm 3 for programs listed in the first row of Table 2. The programs use different algorithms to implement a recurrent formula [19]; the dark blue color highlights overlapping histograms.

According to Figure 9, the CPython assembly language instruction occurrence frequencies differ significantly depending on the algorithm implemented in source code. This is consistent with the results obtained in [18,28]. We observed that histograms for programs solving different tasks differ even more, as this is important in the considered task detection problem. Figure 10 summarizes the process of opcode histogram-based embedding of programs into vector space.

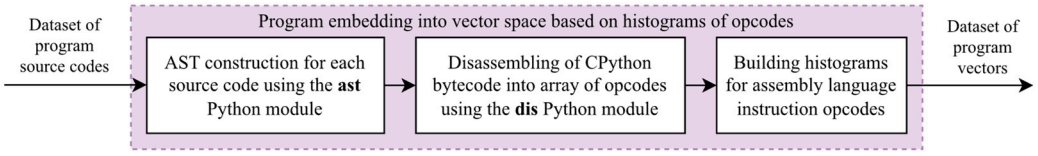


Figure 10. Program embedding into vector space based on opcode histograms.

However, the bytecode histogram-based approach described by Figure 10 and Algorithm 3 has a limitation that narrows its possible applications. The approach depends on the CPython interpreter implementation details [46] and will not work for domain specific languages (DSLs) that are based on the Python programming language syntax. Such DSLs include PyLog [47] and PyMTL3 [48].

4.5. Histograms of Abstract Syntax Tree Node Types

Similar to histograms of assembly language instruction opcodes described in Section 4.4, we include histograms of AST node types in the comparison. The process of program embedding into vector space based on histograms of AST node types is described by Algorithm 4. In contrast to Algorithm 3, Algorithm 4 does not depend on the CPython interpreter implementation details. This makes the AST node type histogram-based approach similar to AST-based Markov chains, but without taking into account transitions among AST vertices.

Algorithm 4: Source code embedding based on AST node type histograms

Input: S ▷ a set of source codes to be converted into vector representations.
1. $N = \varnothing$ ▷ the ordered set of node type multisets associated with source codes.
2. $T = \varnothing$ ▷ the ordered set of known node types.
3. **For each** source code $s \in S$ **do:**
4. **Construct** an AST a for the program s using the **ast.parse** function [41].
5. **Extract** a multiset of node types n that are present in a from a.
6. $N \leftarrow N \cup \{n\}$ ▷ add the multiset of node types n to N.
7. $T \leftarrow T \cup n$ ▷ add discovered node types to the T set.
8. **End loop.**
9. $V = \varnothing$▷ the ordered set of vectors.
10. **For each** multiset of node types $n \in N$ **do:**
11. $\vec{v} = \varnothing$.
12. **For each** known node type $t \in T$ **do:**
13. $\omega = \frac{1}{|n|}|\{n_k : n_k \in n \wedge n_k = t\}|$ ▷ node type occurrence frequency.
14. $\vec{v} \leftarrow \vec{v} \cup \{\omega\}$ ▷ add a new component to the program vector.
15. **End loop.**
16. **End loop.**
17. **Return** the V set of source code vector-based representations.

The histograms of relative occurrence frequencies of AST nodes of different types obtained for sample programs implementing recurrent formulas (see Table 1) and listed in the first row of Table 2 are shown in Figure 11. The histograms shown in Figure 11 were obtained using Algorithm 4. The first approach to recurrent formula implementation highlighted in magenta in Figure 9 uses a loop (see the first row of Table 2), and the second approach highlighted in cyan in Figure 9 uses a function that calls itself.

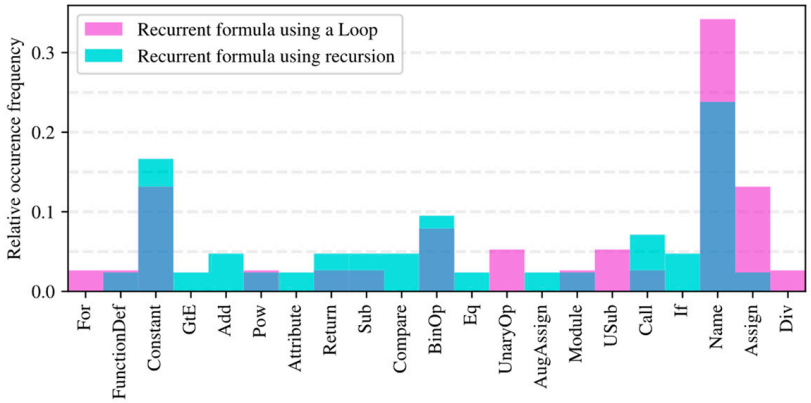


Figure 11. Histograms of AST node types obtained using Algorithm 4 for programs from the first row of Table 2. The programs use different approaches to implementing a recurrent formula [19]; the dark blue color highlights overlapping histograms.

Similar to Figure 9, we observe that the histograms shown in Figure 11 differ significantly depending on the algorithm and task implemented in source code, and histograms for programs solving different tasks differ even more. Figure 12 summarizes the process of AST node type histogram-based embedding of programs into vector space.

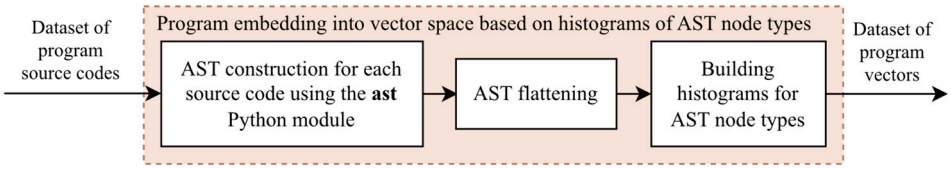


Figure 12. Program embedding into vector space based on AST node type histograms.

5. Experimental Studies

This section provides answers to the research questions discussed in Section 1.

5.1. Experimental Setup

The experiments were conducted on Windows 10 with Python 3.10 installed. The dataset [19] containing Python programs solving tasks of various types, and the task detection problem formulation, are described in Section 3. The results of the conducted experiments do not depend on hardware characteristics as the estimation of the performance of the algorithms is not the subject of the current study.

In the word2vec-based approach to program embedding into vector space (see Section 4.1), we used the CBOW neural network [39] implementation available in the Gensim package [40]. Python programs were preliminary transformed into ordered multisets of tokens using the **tokenize** module [38]. The code2vec-based approach to program embedding (see Section 4.2) involved the use of PathMiner 0.9.0 [27] available as a Docker image tagged **voudy/astminer** on DockerHub. The code2vec neural network implementation was borrowed from [9] and adopted for the use in Windows-based environments.

The Markov-chain-based approach to program transformation into vector-based representations (see Section 4.3) and the histogram-based algorithms (see Sections 4.4 and 4.5) were implemented from scratch in pure Python due to their simplicity, using only modules from the standard library, such as **ast** [41] and **dis** [46].

The used classifier implementations based on KNN [29,30], SVM [24], RF [31], and MLP [22] were taken from the sklearn library [49]. During the conducted experiments, we apply *k*-fold cross-validation during every test run with the aim of verifying classifier quality on different parts of the dataset. Figure 13 describes the *k*-fold cross-validation scheme.

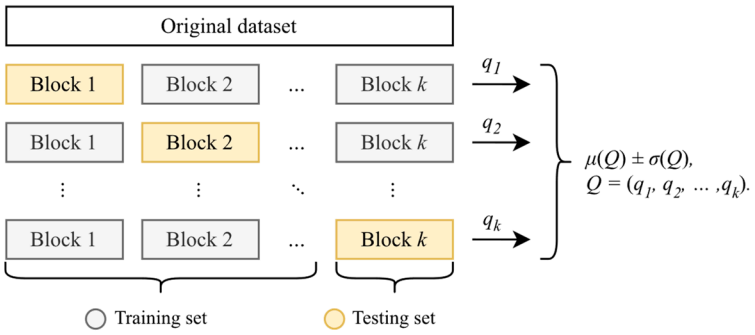


Figure 13. *k*-fold cross-validation for evaluating classifier performance.

According to Figure 13, in *k*-fold cross-validation, the quality of a classifier is measured by splitting the original dataset into k folds k times. $k-1$ blocks are used during classifier training, and 1 block is used during classifier testing using a specialized metric. The blocks that are used during classifier training are highlighted in grey in Figure 13, and the blocks used for classifier testing are highlighted in yellow. Each of the k runs outputs q_i, the i-th value of a specialized quality metric obtained during classifier testing, $i = 1, \ldots, k$. The result of k-fold cross-validation is represented by a tuple $Q = (q_1, q_2, \ldots, q_k)$ containing the obtained specialized metric values. The mean value μ for Q and the standard deviation value σ for Q are given by the following equations:

$$\mu(Q) = \frac{1}{k}\sum_{i=1}^{k} q_i, \tag{7}$$

$$\sigma(Q) = \sqrt{\frac{1}{k}\sum_{i=1}^{k}(q_i - \mu(Q))^2}, \text{ where } Q = (q_1, q_2, \ldots, q_k). \tag{8}$$

In our experimental setup, we assume $k = 5$ for cross-validation. Figure 14 describes the implemented source code classification framework used in our study.

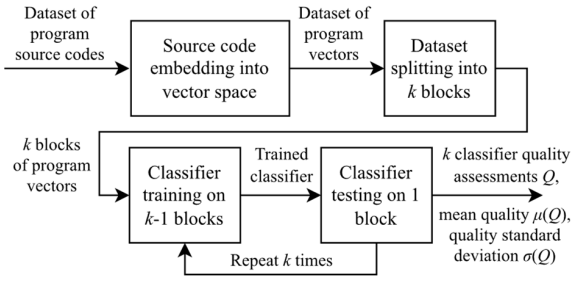


Figure 14. The source code classification framework used in the experiments.

5.2. Influence of Dataset Size and Program Embedding on Classifier Quality

As described in Section 5.1, specialized metrics are used for classification quality assessment on testing parts of the dataset. In a binary classification problem, a set of possible answers $Y = \{-1, +1\}$, where "-1" denotes the negative class, and "+1" denotes the positive class (see Section 3 for a classification problem definition). Binary classifiers can make two types of mistakes by either erroneously assigning an object to class "+1", which is known as *false positive* (*FP*), or erroneously assigning an object to class "-1", which is called *false negative* (*FN*). The correct answer given by a classifier can be either *true positive* (*TP*) or *true negative* (*TN*). For a classifier $a : \mathbb{X} \to Y$,

$$FP = |\{(x_i, y_i) \in U : a(x_i) = +1 \wedge y_i = -1\}|, \tag{9}$$

$$FN = |\{(x_i, y_i) \in U : a(x_i) = -1 \wedge y_i = +1\}|, \tag{10}$$

$$TP = |\{(x_i, y_i) \in U : a(x_i) = +1 \wedge y_i = +1\}|, \tag{11}$$

$$TN = |\{(x_i, y_i) \in U : a(x_i) = -1 \wedge y_i = -1\}|, \tag{12}$$

where U is the dataset (see Section 3) containing s pairs (x_i, y_i), $x_i \in X$ is the i-th object belonging to the set of possible objects that can be passed to a classifier $a : \mathbb{X} \to Y$, and $y_i \in Y$ is the answer associated with the i-th object x_i.

In a multiclass classification problem, the widely used quality assessment metrics include macro-*Accuracy*, macro-*Precision*, macro-*Recall*, and macro-*F1 Score* [15,32]. In these metrics, *FP* (9), *FN* (10), *TP* (11), and *TN* (12) values are computed separately for each class k, treating all other classes as negative, and then combined according to:

$$Accuracy = \frac{1}{c}\sum_{k=1}^{c}\left(\frac{TP_k + TN_k}{TP_k + TN_k + FP_k + FN_k}\right), \tag{13}$$

$$Precision = \frac{1}{c}\sum_{k=1}^{c}\left(\frac{TP_k}{TP_k + FP_k}\right), \tag{14}$$

$$Recall = \frac{1}{c}\sum_{k=1}^{c}\left(\frac{TP_k}{TP_k + FN_k}\right), \tag{15}$$

$$F1\ Score = 2 \cdot \left(\frac{Precision\ \cdot\ Recall}{Precision + Recall}\right), \tag{16}$$

where c denotes the total class count, TP_k, TN_k, FP_k, and FN_k denote (9)–(12) values for the k-th class versus all other classes, the class k is treated as positive, and other classes are treated as negative; these metrics take class imbalance into account [32].

The parameters of the considered classification algorithms are listed in Table 3, alongside the parameters of word2vec Gensim implementation (see Section 4.1), code2vec implementation [9], and PathMiner (see Section 4.2). The approaches to source code embedding into vector space that are based on first-order and second-order Markov chains (see Section 4.3), as well as the histogram-based approaches (see Sections 4.4 and 4.5), do not have any hyperparameters. The parameters that are not listed in Table 3 are initialized with the default values recommended by [9,38,49].

Table 3. Parameters of source code embedding, utility, and classification algorithms.

Algorithm	Parameters
KNN	4 neighbors, distance-based neighbor weighting.
SVM	Radial-basis function kernel, C = 30, one-vs-one strategy.
RF	Max. tree depth is 40, max. tree count is 300, gini criterion.
MLP	100 hidden layer neurons, ReLU activation, Adam optimizer with learning rate = 0.001, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.
word2vec	100 vector components, window size = 5, 5 negative samples, α = 0.025, 5 epochs.
code2vec	384 vector components, dropout keep rate = 0.75, 20 epochs.
PathMiner	ANTLR Python parser, maximum path length = 30, maximum path width 15, maximum path contexts per entity = 300.

Question 1 (RQ1). *How does the size of the dataset and the selected approach to source code transformation into vector-based representation affect the quality of different classification algorithms in the task detection problem?*

Aiming to determine how the size of the dataset and the selected approach to source code embedding into vector space influence the cross-validated (see Figures 13 and 14) quality in the sense of (13)–(16) of the KNN classifier, we obtained the plots shown in Figure 15. The areas filled with different colors in Figure 15 denote the range limited by

$\mu + 3\sigma$ and $\mu - 3\sigma$ according to the three-sigma rule [50], where μ denotes the mean (7) and σ denotes the standard deviation (8) computed based on the observations obtained using 5-fold cross-validation (see Figures 13 and 14).

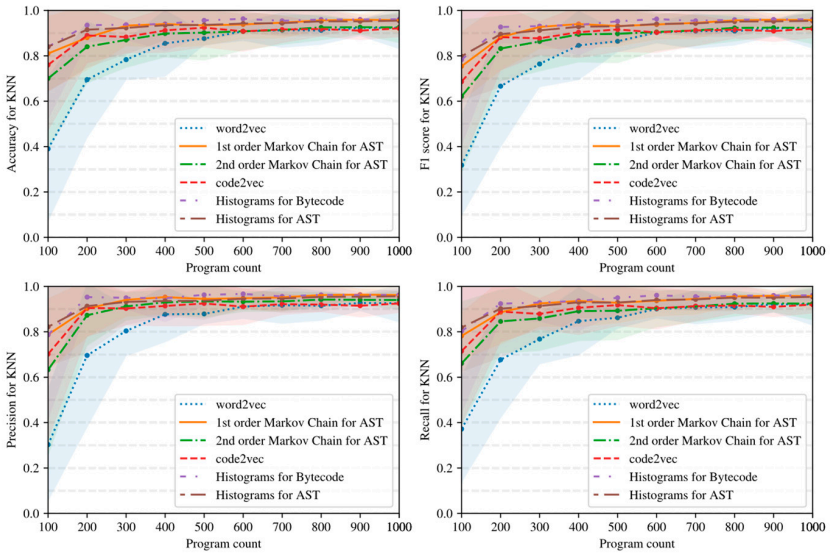


Figure 15. The influence of dataset size and program embedding on the KNN classifier quality.

According to Figure 15, AST-based and bytecode-based program embeddings outperform token-based embeddings on small datasets containing less than 500 programs. The situation changes for medium-sized datasets containing around 1000 programs, where word2vec shows the same results as code2vec and second-order Markov chains. The use of AST-based histograms, first-order Markov chains, and bytecode-based histograms leads to the best KNN classifier quality, although these are the simplest embeddings among the considered ones that do not involve neural network training or hyperparameter tuning. The results are consistent with [28,51], where the authors show that low-resource algorithms have the potential to outperform deep neural networks.

Figure 16 shows the influence of dataset size and program embedding on the quality of classification results obtained with the SVM algorithm.

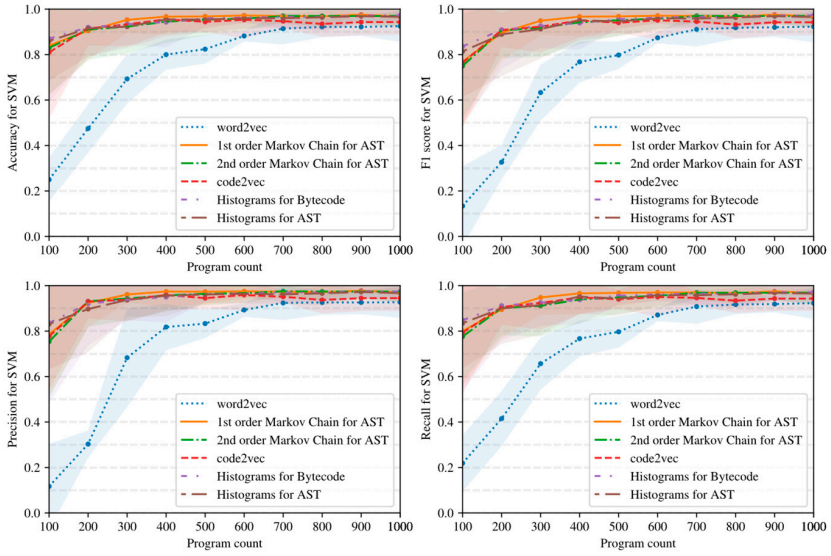


Figure 16. The influence of dataset size and program embedding on the SVM classifier quality.

As shown in Figure 16, AST-based and bytecode-based embeddings again outperform token-based embeddings on small-sized and medium-sized datasets. Source code

vector representations that were obtained using first-order Markov chains and bytecode histograms perform best on small datasets when used with SVM. The increase in the dataset size leads to the increase in token-based embeddings quality.

Figure 17 compares the embeddings used together with an RF-based classifier.

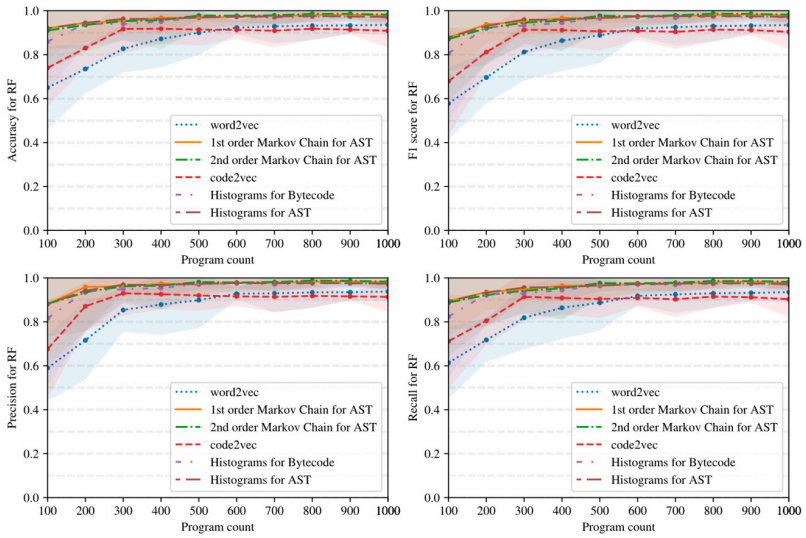


Figure 17. The influence of dataset size and program embedding on the RF classifier quality.

According to Figure 17, first-order and second-order Markov chains, and AST-based and bytecode-based histograms, outperform word2vec and code2vec on small-sized and medium-sized datasets when used with an RF-based classifier in the considered task detection problem. AST-based histograms and Markov chains require fewer source code examples for obtaining high RF classifier quality on small datasets containing around 100 programs when compared to bytecode-based histograms. This may indicate that the neural-network-based continuous vectors are harder for RF to cope with, and AST-based features are of high importance in the task detection problem.

Figure 18 compares the embeddings used together with an MLP-based classifier.

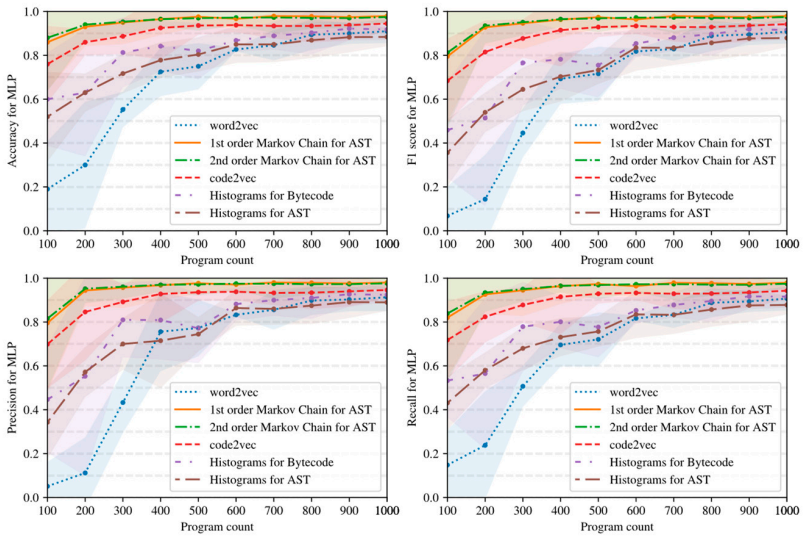


Figure 18. The influence of dataset size and program embedding on the MLP classifier quality.

According to Figure 18, AST-based Markov chains show the best results among the considered approaches to source code transformation into vectors used together with an MLP-based classifier. In contrast to RF (see Figure 17) and KNN (see Figure 15), AST-based and bytecode-based histograms perform significantly worse with MLP, and worse than

token-based embeddings on medium-sized datasets. This may indicate that additional hyperparameter tuning is required when using MLP with histograms.

Aiming to determine which pairs of an embedding and a classifier perform best on a small dataset containing only 100 programs, we obtained the plots shown in Figure 19.

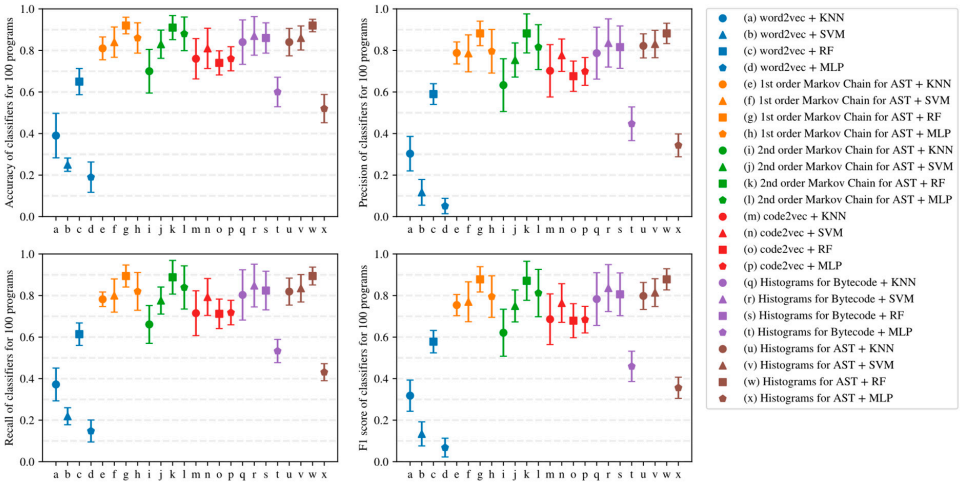


Figure 19. Classification quality for different pairs of a program embedding and a classifier for a small-sized dataset of source codes containing 100 programs from the original [19] dataset.

The line segments shown in Figure 19 denote intervals limited by $\mu + \sigma$ and $\mu - \sigma$, where μ denotes the mean (7) and σ denotes the standard deviation (8) computed based on the observations obtained from 5-fold cross-validation (see Figures 13 and 14). The data visualized in Figure 19 is also presented in Table 4.

Table 4. Classification quality for different pairs of a program embedding and a classifier assessed using different metrics for a small-sized dataset of source codes containing 100 programs. The top 3 best classifier quality scores for each metric are highlighted in bold.

Embedding	Classifier	*Accuracy* (13)	*Precision* (14)	*Recall* (15)	*F1 Score* (16)
word2vec (See Section 4.1)	KNN	39.0 ± 10.7	30.3 ± 8.3	37.2 ± 7.9	31.8 ± 7.5
	SVM	25.0 ± 3.2	11.7 ± 6.2	21.9 ± 4.1	13.4 ± 5.8
	RF	65.0 ± 6.3	59.0 ± 5.0	61.4 ± 5.4	57.8 ± 5.4
	MLP	19.0 ± 7.3	5.1 ± 3.7	14.8 ± 5.3	6.8 ± 4.5
First-order AST-based Markov chain (See Section 4.3)	KNN	81.0 ± 5.5	78.8 ± 5.3	78.2 ± 3.5	75.4 ± 5.1
	SVM	84.0 ± 7.3	78.6 ± 8.9	80.0 ± 8.0	77.0 ± 9.6
	RF	$\mathbf{92.0 \pm 4.0}$	$\mathbf{88.2 \pm 5.9}$	$\mathbf{89.4 \pm 5.3}$	$\mathbf{87.8 \pm 6.1}$
	MLP	86.0 ± 7.3	79.6 ± 10.5	82.0 ± 9.1	79.5 ± 10.0
Second-order AST-based Markov chain (See Section 4.3)	KNN	70.0 ± 10.5	63.3 ± 12.7	66.1 ± 9.1	62.1 ± 11.3
	SVM	83.0 ± 6.8	75.4 ± 8.2	77.6 ± 6.5	75.0 ± 7.7
	RF	$\mathbf{91.0 \pm 5.8}$	$\mathbf{88.2 \pm 9.4}$	$\mathbf{88.8 \pm 8.1}$	$\mathbf{87.1 \pm 9.4}$
	MLP	88.0 ± 8.1	81.6 ± 10.8	83.9 ± 10.4	81.2 ± 11.4
code2vec (See Section 4.2)	KNN	76.0 ± 9.7	70.2 ± 12.6	71.5 ± 10.8	68.6 ± 12.2
	SVM	81.0 ± 9.7	77.7 ± 7.8	79.3 ± 8.9	76.4 ± 9.3
	RF	74.0 ± 5.8	67.6 ± 7.3	71.2 ± 7.1	67.9 ± 8.2
	MLP	76.0 ± 5.8	69.9 ± 6.7	71.8 ± 5.9	68.4 ± 6.4
Histograms for Bytecode (See Section 4.4)	KNN	84.0 ± 10.7	78.7 ± 12.5	80.3 ± 12.1	78.3 ± 12.7
	SVM	87.0 ± 9.3	83.6 ± 11.6	84.8 ± 10.3	83.6 ± 11.3
	RF	86.0 ± 7.3	81.6 ± 10.2	82.4 ± 9.3	80.6 ± 10.3
	MLP	60.0 ± 7.1	44.7 ± 8.1	53.3 ± 5.6	45.9 ± 7.3
Histograms for AST (See Section 4.5)	KNN	84.0 ± 6.6	82.2 ± 5.8	81.9 ± 6.5	79.8 ± 6.5
	SVM	86.0 ± 5.8	83.1 ± 6.6	83.5 ± 6.6	81.4 ± 6.7
	RF	$\mathbf{92.0 \pm 3.0}$	$\mathbf{88.2 \pm 4.9}$	$\mathbf{89.4 \pm 4.3}$	$\mathbf{87.8 \pm 5.1}$
	MLP	52.0 ± 6.8	34.3 ± 5.5	43.1 ± 4.1	35.6 ± 5.1

When using a small dataset of source codes, RF and SVM-based classifiers perform best with embeddings that utilize structural information about programs, such as Markov chains, code2vec, AST node-type histograms, and bytecode histograms, as shown in Figure 19 and in Table 4. The simple KNN classifier shows good results when used together with first-order Markov chains or together with histogram-based embeddings. Moreover, first-order and second-order AST-based Markov chains, as well as histograms of AST node types used together with RF, show the best classification quality compared to other pairs of a program embedding and this classifier. The surprisingly low quality of MLP-based classifiers used together with histogram-based embeddings may indicate that further hyperparameter tuning is required for MLP when used with histograms.

The data shown in Figure 20 and Table 5 compare the classification quality for different pairs of an embedding and a classifier for a dataset containing 1000 samples.

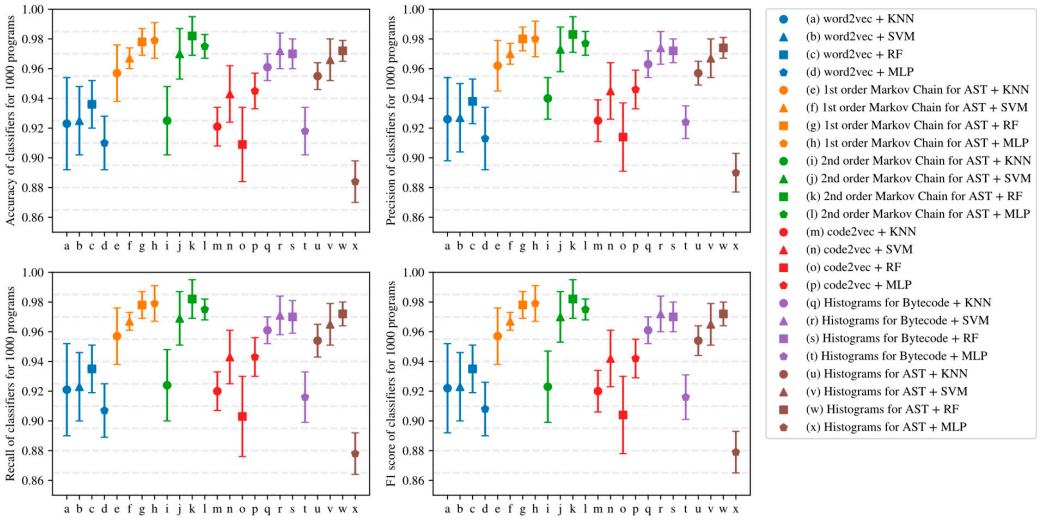


Figure 20. Classification quality for different pairs of a program embedding and a classifier for a small-sized dataset of source codes containing 1000 programs from the original [19] dataset.

The results obtained for 1000 programs shown in Figure 20 and Table 5 are partially similar to the results obtained for 100 programs (see Figure 19). According to Figures 19 and 20, second-order AST-based Markov chains perform significantly worse when used with KNN when compared to first-order Markov chains. Both first-order and second-order Markov chains achieve the best *Accuracy* (13), *Precision* (14), *Recall* (15), and *F1 Score* (16) when used with RF and MLP. As shown in Figures 19 and 20, increasing the order of an AST-based Markov chain gives no noticeable benefits, as first-order and second-order Markov chains show similar performance when used with RF. In the case of KNN, increasing the order of a Markov chain can decrease classifier quality.

According to Figure 20 and Table 5, the token-based embeddings perform significantly better with the increase in the dataset size, leaving behind code2vec-based embeddings used together with RF and histogram-based embeddings used with MLP. Token context in token-based embeddings is represented only by the surrounding tokens, and hence such embeddings are limited by the lack of the structural information of a code snippet. Programs implementing different algorithms have differences in structure that might not be taken into account in token-based embeddings. Additionally, token contexts that appear in programs implementing different algorithms introduce additional noise that might lead to the degradation of the quality of classifiers. The influence of such noise is especially noticeable on small datasets, with a small number of program examples that solve the same task and have many common token contexts.

Table 5. Classification quality for different pairs of a program embedding and a classifier assessed using different metrics for a small-sized dataset of source codes containing 1000 programs. The top 3 best classifier quality scores for each metric are highlighted in bold.

Embedding	Classifier	*Accuracy* (13)	*Precision* (14)	*Recall* (15)	*F1 Score* (16)
word2vec (See Section 4.1)	KNN	92.3 ± 3.1	92.6 ± 2.8	92.1 ± 3.1	92.2 ± 3.0
	SVM	92.5 ± 2.3	92.7 ± 2.3	92.3 ± 2.3	92.3 ± 2.3
	RF	93.6 ± 1.6	93.8 ± 1.5	93.5 ± 1.6	93.5 ± 1.6
	MLP	91.0 ± 1.8	91.3 ± 2.1	90.7 ± 1.8	90.8 ± 1.8
First-order AST-based Markov chain (See Section 4.3)	KNN	95.7 ± 1.9	96.2 ± 1.7	95.7 ± 1.9	95.7 ± 1.9
	SVM	96.7 ± 0.7	97.0 ± 0.7	96.7 ± 0.6	96.7 ± 0.6
	RF	**97.8 ± 0.9**	**98.0 ± 0.8**	**97.8 ± 0.9**	**97.8 ± 0.9**
	MLP	**97.9 ± 1.2**	**98.0 ± 1.2**	**97.9 ± 1.2**	**97.9 ± 1.2**
Second-order AST-based Markov chain (See Section 4.3)	KNN	92.5 ± 2.3	94.0 ± 1.4	92.4 ± 2.4	92.3 ± 2.4
	SVM	97.0 ± 1.7	97.3 ± 1.5	96.9 ± 1.8	97.0 ± 1.7
	RF	**98.2 ± 1.3**	**98.3 ± 1.2**	**98.2 ± 1.3**	**98.2 ± 1.3**
	MLP	97.5 ± 0.8	97.7 ± 0.8	97.5 ± 0.7	97.5 ± 0.7
code2vec (See Section 4.2)	KNN	92.1 ± 1.3	92.5 ± 1.4	92.0 ± 1.3	92.0 ± 1.4
	SVM	94.3 ± 1.9	94.5 ± 1.9	94.3 ± 1.8	94.2 ± 1.9
	RF	90.9 ± 2.5	91.4 ± 2.3	90.3 ± 2.7	90.4 ± 2.6
	MLP	94.5 ± 1.2	94.6 ± 1.3	94.3 ± 1.3	94.2 ± 1.3
Histograms for Bytecode (See Section 4.4)	KNN	96.1 ± 0.9	96.3 ± 0.9	96.1 ± 0.9	96.1 ± 0.9
	SVM	97.2 ± 1.2	97.4 ± 1.1	97.1 ± 1.3	97.2 ± 1.2
	RF	97.0 ± 1.0	97.2 ± 0.8	97.0 ± 1.1	97.0 ± 1.0
	MLP	91.8 ± 1.6	92.4 ± 1.1	91.6 ± 1.7	91.6 ± 1.5
Histograms for AST (See Section 4.5)	KNN	95.5 ± 0.9	95.7 ± 0.8	95.4 ± 1.1	95.4 ± 1.0
	SVM	96.6 ± 1.4	96.7 ± 1.3	96.5 ± 1.4	96.5 ± 1.4
	RF	97.2 ± 0.7	97.4 ± 0.7	97.2 ± 0.8	97.2 ± 0.8
	MLP	88.4 ± 1.4	89.0 ± 1.3	87.8 ± 1.4	87.9 ± 1.4

The embeddings based on first-order Markov chains tend to be least sensitive to the used classifier, showing good results in the sense of (13)–(16) with all of the considered classifiers, according to Figures 19 and 20. Aiming to either confirm or disprove this, we numerically assessed the sensitivity of different approaches to source code transformation into vector-based representations by analyzing the collected data presented in Tables 4 and 5; the results of assessments are presented in Section 5.3.

Aiming to statistically verify the observed superiority of program embeddings that are based on first-order Markov chains, we applied the Wilcoxon signed rank test [52] to the obtained quality assessments of different classifiers. In order to obtain statistically representative results, we repeated the evaluation of each pair of a classifier and an embedding 10 times; each time the datasets containing 100 and 1000 programs were split into 5 blocks, and the classifiers were assessed on 5 different parts of each dataset (see Figure 13). Hence, each of the obtained classifier quality distributions contained 50 values in total. The distribution obtained for first-order Markov-chain-based program embeddings was compared to every other program embedding studied in this research.

According to the null hypothesis H_0, the two compared distributions had no statistically significant differences [52]. The p-value was set to 0.05. The obtained results are presented in Table 6 for a dataset containing 100 programs (see also Figure 19). Table 7 lists the results for a dataset containing 1000 programs (see also Figure 20). The "=" sign in Tables 6 and 7 denotes that no statistically significant differences exist between the compared distributions, the "+" sign denotes that first-order Markov-chain-based embedding outperforms the other embedding, and the "−" sign denotes that first-order Markov-chain-based embedding leads to worse classifier quality.

Table 6. Results of the Wilcoxon signed rank test applied to program embeddings based on first-order Markov chains and all other embeddings used with different classification algorithms trained to solve the task classification problem on a dataset containing 100 programs.

Classifier	KNN		SVM		RF		MLP	
Embedding	**Sign**	***p*-Value**	**Sign**	***p*-Value**	**Sign**	***p*-Value**	**Sign**	***p*-Value**
word2vec	+	0.004	+	0.004	+	0.004	+	0.004
Second-order AST-based Markov chain	+	0.011	=	0.250	=	0.916	=	0.652
code2vec	=	0.820	=	0.359	+	0.004	+	0.008
Histograms for bytecode	−	0.019	=	0.359	=	0.207	+	0.004
Histograms for AST	−	0.004	=	0.164	=	0.498	+	0.004

Table 7. Results of the Wilcoxon signed rank test applied to program embeddings based on first-order Markov chains and all other embeddings used with different classification algorithms trained to solve the task classification problem on a dataset containing 1000 programs.

Classifier	KNN		SVM		RF		MLP	
Embedding	**Sign**	***p*-Value**	**Sign**	***p*-Value**	**Sign**	***p*-Value**	**Sign**	***p*-Value**
word2vec	+	0.004	+	0.004	+	0.004	+	0.004
Second-order AST-based Markov chain	+	0.004	=	0.652	=	0.150	+	0.027
code2vec	+	0.004	+	0.004	+	0.004	+	0.004
Histograms for bytecode	=	0.570	=	0.359	+	0.027	+	0.004
Histograms for AST	=	0.426	=	0.074	+	0.042	+	0.004

According to Tables 6 and 7, first-order Markov-chain-based program embeddings outperform second-order Markov-chain-based embeddings on datasets containing 100 and 1000 programs when used together with KNN and MLP. No statistically significant differences were found between the forementioned embeddings when they were used together with RF and MLP. This may indicate that increasing the order of a Markov chain offers no benefits, and first-order AST-based Markov chains are most suitable for practical applications. First-order Markov chains outperform histograms of AST node types and histograms of assembly language instruction opcodes when the embeddings are used with RF and MLP. Histogram-based embeddings outperform Markov chains on the smallest dataset when used together with the distance-based KNN classifier. As shown in Tables 6 and 7, program embeddings based on Markov chains outperform embeddings based on word2vec and code2vec.

The discovered differences among the considered pairs of a program embedding and a classifier might be caused by the differences in the structure and count of components in the continuous vectors representing programs. Table 8 lists the total component count, non-zero component count, and zero component count in a sample vector for each of the considered embeddings for a dataset containing 100 programs.

Table 8. Total component count and non-zero component count in different embeddings.

Program Embedding	Total Components	Non-Zero	Zero
word2vec	100	100	0
First-order Markov chain	4096	17	4079
Second-order Markov chain	35,721	23	35,698
code2vec	384	384	0
Histograms for bytecode	73	15	58
Histograms for AST	75	19	56

Program vectors representing Markov chains are sparse. They contain the largest number of components among the considered embeddings, as shown in Table 8, but a very small number of sample vector components contain values other than zero. Components of an AST-based Markov chain vector encode transition probabilities between AST node types, and this information appears to be crucial in the considered task detection problem, especially for small-sized datasets (see Figure 19).

MLP achieves high classification quality when used with Markov-chain-based vectors mostly due to high dimensionality and sparsity of the vectors. The size of the input vector in MLP influences the number of connections among input and hidden layer neurons, and the count of connections among neurons increases with the increase in input vector dimensionality. The sparsity of the input vector leads to the activation of only a small number of neurons, and programs solving the same task have many activated neurons in common. This leads to improvements in MLP-based classifier quality.

The low quality of MLP-based classifiers used together with histogram-based embeddings (see Figures 19 and 20) may indicate that the amount of hidden layer neurons of a neural network used for classification is poorly selected and requires fine tuning (see Table 3 for the used MLP parameters for all algorithms), as according to Table 8 histogram vectors contain the least number of components.

5.3. Sensitivity of Program Embeddings to the Used Classification Algorithm

Question 2 (RQ2). *Which of the considered approaches to source code transformation into vector-based representation is least sensitive to the used classification algorithm in the task detection problem?*

A program embedding that does not depend on a specific classification algorithm and works well with different classifiers has the potential to find more possible applications in different tasks and domains. For example, a simple KNN algorithm is used when labeled data are hard or impossible to obtain, or when a non-parametric classification technique is required [51]. RF, SVM, and MLP classifiers are used when high accuracy is required, and the specific classifier is typically chosen depending on the domain and data.

In order to numerically determine which approach to source code embedding into vector space is least sensitive to the used classification algorithm in the considered task detection problem, we use the weighted standard deviation (weighted SD) as the measure of embedding sensitivity to the classification algorithm used:

$$\overline{\sigma}(\mathbb{Q}) = \sqrt{\sum_{j=1}^{n} \left(\mu(Q_j) - \overline{\mu}(\mathbb{Q})\right)^2 \frac{\sigma^{-1}(Q_j)}{\sum_{i=1}^{n} \sigma^{-1}(Q_i)}}, \tag{17}$$

$$\overline{\mu}(\mathbb{Q}) = \sum_{j=1}^{n} \mu(Q_j) \frac{\sigma^{-1}(Q_j)}{\sum_{i=1}^{n} \sigma^{-1}(Q_i)}, \tag{18}$$

where n is the count of different classifier assessments in the $\mathbb{Q}$ set obtained for a given program embedding, $\mathbb{Q} = \{Q_1, \ldots, Q_j, \ldots, Q_n\}$; $j = \overline{1,n}$ is the classifier number; Q_j is the set containing k classifier quality assessments obtained for the j-th classifier using k-fold cross-validation (see Figures 13 and 14); $\mu(Q_j)$ is the mean quality of the j-th classifier computed according to (7) based on k-fold cross-validation output Q_j; and $\sigma(Q_j)$ is the standard deviation of the j-th classifier quality, computed according to (8) based on k-fold cross-validation output Q_j.

The weighted mean classifier quality (18) computed for a given program embedding is the overall quality of the program embedding, and the weighted SD (17) is the sensitivity of the program embedding to the used classification algorithm. In (17) and (18), the classifier quality assessments that have a small standard deviation $\sigma(Q_j)$ have a greater influence on the weighted SD value $\overline{\sigma}(\mathbb{Q})$.

For example, if we assess the sensitivity to the used classification algorithm of the embedding that is based on the first-order Markov chains, on a small dataset containing 100 programs using the *Accuracy* (13) metric, for classifiers such as KNN, SVN, RF, and MLP,

we respectively obtain the means $\mu(Q_1) = 81.0$, $\mu(Q_2) = 84.0$, $\mu(Q_3) = 92.0$, $\mu(Q_4) = 86.0$ (see Table 4), the standard deviations $\sigma(Q_1) = 5.5$, $\sigma(Q_2) = 7.3$, $\sigma(Q_3) = 4.0$, $\sigma(Q_4) = 7.3$, and the weighted SD value $\overline{\sigma}(\mathbb{Q}_1) \approx 4.45$, $\mathbb{Q}_1 = \{Q_1, Q_2, Q_3, Q_4\}$. If we assess the sensitivity to the used classifier of the embedding that is based on second-order Markov chains on the same dataset according to (13) and Table 4, we obtain $\overline{\sigma}(\mathbb{Q}_2) \approx 7.36$. As $\sigma(\mathbb{Q}_1) < \sigma(\mathbb{Q}_2)$, the first-order Markov-chain-based program embedding is less sensitive to the used classifier than the second-order Markov-chain-based embedding.

The comparison of sensitivity to the used classifier of the considered embeddings on a small dataset containing 100 programs using the *Accuracy* (13), *Precision* (14), *Recall* (15), and *F1 Score* (16) metrics computed according to weighted SD (17) is shown in Figure 21. According to Figure 21, code2vec and first-order Markov chains are least sensitive to the used classification algorithm on a dataset containing 100 programs solving 11 different tasks in the task detection problem.

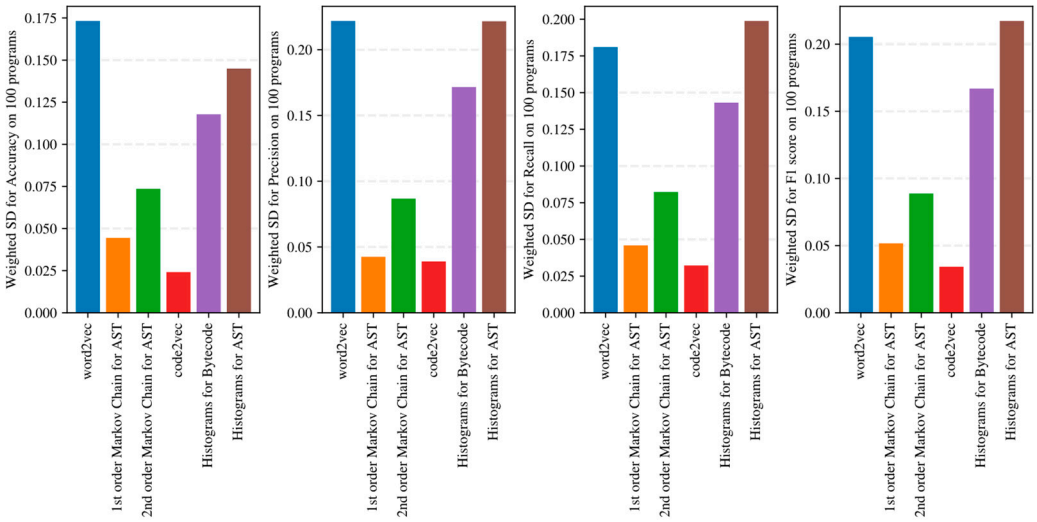


Figure 21. The comparison of sensitivity to the used classifier of the considered program embeddings on a small dataset with 100 programs based on the data from Table 4 (lower is better).

The comparison of sensitivity to the used classifier of the considered embeddings on a dataset containing 1000 programs using different classifier quality metrics computed according to (17) is shown in Figure 22.

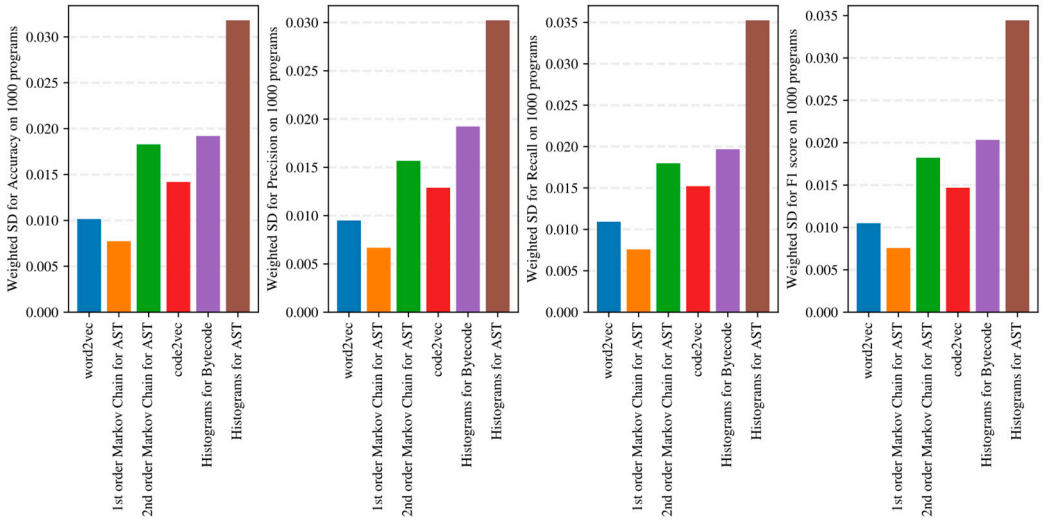


Figure 22. The comparison of sensitivity to the used classifier of the considered program embeddings on a dataset with 1000 programs based on the data from Table 5 (lower is better).

According to Figure 22, first-order Markov chains and word2vec are the embeddings that are least sensitive to the used classification algorithm on a dataset containing 1000 programs solving 11 different tasks in the task detection problem. This matches the observations discussed in Section 5.2, and the quality of the token-based word2vec embedding improves with the increase in the dataset size.

According to both Figures 21 and 22, first-order Markov chain is the embedding that is least sensitive to the used classification algorithm regardless of the dataset size. The increase in Markov chain order increases the sensitivity of the embeddings to the used classifier. According to Figures 19 and 20, the KNN algorithm shows worse performance with the increase in the Markov chain order.

6. Discussion

In the presented research, we considered the task detection problem, which is a multiclass classification problem. Given the dataset [19] containing Python programs solving tasks of different types, the classifiers are trained to find the type of the task that a program solves (see Section 3). The classification algorithms considered in this study are KNN, SVM, RF, and MLP. These algorithms take vectors as their input, so we perform a preliminary transform of the source codes of programs into vector-based representations. We consider approaches to source code embedding into vector space such as the token-based word2vec model [39], the token and AST-based code2vec model [9], AST-based Markov chains [15,16] of order 1 and 2, histograms of assembly language instruction opcodes [18,28], and histograms of AST node types. The approaches are discussed in detail in Section 4.

Aiming to determine how the size of the dataset and the selected approach to source code embedding into vector space influence the cross-validated *Accuracy* (13), *Precision* (14), *Recall* (15), and *F1 Score* (16) of classification algorithms such as KNN, SVM, RF, and MLP (RQ1), we conducted the numerical experiments using the experimental setup and methodology as described in Section 5.1. The results are provided and discussed in Section 5.2. Overall, we reached the following conclusions:

- Simple AST-based or bytecode-based representations of programs such as Markov chains and histograms outperform complex neural network-based embeddings with many hyperparameters in the considered task detection problem, especially when the amount of training data is limited (see Figures 15–17).
- Embeddings of programs that are based on either AST or bytecode outperform token-based embeddings on small-sized datasets; however, with the increase in the dataset size, the difference in the quality of classifiers decreases and has the potential to vanish on larger datasets (see, for example, Figures 15 and 16).
- Source code embeddings that are based on first-order Markov chains built for ASTs used with RF demonstrate the best classification quality in the sense of (13)–(16) (see Tables 4 and 5); the results were verified using the Wilcoxon signed rank statistical test (see Tables 6 and 7).
- Increasing the order of AST-based Markov chains offers no noticeable improvement in classifier quality in the sense of (13)–(16), and can even lead to quality degradation in the case of KNN (see Figures 19 and 20); this makes AST-based Markov chains of order 1 most suitable for practical applications.

Aiming to determine which of the considered approaches to program embedding is least sensitive to the used classification algorithm in the considered task detection problem (RQ2), we numerically assessed the sensitivity using (17) and cross-validated *Accuracy* (13), *Precision* (14), *Recall* (15), and *F1 Score* (16) of KNN, SVM, RF, and MLP. According to the results provided in Section 5.3, code2vec and first-order AST-based Markov chains are the embeddings that are least sensitive to the used classifier on a small dataset containing 100 programs, whereas word2vec and first-order AST-based Markov chains are the embeddings that are least sensitive to the used classifier on a dataset containing 1000 programs. Overall, first-order AST-based Markov chains are program embeddings that are least sensitive to the

used classifier. Higher-order Markov chains show worse results in the sense of (17) when compared to simple first-order AST-based Markov chains.

Algorithms 1 and 2, which are responsible for construction of Markov chains of any order, have limitations, as they only support the Python programming language and construct program ASTs using the Python standard library [41]; this also applies to Algorithm 4. However, the task of adding support for other languages to these algorithms reduces to the replacement of calls to the Python standard library with calls to syntax tree builders specific to every other supported language; the syntax tree builders can be implemented using, for example, ANTLR-based parsers [43]. Another option of adding support for more languages is the incorporation of language-agnostic AST (LAAST) builders [53] into Algorithms 1, 2, and 4. Algorithm 3, in contrast, depends on the CPython interpreter implementation details, and cannot be used for the analysis of code written in Python-based DSLs for specialized processors [47,48]. LAAST builders were not considered in the current research as the dataset used for benchmarking the embeddings contained only Python programs, so future work could cover quality assessments of language-agnostic versions of the discussed algorithms.

7. Conclusions

The results of the experimental studies conducted in the current research show that program vector representations that are based on first-order Markov chains constructed from ASTs are most suitable for practical applications when compared to higher-order Markov chain models. Simple embeddings described by Algorithms 2–4 have the potential to outperform complex neural network-based models and can be easily ported to other programming languages. Future research could cover the applicability of Markov-chain-based embeddings to the discovery of code sharing similar concepts in large software projects, with the aim to eliminate duplicate implementations of the same algorithm from the code base. Another possible application of Markov-chain-based embeddings might include the search of code fragments suitable for hardware acceleration based on the given database of examples using the KNN algorithm. Specialized processors such as the field programmable gate array (FPGA) [54,55] can be configured to implement a domain-specific algorithm in hardware, offering significant performance benefits. As shown in Table 2, different algorithms can be used to solve the same task, and in real world scenarios it is often hard to obtain a large dataset containing problem-specific code solving different tasks. This means that an ideal classifier should be able to detect a task using only one example of every known algorithm that can solve the task. Markov chains are simple models that can be constructed for graphs other than AST (see Algorithm 2), so future work could cover quality assessment of embeddings that are based on Markov chains built for either control flow graphs or data flow graphs for either high-level or low-level programming languages. Regarding educational data mining, future research could focus on anomaly detection [56] in source codes of programs submitted by students [19].

Author Contributions: Conceptualization, methodology L.A.D., P.N.S. and A.V.G.; software, resources, visualization, testing A.V.G.; investigation, formal analysis, L.A.D., P.N.S. and A.V.G.; validation, supervision, project administration L.A.D. and P.N.S.; writing—original draft preparation, A.V.G.; writing—review and editing, L.A.D. and P.N.S.; All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data analyzed in this study are openly available in Zenodo at https://doi.org/10.5281/zenodo.7799971 (accessed on 8 July 2023).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Simon, F.; Steinbruckner, F.; Lewerentz, C. Metrics based refactoring. In Proceedings of the 5th European Conference on Software Maintenance and Reengineering, Lisbon, Portugal, 14–16 March 2001; IEEE: Piscataway, NJ, USA, 2001; pp. 30–38.

2. Campbell, G.A. Cognitive Complexity: An Overview and Evaluation. In Proceedings of the 2018 International Conference on Technical Debt, Gothenburg, Sweden, 27–28 May 2018; Association for Computing Machinery: New York, NY, USA, 2018; pp. 57–58.
3. Chen, Z.; Chen, L.; Ma, W.; Xu, B. Detecting Code Smells in Python Programs. In Proceedings of the 2016 International Conference on Software Analysis, Testing and Evolution (SATE), Harbin, China, 3–4 November 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 18–23.
4. Zhang, Z.; Xing, Z.; Xia, X.; Xu, X.; Zhu, L. Making Python Code Idiomatic by Automatic Refactoring Non-Idiomatic Python Code with Pythonic Idioms. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Singapore, 14–16 November 2022; Association for Computing Machinery: New York, NY, USA, 2022; pp. 696–708.
5. Bruch, M.; Monperrus, M.; Mezini, M. Learning from Examples to Improve Code Completion Systems. In Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Amsterdam, The Netherlands, 24–28 August 2009; Association for Computing Machinery: New York, NY, USA, 2009; pp. 213–222.
6. Li, X.; Wang, L.; Yang, Y.; Chen, Y. Automated Vulnerability Detection in Source Code Using Minimum Intermediate Representation Learning. *Appl. Sci.* **2020**, *10*, 1692. [CrossRef]
7. Shi, K.; Lu, Y.; Chang, J.; Wei, Z. PathPair2Vec: An AST path pair-based code representation method for defect prediction. *J. Comput. Lang.* **2020**, *59*, 100979. [CrossRef]
8. Li, Y.; Wang, S.; Nguyen, T. A Context-based Automated Approach for Method Name Consistency Checking and Suggestion. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering, Madrid, Spain, 22–30 May 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 574–586.
9. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.* **2019**, *3*, 40. [CrossRef]
10. Ziadi, T.; Frias, L.; Da Silva, M. Feature Identification from the Source Code of Product Variants. In Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering, Szeged, Hungary, 27–30 March 2012; IEEE: Piscataway, NJ, USA, 2012; pp. 417–422.
11. Rolim, R.; Soares, G.; D'Antoni, L.; Polozov, O.; Gulwani, S.; Gheyi, R.; Suzuki, R.; Hartmann, B. Learning Syntactic Program Transformations from Examples. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, 20–28 May 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 404–415.
12. Allamanis, M.; Sutton, C. Mining Idioms from Source Code. In Proceedings of the 22nd ACM Sigsoft International Symposium on Foundations of Software Engineering, Hong Kong, China, 16–21 November 2014; Association for Computing Machinery: New York, NY, USA, 2014; pp. 472–483.
13. Iwamoto, K.; Wasaki, K. Malware Classification Based on Extracted API Sequences Using Static Analysis. In Proceedings of the 8th Asian Internet Engineering Conference, Bangkok, Thailand, 14–16 November 2012; Association for Computing Machinery: New York, NY, USA, 2012; pp. 31–38.
14. Russell, R.; Kim, L.; Hamilton, L.; Lazovich, T.; Harer, J.; Ozdemir, O.; Ellingwood, P.; McConley, M. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In Proceedings of the 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), Orlando, FL, USA, 17–20 December 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 757–762.
15. Demidova, L.A.; Gorchakov, A.V. Classification of Program Texts Represented as Markov Chains with Biology-Inspired Algorithms-Enhanced Extreme Learning Machines. *Algorithms* **2022**, *15*, 329. [CrossRef]
16. Wu, Y.; Feng, S.; Zou, D.; Jin, H. Detecting Semantic Code Clones by Building AST-based Markov Chains Model. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, Rochester, MI, USA, 10–14 October 2022; Association for Computing Machinery: New York, NY, USA, 2022; pp. 1–13.
17. Wiem, B.; Marwa, H. Supervised Hardware/Software Partitioning Algorithms for FPGA-based Applications. In Proceedings of the 12th International Conference on Agents and Artificial Intelligence (ICAART 2020), Valletta, Malta, 22–24 February 2020; Springer: Berlin, Germany, 2020; Volume 2, pp. 860–864.
18. Damásio, T.; Canesche, N.; Pacheco, V.; Botacin, M.; da Silva, A.F.; Pereira, F.M.Q. A Game-Based Framework to Compare Program Classifiers and Evaders. In Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization, Montréal, QC, Canada, 25 February–1 March 2023; Association for Computing Machinery: New York, NY, USA, 2023; pp. 108–121.
19. Demidova, L.A.; Andrianova, E.G.; Sovietov, P.N.; Gorchakov, A.V. Dataset of Program Source Codes Solving Unique Programming Exercises Generated by Digital Teaching Assistant. *Data* **2023**, *8*, 109. [CrossRef]
20. Sovietov, P.N.; Gorchakov, A.V. Digital Teaching Assistant for the Python Programming Course. In Proceedings of the 2022 2nd International Conference on Technology Enhanced Learning in Higher Education (TELE), Lipetsk, Russia, 26–27 May 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 272–276.
21. Qiao, Y.; Zhang, W.; Du, X.; Guizani, M. Malware classification based on multilayer perception and Word2Vec for IoT security. *ACM Trans. Internet Technol.* **2021**, *22*, 10. [CrossRef]
22. Rosenblatt, F. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychol. Rev.* **1958**, *65*, 386–408. [CrossRef]

23. Barchi, F.; Parisi, E.; Urgese, G.; Ficarra, E.; Acquaviva, A. Exploration of Convolutional Neural Network models for source code classification. *Eng. Appl. Artif. Intell.* **2021**, *97*, 104075. [CrossRef]
24. Cortes, C.; Vapnik, V. Support-vector networks. *Mach. Learn.* **1995**, *20*, 273–297. [CrossRef]
25. Bagheri, A.; Hegedűs, P. A Comparison of Different Source Code Representation Methods for Vulnerability Prediction in Python. In Proceedings of the Quality of Information and Communications Technology: 14th International Conference, QUATIC 2021, Algarve, Portugal, 8–11 September 2021; Springer: Berlin, Germany, 2021; Volume 14, pp. 267–281.
26. Fein, B.; Graßl, I.; Beck, F.; Fraser, G. An Evaluation of code2vec Embeddings for Scratch. In Proceedings of the 15th International Conference on Educational Data Mining, Durham, UK, 24–27 July 2022; International Educational Data Mining Society: Massachusetts, USA, 2022; pp. 368–375.
27. Kovalenko, V.; Bogomolov, E.; Bryksin, T.; Baccheli, A. PathMiner: A Library for Mining of Path-Based Representations of Code. In Proceedings of the 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), Montreal, QC, Canada, 26–27 May 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 13–17.
28. Da Silva, A.F.; Borin, E.; Pereira, F.M.Q.; Queiroz, N.L., Jr.; Napoli, O.O. Program Representations for Predictive Compilation: State of Affairs in the Early 20's. *J. Comput. Lang.* **2022**, *73*, 101171. [CrossRef]
29. Fix, E.; Hodges, J. Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties. *Int. Stat. Rev.* **1989**, *57*, 238–247. [CrossRef]
30. Altman, N.S. An introduction to kernel and nearest-neighbor nonparametic regression. *Am. Stat.* **1992**, *46*, 175–185.
31. Ho, T.K. Random Decision Forests. In Proceedings of the 3rd International Conference on Document Analysis and Recognition, Montreal, QC, Canada, 14–16 August 1995; IEEE: Piscataway, NJ, USA, 1995; pp. 278–282.
32. Grandini, M.; Bagli, E.; Visani, G. Metrics for Multi-class Classification: An Overview. *arXiv* **2020**, arXiv:2008.05756.
33. Taherkhani, A.; Malmi, L.; Korhonen, A. Algorithm Recognition by Static Analysis and Its Application in Students' Submissions Assessment. In Proceedings of the 8th International Conference on Computing Education Research, Koli, Finland, 13–16 November 2008; Association for Computing Machinery: New York, NY, USA, 2008; pp. 88–91.
34. Parsa, S.; Zakeri-Nasrabadi, M.; Ekhtiarzadeh, M.; Ramezani, M. Method name recommendation based on source code metrics. *J. Comput. Lang.* **2023**, *74*, 10117. [CrossRef]
35. Bui, N.D.Q.; Jiang, L.; Yu, Y. Cross-Language Learning for Program Classification using Bilateral Tree-based Convolutional Neural Networks. *arXiv* **2017**, arXiv:1710.06159.
36. Alias, C.; Barthou, D. Algorithm Recognition based on Demand-Driven Dataflow Analysis. In Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003), Victoria, BC, Canada, 13–16 November 2003; IEEE: Piscataway, NJ, USA, 2003; p. ensl01663748.
37. Pérez-Ortiz, M.; Jiménez-Fernández, S.; Gutiérrez, P.A.; Alexandre, E.; Hervás-Martínez, C.; Salcedo-Sanz, S. A Review of Classification Problems and Algorithms in Renewable Energy Applications. *Energies* **2016**, *9*, 607. [CrossRef]
38. Python Software Foundation. Tokenize—Tokenizer for Python Source. 2023. Available online: https://docs.python.org/3/library/tokenize.html (accessed on 10 July 2023).
39. Mikolov, T.; Chen, K.; Corrado, G.; Dean, J. Efficient estimation of word representations in vector space. *arXiv* **2013**, arXiv:1301.3781.
40. Rehurek, R.; Sojka, P. *Gensim–Python Framework for Vector Space Modelling*; NLP Centre, Faculty of Informatics, Masaryk University: Brno, Czech Republic, 2011; Volume 3.
41. Python Software Foundation. AST—Abstract Syntax Trees. 2023. Available online: https://docs.python.org/3/library/ast.html (accessed on 15 July 2023).
42. Gansner, E.R.; North, S.C. An Open Graph Visualization System and its Applications to Software Engineering. *Softw. Pract. Exp.* **2000**, *30*, 1203–1233. [CrossRef]
43. Parr, T.J.; Quong, R.W. ANTLR: A predicated-LL (k) parser generator. *Softw. Pract. Exp.* **1995**, *25*, 789–810. [CrossRef]
44. Canfora, G.; Mercaldo, F.; Visaggio, C.A. Mobile malware detection using op-code frequency histograms. In Proceedings of the 2015 12th International Joint Conference on e-Business and Telecommunications (ICETE), Colmar, France, 20–22 July 2015; IEEE: Piscataway, NJ, USA, 2015; Volume 4, pp. 27–38.
45. Rad, B.B.; Masrom, M.; Ibrahim, S. Opcodes histogram for classifying metamorphic portable executables malware. In Proceedings of the 2012 International Conference on e-Learning and e-Technologies in Education (ICEEE), Lodz, Poland, 24–26 September 2012; IEEE: Piscataway, NJ, USA, 2012; pp. 209–213.
46. Python Software Foundation. Dis—Disassembler for Python Bytecode. 2023. Available online: https://docs.python.org/3/library/dis.html (accessed on 17 July 2023).
47. Huang, S.; Wu, K.; Jeong, H.; Wang, C.; Chen, D.; Hwu, W.M. PyLog: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow. *IEEE Trans. Comput.* **2021**, *70*, 2015–2028.
48. Jiang, S.; Pan, P.; Ou, Y.; Batten, C. PyMTL3: A Python Framework for Open-Source Hardware Modeling, Generation, Simulation, and Verification. *IEEE Micro* **2020**, *40*, 58–66. [CrossRef]
49. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. Scikit-learn: Machine learning in Python. *J. Mach. Learn. Res.* **2011**, *12*, 2825–2830.
50. Pukelsheim, F. The Three Sigma Rule. *Am. Stat.* **1994**, *48*, 88–91.

51. Jiang, Z.; Yang, M.Y.R.; Tsirlin, M.; Tang, R.; Dai, Y.; Lin, J. "Low-Resource" Text Classification: A Parameter-Free Classification Method with Compressors. In Proceedings of the Findings of the Association for Computational Linguistics: EACL 2023, Dubrovnik, Croatia, 2–6 May 2023; pp. 6810–6828.
52. Demšar, J. Statistical Comparisons of Classifiers over Multiple Data Sets. *J. Mach. Learn. Res.* **2006**, *7*, 1–30.
53. Curtis, J. Student Research Abstract: On Language-Agnostic Abstract-Syntax Trees. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing, Virtual, 25–29 April 2022; pp. 1619–1625.
54. Sovetov, S.I.; Tyurin, S.F. Method for synthesizing a logic element that implements several functions simultaneously. *Russ. Technol. J.* **2023**, *11*, 46–55. [CrossRef]
55. Arato, P.; Juhasz, S.; Mann, Z.A.; Orban, A.; Papp, D. Hardware-software partitioning in embedded system design. In Proceedings of the IEEE International Symposium on Intelligent Signal Processing, Budapest, Hungary, 6 September 2003; IEEE: Piscataway, NJ, USA, 2003; pp. 197–202.
56. Demidova, L.A.; Sovietov, P.N.; Andrianova, E.G.; Demidova, A.A. Anomaly Detection in Student Activity in Solving Unique Programming Exercises: Motivated Students against Suspicious Ones. *Data* **2023**, *8*, 129. [CrossRef]