Data Descriptor

Dataset of Program Source Codes Solving Unique Programming Exercises Generated by Digital Teaching Assistant

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

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

* Correspondence: liliya.demidova@rambler.ru (L.A.D.); andrianova@mirea.ru (E.G.A.); sovetov@mirea.ru (P.N.S.); worldbeater-dev@yandex.ru (A.V.G.)

Abstract: This paper presents a dataset containing automatically collected source codes solving unique programming exercises of different types. The programming exercises were automatically generated by the Digital Teaching Assistant (DTA) system that automates a massive Python programming course at MIREA—Russian Technological University (RTU MIREA). Source codes of the small programs grouped by the type of the solved task can be used for benchmarking source code classification and clustering algorithms. Moreover, the data can be used for training intelligent program synthesizers or benchmarking mutation testing frameworks, and more applications are yet to be discovered. We describe the architecture of the DTA system, aiming to provide detailed insight regarding how and why the dataset was collected. In addition, we describe the algorithms responsible for source code analysis in the DTA system. These algorithms use vector representations of programs based on Markov chains, compute pairwise Jensen–Shannon divergences of programs, and apply hierarchical clustering algorithms in order to automatically discover high-level concepts used by students while solving unique tasks. The proposed approach can be incorporated into massive programming courses when there is a need to identify approaches implemented by students.

Dataset: The data presented in this study are openly available in Zenodo at https://doi.org/10.5281/zenodo.7799971 (accessed on 10 June 2023).

Dataset License: CC-BY-4.0

Keywords: autograding; programming exercise generation; python; online education; program text analysis; source code analysis; abstract syntax trees; clustering algorithms

check for updates

Citation: 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. https://doi.org/10.3390/data8060109

Academic Editor: Kassim S. Mwitondi

Received: 4 April 2023
Revised: 11 June 2023
Accepted: 12 June 2023
Published: 14 June 2023

1. Summary

The digitalization of the economy leads to a constant expansion of the range of tasks facing specialists in information technologies, especially software developers. Research in static analysis of program source codes introduced a number of algorithms aiming to simplify and speed up the software development process and to improve the reliability and maintainability of software. Many such algorithms are now widely used in industry [1], including automated bug detection techniques based on static analysis of source codes [2–4]. Cyclomatic complexity [5] and cognitive complexity [6] checks are incorporated into static analyzers developed by SonarSource™, as well as into integrated development environments distributed by JetBrains™. These checks provide a quantitative estimate of how complex a function or a class method is, suggesting developers either automatically or manually refactor a code snippet in order to improve the maintainability of the piece of software they are working on. Recent research introduced a number of intelligent program text analysis algorithms for code completion [7–9], as well as for function name and variable name prediction [10–12]. The OpenAI™ Codex® model used by GitHub™ Copilot for code completion has been of great interest among software developers around the globe. There is still an increasing demand for further improvements of intelligent static analyzers of source codes.

Duplication of approaches to solving problems that arise during the software development process often takes place both in complex software systems with an extensive codebase and in open-source repositories that are not related to each other. Software developers often borrow publicly available source code that is then adapted to be integrated into the system being developed. However, a developer is also able to independently reinvent and reimplement a well-known approach to solving a specific problem, and the approach might be already implemented and used in the system being worked on. Recent research introduced a number of algorithms allowing the detection of similar patterns in source codes of programs [13,14]. Such algorithms can be used to perform automatic refactoring in order to get rid of code duplication, search for plagiarism, or to detect errors.

As a result of the expansion of the range of tasks in the IT sector of the economy, there is a demand for mass training of IT specialists in institutions of higher education. The massive nature of programming courses increases the burden on employees of educational organizations; course instructors have to come up with a sufficient number of various programming tasks under circumstances when regular cheating on the part of students becomes the norm. It is also difficult for an instructor of a massive programming course to equally dedicate time to all students and to check the solutions of every student in detail. The increased burden on teachers can result in emotional burnout, in a decrease in their desire to be creative and involved in the teaching process [15,16].

Recently, a few autograding systems were developed and incorporated into the educational process [17–19]. Such systems support automated checking of programming exercises; the teachers are expected to manually configure the exercises available in such systems, as well as test cases for them. In order to prevent cheating, plagiarism detection techniques are often used in autograding systems [19]; such techniques include clustering approaches that are based on pairwise comparisons of weighted keywords extracted from source codes [14], abstract syntax tree-based code clone detection algorithms [20,21], and methods that are based on the preliminary transformation of the program source code into vectors [22]. However, students can cheat plagiarism detection software by altering their source code to make the abstract syntax tree look completely different.

Recent research introduced algorithms for the automatic generation of unique programming exercises of different types based on the heuristic generate-and-test method [23]. On top of such algorithms, the Digital Teaching Assistant (DTA) system was developed in MIREA—Russian Technological University (RTU MIREA) [15,16], with the aim to completely eliminate the source code plagiarism problem in a massive Python programming course with over 1500 students. In the DTA system, every student receives a set of unique, automatically generated programming exercises of various types at the beginning of the semester. Students submit their source codes solving the programming exercises to DTA using a web interface, and the DTA system performs static and dynamic analysis of the source codes in a background process.

At the end of the semester, the received source codes are analyzed using intelligent algorithms in order to identify the most common approaches used by students while solving the unique programming exercises [24,25]. Several methods exist that allow mining idioms from source codes [26,27]. The aim of the current research is the identification of the knowledge gaps of students from different groups and departments in order to explain the least frequently used language features of the programming language to students before the final test [25]. Another aim of the source code analysis is the quality control of the automatic generators of programming exercises. A generator needs improvements if only typical, non-variable solutions are possible for the generated exercises [24]. In order to find similar approaches to solving unique programming exercises in a dataset containing more than 14,000 unique source codes that were checked and accepted by the DTA system, we applied code vectorization techniques based on Markov chains [25] with pairwise Jensen–Shannon divergences computation [28,29], as well as agglomerative hierarchical clustering algorithm with average linkage [30].

This data article provides the source codes received by the DTA system during the spring semester of 2022. The dataset contains only programs written in the Python programming language that were successfully checked during static and dynamic analysis and accepted by the DTA autograding system. All docstrings and comments were preliminarily removed from the source code datasets. The programs are grouped into 11 files, programs from the same file represent solutions for unique programming exercises of a given type.

The rest of the paper is structured as follows. Section 2 describes the DTA system and the different types of tasks available in the system, aiming to provide a complete picture of how and why the dataset was collected. Section 3 describes the collected dataset and methods used while preprocessing the source codes. Section 4 briefly describes the methods incorporated into the DTA analytics module and used for finding the most common approaches used by students while solving unique automatically generated programming exercises of different types.

2. Digital Teaching Assistant

The DTA system that automates the Python programming course at RTU MIREA consists of the core module, the web application module, and the analytics module. The architecture of the DTA system is shown in Figure 1.

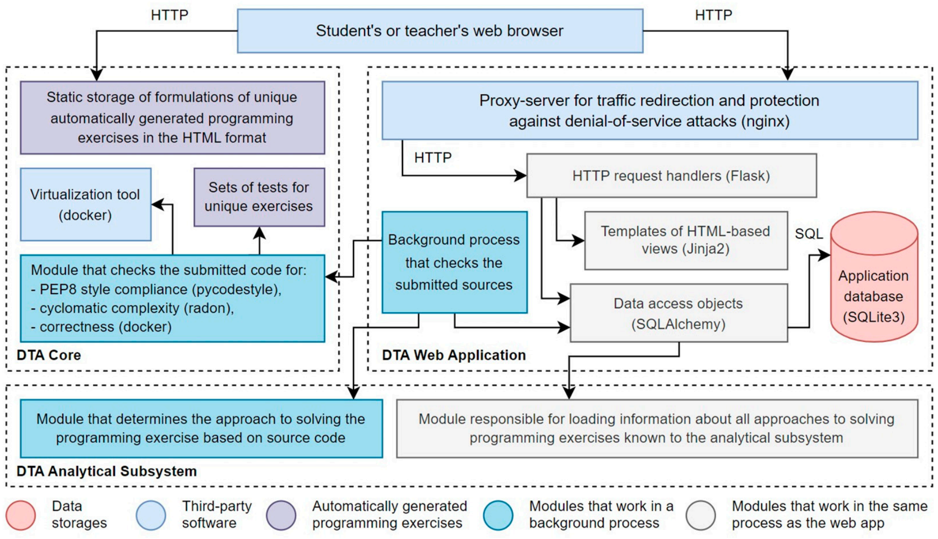

HTTP
Student's or teacher's web browser
HTTP
Static storage of formulations of unique automatically generated programming exercises in the HTML format
Virtualization tool (docker)
Sets of tests for unique exercises
Module that checks the submitted code for:
- PEP8 style compliance (pycodestyle),
- cyclomatic complexity (radon),
- correctness (docker)
DTA Core
Proxy-server for traffic redirection and protection against denial-of-service attacks (nginx)
HTTP
HTTP request handlers (Flask)
Templates of HTML-based views (Jinja2)
SQL
Background process that checks the submitted sources
Data access objects (SQLAlchemy)
Application database (SQLite3)
DTA Web Application
Module that determines the approach to solving the programming exercise based on source code
Module responsible for loading information about all approaches to solving programming exercises known to the analytical subsystem
DTA Analytical Subsystem
Data storages
Third-party software
Automatically generated programming exercises
Modules that work in a background process
Modules that work in the same process as the web app

Figure 1. Architecture of the Digital Teaching Assistant system.

2.1. Digital Teaching Assistant Core

The core of the system is responsible for the automatic generation of unique programming exercises for every student; every exercise is represented by a task formulation in the HTML format, a set of open tests, and a set of hidden tests. As shown in Figure 1, the core checks source codes received from students by performing static analysis, including PEP8 formatting checks and cyclomatic complexity [5] checks. Dynamic source code analysis is carried out afterward in order to verify the correctness of the submitted code on both open and hidden tests. The code is executed in a jailed Docker-based sandbox [31] with the gVisor secure runtime [32]. If an error has occurred while performing the tests, the DTA system only demonstrates the input data of only one erroneous test case.

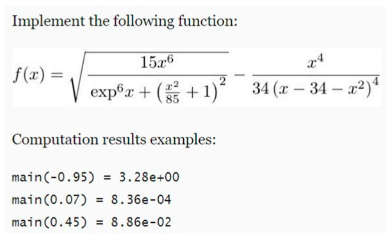
The programming exercise generation problem can be formulated as a constraint satisfaction problem and solved using the general generate-and-test method [23,33]. The formulations of the exercises are generated at the beginning of the semester by the core of the DTA system; they are represented by a set of HTML documents created using the Pandoc utility [34] from files in the Markdown format. LaTeX is used to render mathematical equations, and the DOT language is used to generate graphs using the graphviz tool [35]. The DTA core supports the generation of unique programming exercises of 11 different types. These types fall into two categories:

- Translation of some formal notation into source code;
- Conversion between two different data formats.

The 11 types of programming exercises are listed in Table 1. Formulation examples of the generated unique exercises of the first nine different types are shown in Figure 2. The long formulations for the 10th and 11th exercises are provided in Appendix A.

Table 1. Different types of unique programming exercises supported in DTA.

Task	Programming Exercise Type	Category of the Exercise Type
1.	Implement a function	Notation into code translation
2.	Implement a piecewise function	Notation into code translation
3.	Implement an iterative function	Notation into code translation
4.	Implement a recurrent function	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 bit field conversion	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 transformations	Conversion between data formats
11.	Implement a binary format parser	Conversion between data formats

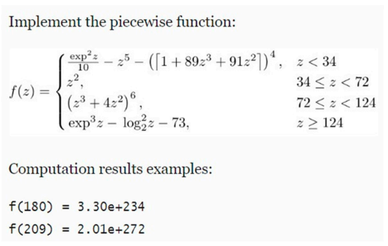

Implement the following function:

$$f(x) = \sqrt{\frac{15x^6}{\exp^6 x + \left(\frac{x^2}{85} + 1\right)^2}} - \frac{x^4}{34\left(x - 34 - x^2\right)^4}$$

Computation results examples:

```
main(-0.95) = 3.28e+00
main(0.07) = 8.36e-04
main(0.45) = 8.86e-02
```

(**a**)

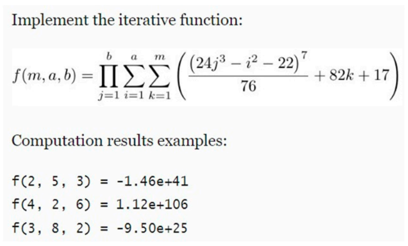

Implement the piecewise function:

$$f(z) = \begin{cases} \frac{\exp^2 z}{10} - z^5 - \left(\lceil 1 + 89z^3 + 91z^2 \rceil\right)^4, & z < 34 \\ z^2, & 34 \le z < 72 \\ \left(z^3 + 4z^2\right)^6, & 72 \le z < 124 \\ \exp^3 z - \log_2^2 z - 73, & z \ge 124 \end{cases}$$

Computation results examples:

```
f(180) = 3.30e+234
f(209) = 2.01e+272
```

(**b**)

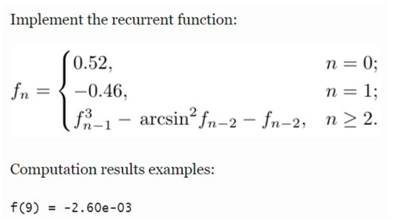

Implement the iterative function:

$$f(m, a, b) = \prod_{j=1}^{b} \sum_{i=1}^{a} \sum_{k=1}^{m} \left(\frac{\left(24j^3 - i^2 - 22\right)^7}{76} + 82k + 17 \right)$$

Computation results examples:

```
f(2, 5, 3) = -1.46e+41
f(4, 2, 6) = 1.12e+106
f(3, 8, 2) = -9.50e+25
```

(**c**)

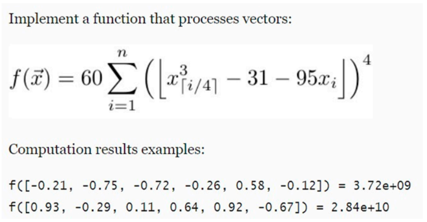

Implement the recurrent function:

$$f_n = \begin{cases} 0.52, & n = 0; \\ -0.46, & n = 1; \\ f_{n-1}^3 - \arcsin^2 f_{n-2} - f_{n-2}, & n \ge 2. \end{cases}$$

Computation results examples:

```
f(9) = -2.60e-03
```

(**d**)

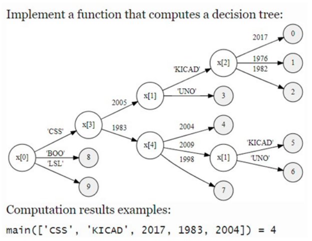

Implement a function that processes vectors:

$$f(\vec{x}) = 60 \sum_{i=1}^{n} \left(\left\lfloor x_{\lceil i/4 \rceil}^3 - 31 - 95x_i \right\rfloor \right)^4$$

Computation results examples:

```
f([-0.21, -0.75, -0.72, -0.26, 0.58, -0.12]) = 3.72e+09
f([0.93, -0.29, 0.11, 0.64, 0.92, -0.67]) = 2.84e+10
```

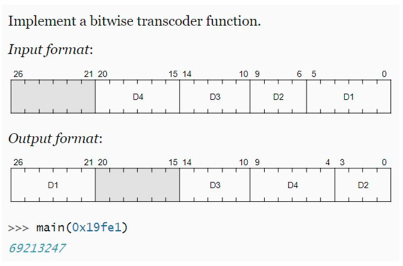
(**e**)


Implement a function that computes a decision tree:

Computation results examples:

```
main(['CSS', 'KICAD', 2017, 1983, 2004]) = 4
```

(**f**)

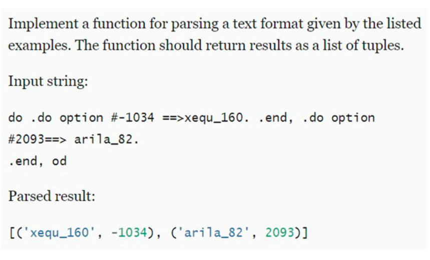

Implement a bitwise transcoder function.

Input format:

Output format:

```
>>> main(0x19fe1)
69213247
```

(**g**)


Implement a function for parsing a text format given by the listed examples. The function should return results as a list of tuples.

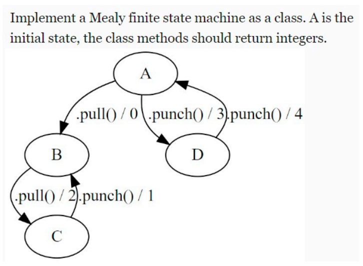
Input string:

```
do .do option #-1034 ==>xequ_160. .end, .do option
#2093==> arila_82.
.end, od
```

Parsed result:

```
[('xequ_160', -1034), ('arila_82', 2093)]
```

(**h**)


Implement a Mealy finite state machine as a class. A is the initial state, the class methods should return integers.

(**i**)

Figure 2. Formulation examples of unique programming exercises of 9 different types listed in Table 1, 10-th and 11-th exercise examples were omitted for brevity: (**a**) type 1; (**b**) type 2; (**c**) type 3; (**d**) type 4; (**e**) type 5; (**f**) type 6; (**g**) type 7; (**h**) type 8; (**i**) type 9.

As shown in Figure 2a–e, the first block of 11 programming exercises that students have to solve during the semester consists of mathematical notation to source code translation tasks. While solving such tasks, the students familiarize themselves with the standard library of the Python programming language. The second block (see Figure 2f–i) contains more complex automatically generated exercises, including conversion between two data formats tasks and translation of graph-based notations into code tasks. The last two task types, which are listed in Table 1 as the 10th and 11th exercise types, respectively, are considered the most complex ones. In the 10th task, a unique set of tabular data transformations described in natural language is generated, and students have to implement those transformations in code. The tables are represented by two-dimensional Python lists. In the 11th task, a specification of the automatically generated binary format is given, and students have to write a parser function based on the specification.

2.2. Digital Teaching Assistant Web Application

The DTA web application allows students and teachers to interact with the DTA core (see Figure 1) and automates the checking of the submitted source codes one by one by communicating with the DTA core in a background process using programmatic APIs. Additionally, the web application visualizes student performance statistics that are useful for teachers while tracking student activity. The DTA web application implements the Model-View-Controller (MVC) architecture [36] and consists of HTTP controllers, HTML view templates, business logic modules, and data access modules. The sequence diagram illustrating how a student interacts with the DTA system is shown in Figure 3.

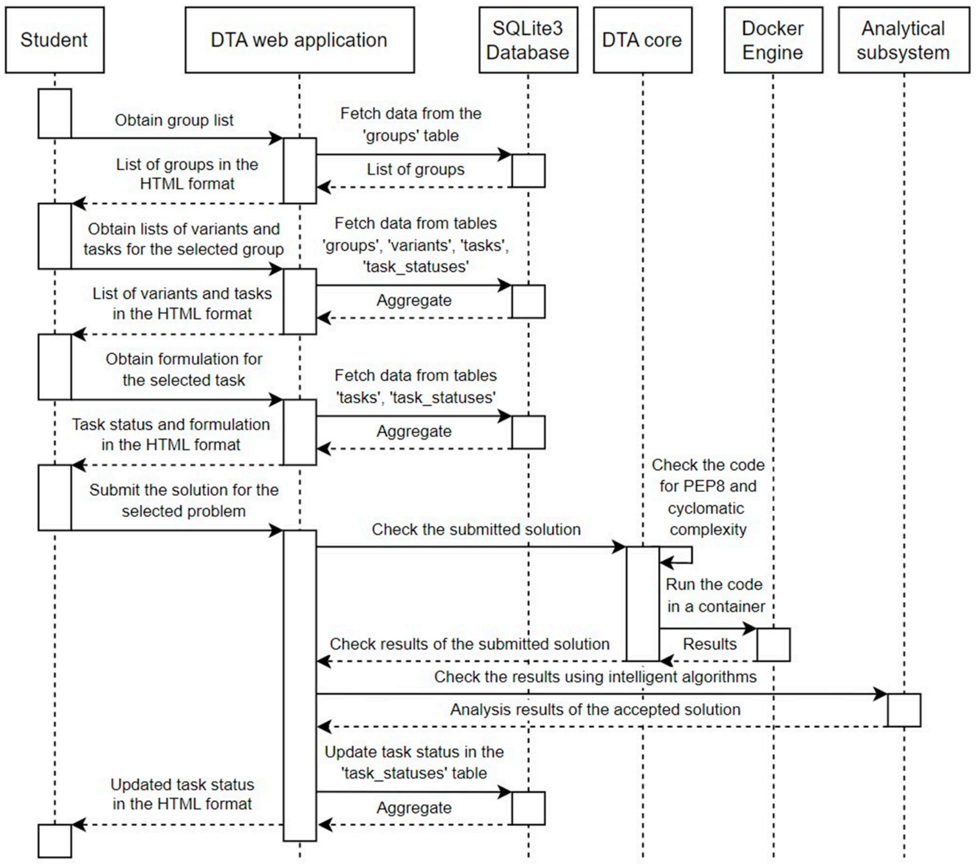
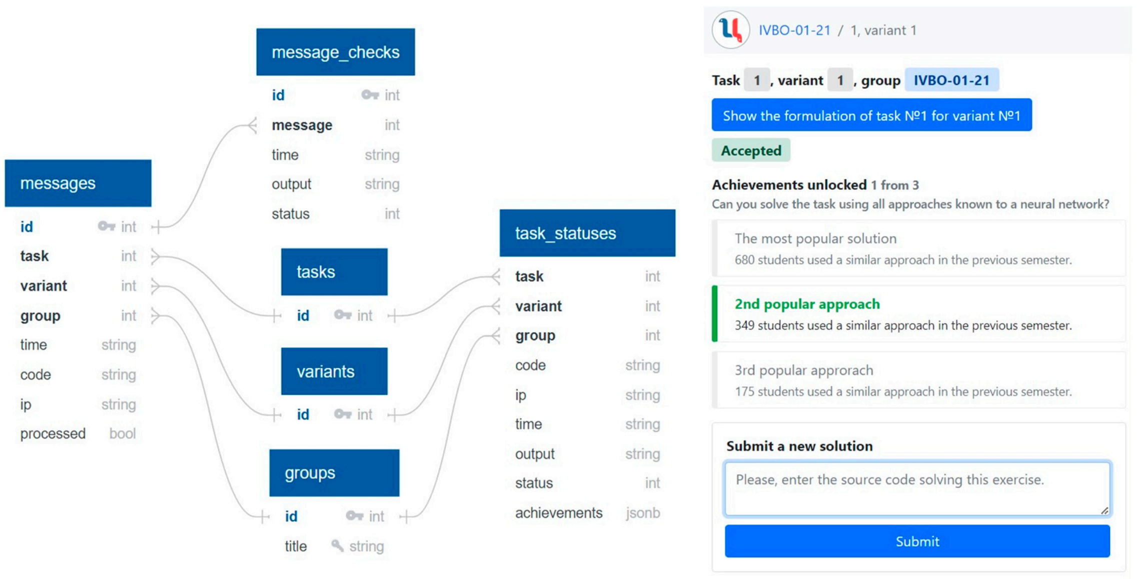

Student
DTA web application
SQLite3 Database
DTA core
Docker Engine
Analytical subsystem
Obtain group list
Fetch data from the 'groups' table
List of groups
List of groups in the HTML format
Obtain lists of variants and tasks for the selected group
Fetch data from tables 'groups', 'variants', 'tasks', 'task_statuses'
Aggregate
List of variants and tasks in the HTML format
Obtain formulation for the selected task
Fetch data from tables 'tasks', 'task_statuses'
Aggregate
Task status and formulation in the HTML format
Submit the solution for the selected problem
Check the code for PEP8 and cyclomatic complexity
Check the submitted solution
Run the code in a container
Results
Check results of the submitted solution
Check the results using intelligent algorithms
Analysis results of the accepted solution
Update task status in the 'task_statuses' table
Aggregate
Updated task status in the HTML format

Figure 3. Sequence diagram illustrating how students interact with the Digital Teaching Assistant.

According to Figure 3, a student first sees the list of groups, selects their group, then selects their variant and sees the list of tasks they have to solve by the end of the semester. Next, the student selects the task that they would like to complete, obtains the formulation of the task, solves it, and submits the solution to the DTA system. The DTA core checks the source code using static analyzers and containerized Docker environment, and if the solution is accepted, it is analyzed using intelligent algorithms in order to determine the implemented approach to solving the task. The results are then passed back to the DTA web application, which writes the received information to the database.

DTA uses the relational SQLite3 database engine; the data access modules shown in Figure 1 contain SQL queries implemented in the SQLAlchemy object-relational mapper (ORM) [37]. The entity-relationship (ER) diagram of the DTA web application database is shown in Figure 4a. The DTA analytical subsystem (see Figure 1) is responsible for determining approaches to solving unique programming exercises based on source codes. Every approach discovered by a student is treated as an educational achievement. The DTA web application interacts with the analytical subsystem after successfully checking a source code (see Figure 3); the decision made by the analytical subsystem indicating the discovered approach to solving a programming exercise is recorded in the "achievements" column of the "task statuses" table (see Figure 4a). The web interface displaying the achievements discovered by a student is shown in Figure 4b.



IVBO-01-21 / 1, variant 1

Task 1, variant 1, group IVBO-01-21

Show the formulation of task №1 for variant №1

Accepted

Achievements unlocked 1 from 3

Can you solve the task using all approaches known to a neural network?

The most popular solution
680 students used a similar approach in the previous semester.

2nd popular approach
349 students used a similar approach in the previous semester.

3rd popular approrach
175 students used a similar approach in the previous semester.

Submit a new solution

Please, enter the source code solving this exercise.

Submit

(**a**) (**b**)

Figure 4. Implementation details of the DTA web application: (**a**) Entity relationship diagram of the relational DTA database; (**b**) User interface of the solution submission page, displaying the current status of the task and the achievements discovered by a student.

According to Figure 4a, the DTA system database includes six tables and stores messages containing source codes received from students, check results for the messages, and task statuses containing the acceptance status of the last received source code. The information about groups, tasks, and variants is also stored in the database. The DTA system uses an event sourcing (ES)-based approach [38]. Hence, the state stored in the "task statuses" table (see Figure 4a) at time T can be restored by re-checking all the source codes from the messages table that were received before the time moment T.

3. Data Description

This article presents a dataset containing source codes solving unique programming exercises of 11 different types generated by the DTA system and also anonymized submissions and checks statistics. The implementation details of the DTA system, as well as the detailed insight regarding how and why the dataset was collected, are provided in Section 2. The dataset associated with this article contains the "messages.csv" file describing the history of source code submissions to the DTA web application. The “messages.csv” file contains a table with seven columns; the columns are described in Table 2.

Table 2. Structure of the “messages.csv” file available in the dataset associated with this paper.

Column	Column Description	Column Data Type	Possible Values
1.	Unique message identifier	Integer	Z
2.	Task number	Integer	{0, . . . , 10}
3.	Variant number	Integer	{0, . . . , 39}
4.	Group number	Integer	{1, . . . , 52}
5.	Message submission time	Timestamp	Z
6.	Task status	Integer	{2, 3}
7.	Message check time	Timestamp	Z

The table in the “messages.csv” file contains 66,553 rows total, excluding the header row. Every row of the table represents the characteristics of a message that was submitted to the DTA web application. As described in Table 2, the 1st column contains a monotonously increasing message identifier; the 2nd, 3rd, and 4th columns contain integers encoding task, variant, and group number, respectively; and the 5th and 7th columns contain timestamps indicating when the message was submitted and checked. Finally, the 6th column contains the check status (see Table 2), “2” means that the message was checked and accepted by the DTA core, and “3” means that there were errors during checks.

During the spring semester of 2022, the DTA system received 66,553 messages, 52,483 solutions were automatically rejected as erroneous, and 14,070 solutions were automatically accepted as correct. The comparison of rejected and accepted program counts is shown in Figure 5a. Figure 5c visualizes the total rejected and accepted program counts grouped by task types (see Table 1). The average accepted and rejected solutions count depending on the time of day is shown in Figure 5b.

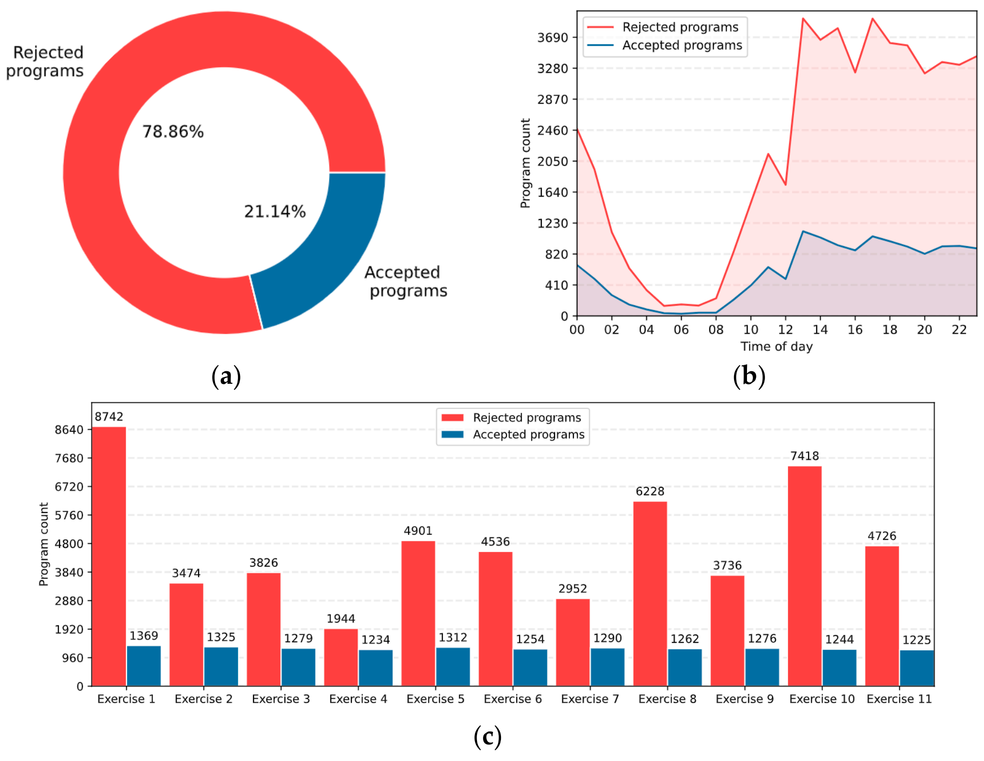
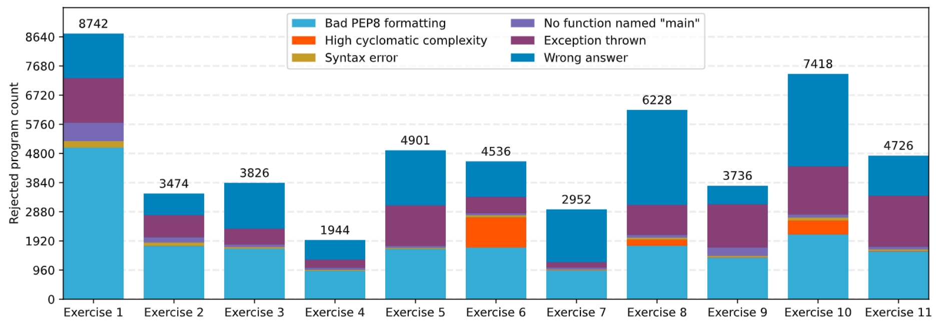


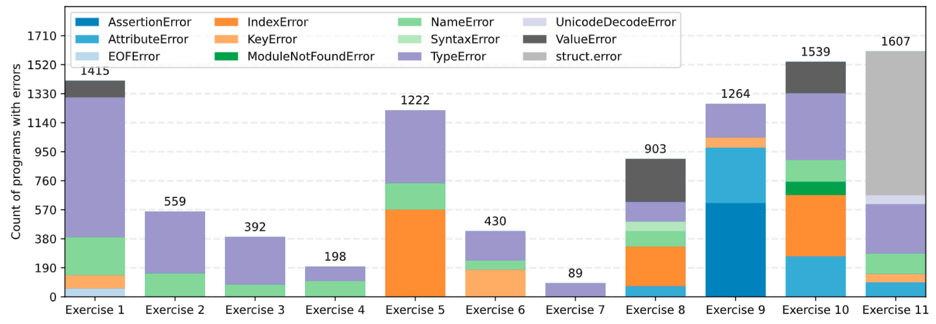
Figure 5. Statistics generated from the “messages.csv” file obtained from DTA: (**a**) Comparison of accepted and rejected program count; (**b**) Average accepted and rejected solutions count depending on time of day; (**c**) Comparison of accepted and rejected program count depending on task type.

As shown in Figure 5c, the first exercise is characterized by the largest number of attempts. This can be explained by the fact that the students who have never interacted with the DTA system before are familiarizing themselves with the system by submitting solutions and looking at the system's reaction. The first exercise is also characterized by the largest number of accepted programs among all exercises; most of the students successfully completed the first task but failed to solve other tasks, especially the last one.

Solutions to unique programming exercises submitted to the DTA system could be rejected due to various reasons. For example, a Python program could be rejected if it is not formatted according to the PEP8 standard, if the cyclomatic complexity of the solution is too high [5], or if there is no function named "main" in the submitted code, or if there is a syntax error, or if an exception has been thrown while testing the submitted solution in a containerized environment. Finally, if the output returned by the code is wrong, the solution is also rejected. The plot displaying the most common program rejection reasons is shown in Figure 6a. Moreover, different exceptions can be thrown while testing the code, which solves a programming exercise. The plot displaying the most commonly thrown exceptions is shown in Figure 6b.



(**a**)



(**b**)

Figure 6. Statistics extracted from the DTA database: (**a**) Most common reasons for rejection of programs solving unique programming exercises; (**b**) Most common exceptions thrown while testing code that solves a programming exercise.

The dataset associated with this paper contains files with programs solving automatically generated unique programming exercises grouped by exercise type; all exercise types are listed in Table 1. The title of the first file is "task-00.csv"; it contains programs solving unique tasks of type 1 (see Table 1 and Figure 2a). The title of the last file is "task-10.csv"; it contains programs solving tasks of type 11 (see Table 1).

The programs contained in the described files were used during the development of classifiers that form the basis of the DTA achievements system (see Figure 4b) [24,25]. The dataset contains only programs that were successfully checked and accepted by the DTA core. The accepted program count for each of the task types listed in Table 1 is shown in Figure 5c. The programs included in the dataset were preprocessed and filtered according to Algorithm 1 using functions from the Python standard library [38] with the aim of excluding any information that can be treated as personal from the dataset.

Algorithm 1: Filtering and preprocessing of source code datasets.

```
Input:  $S$—dataset with source codes of programs that successfully solved the tasks.
   1.   Define set of string literals $K$ = {print(, eval(, exec(, .com, .ru}.
   2.   Define $P = \varnothing$.
   3.   For each $s \in S$ do:
   4.       Construct an AST $a$ for $s$ using the ast.parse function [38].
   5.       Remove docstrings from $a$.
   6.       Restore the program text $s^*$ from $a$ using the ast.unparse function [38].
   7.       If $\nexists k \in K : k \subset s*$ then $P \leftarrow P \cup \{s*\}$.
   8.   End loop.
   9.   Return the filtered set of preprocessed programs.
```

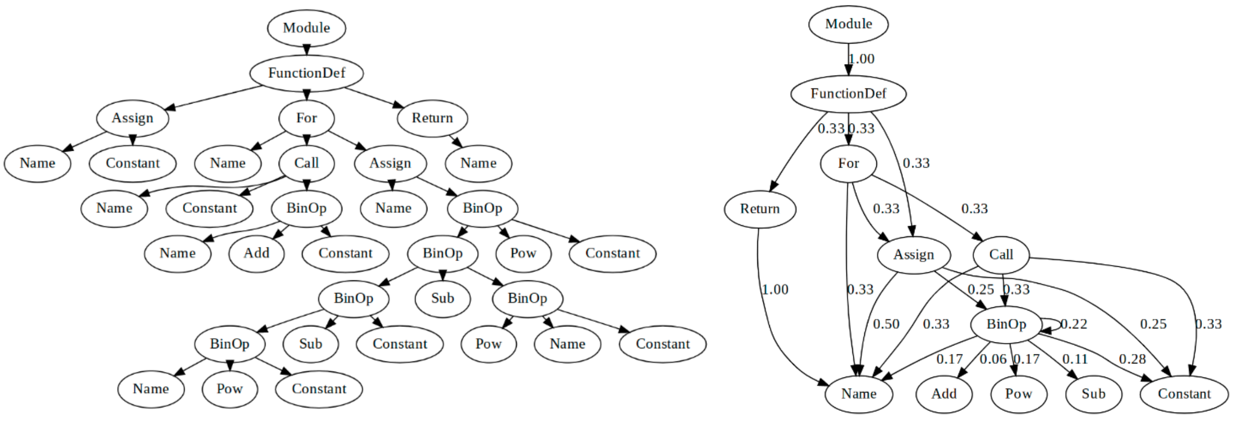
As described in Algorithm 1, each of the program source codes was first normalized by constructing an abstract syntax tree (AST) with a subsequent reconstruction of the source code only based on the information included in the AST. Prior to source code string reconstruction, the AST was traversed, and the documentation strings were removed. Python programs containing such instructions as “eval”, “exec”, or “print” were excluded.

The small programs grouped by the type of the solved task can be used for benchmarking source code classification algorithms [25], cluster analysis algorithms [24], and data visualization algorithms [39,40]. Additionally, the data can be used for training intelligent program synthesizers, benchmarking mutation testing frameworks, or evaluating static analyzers, and more applications are yet to be discovered.

4. Intelligent Source Code Analysis Algorithms Used in Digital Teaching Assistant

In this section, we show one of the possible applications of the dataset described in Section 3. The DTA analytics system includes a clustering module that processes a dataset of source codes solving tasks of a given type and detects the most common approaches used by students [24]. Additionally, the system includes a classification module that identifies the most similar cluster to a newly submitted solution by analyzing the source code [25]. This classification module lies at the core of the achievements system (see Figure 4b) that motivates students to solve their tasks using different approaches.

Both classification and clustering algorithms [24,25] use probabilistic context-free grammar-based representations of source codes. In other words, the ASTs of the source codes are converted into Markov chains, where the state space consists of types of vertices that occur in syntax trees, and weights of edges among the vertices represent probabilities of state transitions from one AST node type to the other. A sample AST is shown in Figure 7a, and a Markov chain for this AST is shown in Figure 7b.



(**a**) (**b**)

Figure 7. Example source code representation: (**a**) as an AST; (**b**) as a Markov chain for the AST.

The series of actions that converts an AST into a Markov chain is shown in Algorithm 2. First, the algorithm constructs an AST using tools from the Python standard library [38]. Next, the AST is simplified, and insignificant nodes are removed. Then, the set of vertex types and the set of edges are constructed. The weights of edges represent probabilities with which a vertex of one type references a vertex of another type.

Algorithm 2: Markov chain construction for the given source code.

```
Input: $s$—source code of a program solving a unique programming exercise.
1.  Construct an AST $A = (V,E)$ for program $s$ using the ast.parse function [38].
2.  Delete from $A$ vertices belonging to set {Load, Store, alias, arguments, args}.
3.  Define the mapping $g{:}V \to \mathrm{T}$ that maps a vertex $v \in V$ to its type.
4.  $M = \varnothing$.
5.  $T = \{g(v) : v \in V\}$.
6.  For each vertex type $t \in T$ do:
7.      $V_d = \{v_d : (v_s, v_d) \in E \wedge g(v_s) = t\}$—descendants of vertices of type $t$.
8.      $T_d = \{g(v_d) : v_d \in V_d\}$—types of descendants of vertices of type $t$.
9.      For each descendant vertex type $t_d \in T_d$ do:
10.         $\omega = \frac{1}{|V_d|}|\{v_d : v_d \in V_d \wedge g(v_d) = t_d\}|$—normed descendant count for $t_d$.
11.         $M \leftarrow M \cup \{(t, t_d, \omega)\}$.
12.     End loop.
13. End loop.
14. Return the weighted state transition graph $(T,M)$ of the Markov chain.
```

However, some ASTs built for programs solving unique exercises may contain vertices that are not present in other ASTs. Given that, the adjacency matrices of Markov chains constructed for such ASTs (see Figure 7b) can have different shapes, and thus their pairwise comparison is complicated. In order to overcome this limitation, we maintain the H set containing all vertices that occur in ASTs during the conversion of a dataset of source codes S into a set of Markov chain-based vector representations V. This approach allows converting all of the weighted adjacency matrices of Markov chains obtained using Algorithm 2 to the fixed shape $\mathrm{R}^{m \times m}$, where $m = |H|$. The matrices for programs belonging to the S set can then be transformed into vectors belonging to R^h, where $h = m^2$, and easily compared by using a distance metric that operates on vectors. Algorithm 3 illustrates the conversion process of the S set of source codes into the V set of their vector representations.

Algorithm 3: Conversion of source codes to vectors based on Markov chains.

```
Input: $S$—a set of source codes to be converted into vector representations.
1.  $H = \varnothing$—a set for AST node types that occur in Markov chains.
2.  $G = \varnothing$.
3.  For each source code $s \in S$ do:
4.      Construct a Markov chain $(T,M)$ for $s$ according to Algorithm 2.
5.      $H \leftarrow H \cup T$—add observed vertices to the $H$ set.
6.      $G \leftarrow G \cup \{M\}$—add a set of weighted edges of a Markov chain to $G$.
7.  End loop.
8.  $V = \varnothing$—a set for vector representations of program source codes.
9.  $m = |H|$—count of different AST node types that occur in Markov chains.
10. For each set of edges $M \in G$ do:
11.     Construct a weighted adjacency matrix $B \in \mathrm{R}^{m \times m}$ for graph $(H,M)$.
12.     Convert the $B \in \mathrm{R}^{m \times m}$ matrix to vector $\vec{v} \in \mathrm{R}^h$, where $h = m^2$.
13.     $V \leftarrow V \cup \left\{\vec{v}\right\}$.
14. End loop.
15. Return the $V$ set containing vector representations of source codes.
```

After obtaining vector representations based on Markov chains using Algorithm 3 from a set of source codes containing programs solving exercises of a given type, a pairwise distance matrix is constructed. As shown in [24], the DTA analytics system computes Jensen-Shannon divergence (JSD) [28,29] for each pair of vectors. JSD is given by

$$\mathrm{JSD}\left(\vec{v}_i, \vec{v}_j\right) = \frac{1}{2}\sum_{k=1}^{h} v_{ik}\log_2 \frac{2v_{ik}}{\left(v_{ik} + v_{jk}\right)} + \frac{1}{2}\sum_{k=1}^{h} v_{jk}\log_2 \frac{2v_{jk}}{\left(v_{ik} + v_{jk}\right)}, \tag{1}$$

where $\vec{v}_i$ and $\vec{v}_j$ denote vector representations of programs based on Markov chains (see Algorithms 2 and 3), v_{ik} denotes k-th component of vector $\vec{v}_i$, v_{jk} denotes k-th component of vector $\vec{v}_j$, h denotes component count in vectors $\vec{v}_i$ and $\vec{v}_j$.

After the construction of a pairwise distance matrix, the matrix is passed to the agglomerative hierarchical clustering algorithm with average linkage [24,30]. The implementation of this algorithm used in DTA is based on the sklearn library [41]. As described in [24], the optimal cluster count is selected based on the silhouette score metric [42] for every dataset containing source codes solving programming exercises. The count of clusters with more than 10 objects for different datasets is shown in Figure 8.

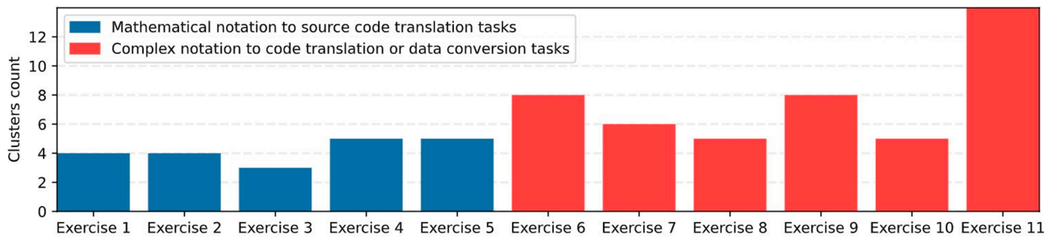


Figure 8. Best cluster counts for datasets containing programs solving exercises of different types.

In the DTA system, the exercises become more complex with the increase of the exercise identifier. According to Figure 8, datasets with source codes of programs that implement mathematical formulas (see Figure 2a–e) contain fewer high-level concepts that can be used as solutions when compared to more complex tasks (see Figure 2f–i). Solutions for unique exercises of type 11 appear to be the most varied.

Examples of the approaches to solving programming exercises that were discovered by applying the hierarchical clustering algorithm to the pairwise distance matrix obtained for Markov chain-based vector representations of programs from the dataset contained in file "task-03.csv" from the dataset associated with this paper are shown in Figure 9.

```
def main(n):
    if n == 0:
        return 1.0
    elif n == 1:
        return 0.57
    else:
        x = main(n - 1) ** 3 / 33
        return x + main(n - 2) ** 2
```
(**a**)

```
from math import ceil
def main(n):
    if n == 0:
        return -0.68
    if n == 1:
        return 0.56
    c = ceil(main(n - 2)) ** 2
    return main(n - 1) + c
```
(**b**)

```
def main(n):
    mas = [-0.3]
    for i in range(1, n+1):
        e = mas[i-1]
        m = e ** 3 / 23
        x = m + e + e ** 2
        mas.append(x)
    return mas[n]
```
(**c**)

```
def main(n):
    res1 = -0.31
    res2 = -0.44
    for i in range(1, n):
        t = res1 - res2 ** 3 / 38
        res1 = res2
        res2 = t
    return temp
```
(**d**)

```
from math import atan

def main(n):
    return -0.82 if n == 0 \
        else atan(70 * main(n-1) - \
        (main(n-1) ** 3) / 4) ** 3
```
(**e**)

Figure 9. Examples of the approaches to solving unique programming exercises of type 4 (see Table 1 and Figure 2d) written in the Python programming language: (**a**) recurrent function implementation using a conditional operator; (**b**) recurrent function implemented using a conditional operator with early return; (**c**) recursion implemented as a loop and a list; (**d**) recursion implemented as a loop with temporary variables; (**e**) recursion implemented with a ternary operator.

The clusters obtained by applying the described methods and algorithms are then used to train Extreme Learning Machine-based classifiers, and training and evaluation of the classifiers are described in detail in [25]. The obtained classifiers lie at the core of the educational achievements system implemented in DTA (see Figure 4b).

5. Discussion

The dataset presented and described in this paper contains source codes of programs solving unique exercises of different types. The programs were successfully checked and accepted by Digital Teaching Assistant (DTA) [15,16] in the spring semester of 2022. The DTA system automates a massive Python programming course at RTU MIREA, the architecture of the system is described in Section 2.

The source codes of programs included in the dataset (see Section 3) are grouped by the type of tasks that they solve (see Table 1 and Figure 2). Each of such source code groups was processed using Algorithm 1, aiming to exclude any information that can be treated as personal from the dataset. During the operation of DTA in the spring semester of 2022, more than 65,000 programs were submitted to the system by programming course students, and more than 52,000 programs were automatically rejected (see Figure 5a) due to different reasons. The most common rejection reasons include formatting issues, too high cyclomatic complexity [5], numerical issues, or thrown exceptions (see Figure 6a). The most common exceptions include type error, index error, name error, and assertion error (see Figure 6b). The erroneous programs are not included in the source code dataset associated with this paper. In addition to source codes of programs, the dataset also includes an event log of code submissions aiming to provide a complete picture of how the data were collected (see Figure 5a,b). The event log can also be a subject for further analysis in terms of anomaly and outburst detection in the time series of the submitted messages.

In Section 4, we provide a detailed description of methods and algorithms used in the DTA system for the automatic discovery of the common approaches to solving programming exercises of a given type as an example of how to extract useful information from the provided dataset of source codes. The DTA system first constructs ASTs for source codes using the Python standard library [38] (see Figure 7a) and then transforms the obtained syntax trees into Markov chains (see Figure 7b) using Algorithm 2. The Markov chain-based representations are then transformed into vectors by applying Algorithm 3 to the obtained weighted graphs. Next, a pairwise distance matrix is built using the obtained vectors and Equation (1); the matrix is then fed to the agglomerative hierarchical clustering algorithm as described in [24]. Examples of the discovered approaches used by students while solving programming exercises of type 4 (see Table 1) are shown in Figure 9. The obtained clusters are then used to train Extreme Learning Machine-based classifiers [25] that lie at the core of the DTA achievements system (see Figure 4b).

The datasets of small programs solving unique programming exercises of different types can be used for training intelligent static analyzers, for the development of program synthesizers, for the evaluation of high-level concept miners [26], for benchmarking mutation testing frameworks, more applications of the datasets are yet to be discovered. The source code analysis algorithms described in this paper can be used for automated detection of corporate programming standards violation [43], vulnerability detection [44], and authorship identification [45].

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

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

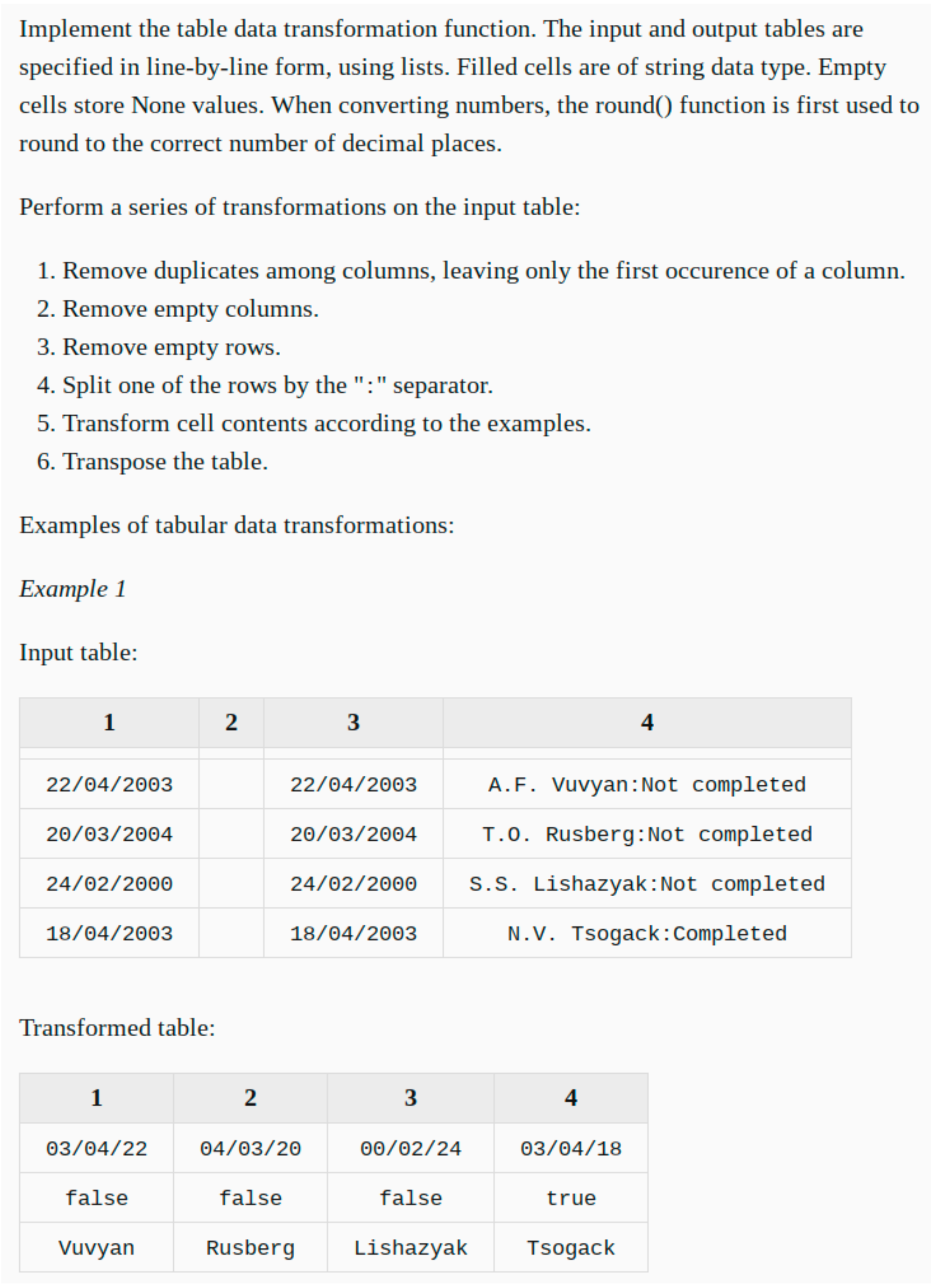
Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are openly available in Zenodo at https://doi.org/10.5281/zenodo.7799971 (accessed on 10 June 2023).

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

Appendix A

In Figure 2, formulation examples for the 10th tabular data transformation task and for the 11th binary data format parser task were omitted for brevity, so we provide the examples of formulations for these two tasks in Figures A1 and A2, respectively.



Implement the table data transformation function. The input and output tables are specified in line-by-line form, using lists. Filled cells are of string data type. Empty cells store None values. When converting numbers, the round() function is first used to round to the correct number of decimal places.

Perform a series of transformations on the input table:

1. Remove duplicates among columns, leaving only the first occurence of a column.
2. Remove empty columns.
3. Remove empty rows.
4. Split one of the rows by the ":" separator.
5. Transform cell contents according to the examples.
6. Transpose the table.

Examples of tabular data transformations:

Example 1

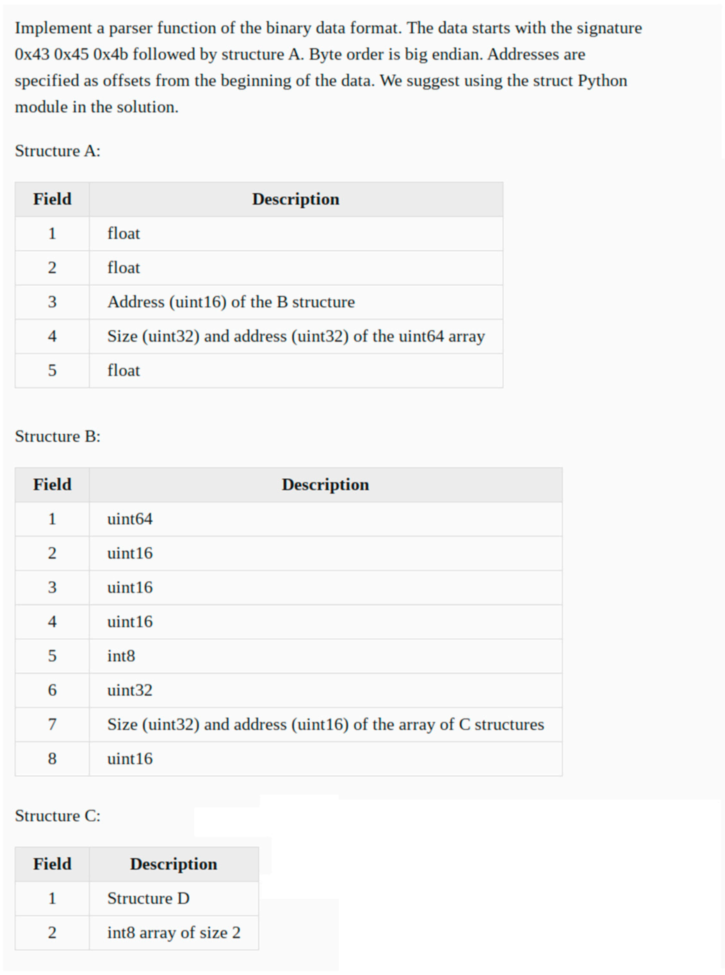
Input table:

1	2	3	4
22/04/2003		22/04/2003	A.F. Vuvyan:Not completed
20/03/2004		20/03/2004	T.O. Rusberg:Not completed
24/02/2000		24/02/2000	S.S. Lishazyak:Not completed
18/04/2003		18/04/2003	N.V. Tsogack:Completed

Transformed table:

1	2	3	4
03/04/22	04/03/20	00/02/24	03/04/18
false	false	false	true
Vuvyan	Rusberg	Lishazyak	Tsogack

Figure A1. Formulation example for the 10th tabular data transformation task (see Table 1).



Implement a parser function of the binary data format. The data starts with the signature 0x43 0x45 0x4b followed by structure A. Byte order is big endian. Addresses are specified as offsets from the beginning of the data. We suggest using the struct Python module in the solution.

Structure A:

Field	Description
1	float
2	float
3	Address (uint16) of the B structure
4	Size (uint32) and address (uint32) of the uint64 array
5	float

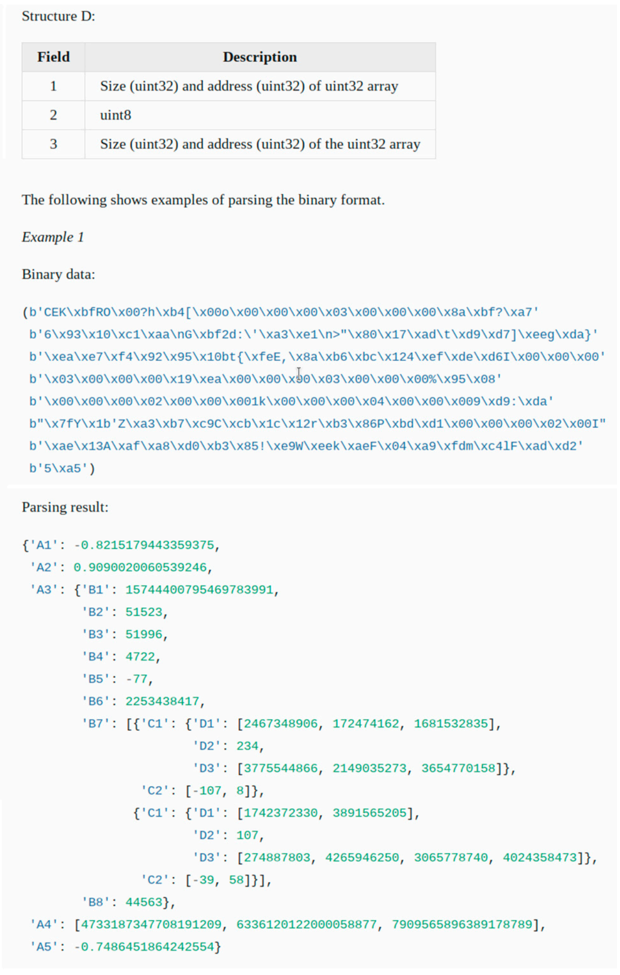
Structure B:

Field	Description
1	uint64
2	uint16
3	uint16
4	uint16
5	int8
6	uint32
7	Size (uint32) and address (uint16) of the array of C structures
8	uint16

Structure C:

Field	Description
1	Structure D
2	int8 array of size 2

(a)



Structure D:

Field	Description
1	Size (uint32) and address (uint32) of uint32 array
2	uint8
3	Size (uint32) and address (uint32) of the uint32 array

The following shows examples of parsing the binary format.

Example 1

Binary data:

```
(b'CEK\xbfRO\x00?h\xb4[\x00o\x00\x00\x00\x03\x00\x00\x00\x8a\xbf?\xa7'
 b'6\x93\x10\xc1\xaa\nG\xbf2d:\'\xa3\xe1\n>"\x80\x17\xad\t\xd9\xd7]\xeeg\xda}'
 b'\xea\xe7\xf4\x92\x95\x10bt{\xfeE,\x8a\xb6\xbc\x124\xef\xde\xd6I\x00\x00\x00'
 b'\x03\x00\x00\x00\x19\xea\x00\x00\x00\x03\x00\x00\x00%\x95\x08'
 b'\x00\x00\x00\x02\x00\x00\x001k\x00\x00\x00\x04\x00\x00\x009\xd9:\xda'
 b"\x7fY\x1b'Z\xa3\xb7\xc9C\xcb\x1c\x12r\xb3\x86P\xbd\xd1\x00\x00\x00\x02\x00I"
 b'\xae\x13A\xaf\xa8\xd0\xb3\x85!\xe9W\xeek\xaeF\x04\xa9\xfdm\xc4lF\xad\xd2'
 b'5\xa5')
```

Parsing result:

```
{'A1': -0.8215179443359375,
 'A2': 0.9090020060539246,
 'A3': {'B1': 15744400795469783991,
        'B2': 51523,
        'B3': 51996,
        'B4': 4722,
        'B5': -77,
        'B6': 2253438417,
        'B7': [{'C1': {'D1': [2467348906, 172474162, 1681532835],
                       'D2': 234,
                       'D3': [3775544866, 2149035273, 3654770158]},
                'C2': [-107, 8]},
               {'C1': {'D1': [1742372330, 3891565205],
                       'D2': 107,
                       'D3': [274887803, 4265946250, 3065778740, 4024358473]},
                'C2': [-39, 58]}],
        'B8': 44563},
 'A4': [4733187347708191209, 6336120122000058877, 7909565896389178789],
 'A5': -0.7486451864242554}
```

(b)

Figure A2. Formulation example for the 11th binary format parser task (see Table 1): (**a**) format definition; (**b**) continuation of the format definition, input, and output examples.

References

1. Emanuelsson, P.; Nilsson, U. A Comparative Study of Industrial Static Analysis Tools. *Electron. Notes Theor. Comput. Sci.* **2008**, *217*, 5–21. [CrossRef]
2. Ayewah, N.; Pugh, W.; Morgenthaler, J.D.; Penix, J. Using Static Analysis to Find Bugs. *IEEE Softw.* **2008**, *25*, 22–29. [CrossRef]
3. Jiang, H.; Yang, H.; Qin, S.; Su, Z.; Zhang, J.; Yan, J. Detecting Energy Bugs in Android Apps Using Static Analysis. In Proceedings of the Formal Methods and Software Engineering: 19th International Conference on Formal Engineering Methods, ICFEM 2017, Xi'an, China, 13–17 November 2017; Springer International Publishing: Cham, Switzerland, 2017; pp. 192–208.
4. McPeak, S.; Gros, C.H.; Ramanathan, M.K. Scalable and Incremental Software Bug Detection. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, Saint Petersburg, Russia, 18–26 August 2013; pp. 554–564.
5. Ebert, C.; Cain, J.; Antoniol, G.; Counsell, S.; Laplante, P. Cyclomatic complexity. *IEEE Softw.* **2016**, *33*, 27–29. [CrossRef]
6. 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; pp. 57–58.
7. 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; pp. 213–222.
8. Svyatkovskiy, A.; Zhao, Y.; Fu, S.; Sundaresan, N. Pythia: Ai-assisted Code Completion System. In Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, Anchorage, AK, USA, 3–7 August 2019; pp. 2727–2735.
9. Terada, K.; Watanobe, Y. Code Completion for Programming Education Based on Recurrent Neural Network. In Proceedings of the 2019 IEEE 11th International Workshop on Computational Intelligence and Applications (IWCIA), Hiroshima, Japan, 9–10 November 2019; pp. 109–114.
10. Alon, U.; Zilberstein, M.; Levy, O.; Yahav, E. code2vec: Learning Distributed Representations of Code. In Proceedings of the ACM on Programming Languages, Providence, RI, USA, 22–26 June 2019; pp. 40:1–40:29.

11. 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 (ICSE), Madrid, Spain, 22–30 May 2021; pp. 574–586.
12. Lacomis, J.; Yin, P.; Schwarts, E.; Allamanis, M.; Goues, C.; Neubig, G.; Vasilescu, B. Dire: A Neural Approach to Decompiled Identifier Naming. In Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019; pp. 628–639.
13. Marcus, A.; Maletic, J.I. Identification of High-level Concept Clones in Source Code. In Proceedings of the 16th Annual International Conference on Automated Software Engineering (ASE 2001), San Diego, CA, USA, 26–29 November 2001; pp. 107–114.
14. Moussiades, L.; Vakali, A. PDetect: A Clustering Approach for Detecting Plagiarism in Source Code Datasets. *Comput. J.* **2005**, *48*, 651–661. [CrossRef]
15. 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; pp. 272–276.
16. Andrianova, E.G.; Demidova, L.A.; Sovetov, P.N. Pedagogical Design of a Digital Teaching Assistant in Massive Professional Training for the Digital Economy. *Russ. Technol. J.* **2022**, *10*, 7–23. [CrossRef]
17. Mekterović, I.; Brkić, L.; Milašinović, B.; Baranović, M. Building a Comprehensive Automated Programming Assessment System. *IEEE Access* **2020**, *8*, 81154–81172. [CrossRef]
18. Queirós, R.A.P.; Leal, J.P. PETCHA: A Programming Exercises Teaching Assistant. In Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education, Haifa, Israel, 3–5 July 2012; pp. 192–197.
19. Combéfis, S. Automated Code Assessment for Education: Review, Classification and Perspectives on Techniques and Tools. *Software* **2022**, *1*, 3–30. [CrossRef]
20. Jiang, L.; Misherghi, G.; Su, Z.; Glondu, S. Deckard: Scalable and Accurate Tree-Based Detection of Code Clones. In Proceedings of the 29-th International Conference on Software Engineering (ICSE'07), Minneapolis, MN, USA, 20–26 May 2007; IEEE: Pistacaway, NJ, USA, 2007; pp. 96–105.
21. Kustanto, C.; Liem, I. Automatic Source Code Plagiarism Detection. In Proceedings of the 2009 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, Daegu, Republic of Korea, 27–29 May 2009; IEEE: Pistacaway, NJ, USA, 2009; pp. 481–486.
22. Yasaswi, J.; Kailash, S.; Chilupuri, A.; Purini, S.; Jawahar, C.V. Unsupervised Learning-Based Approach for Plagiarism Detection in Programming Assignments. In Proceedings of the 10th Innovations in Software Engineering Conference, Jaipur, India, 5–7 February 2017; Association for Computing Machinery: New York, NY, USA, 2017; pp. 117–121.
23. Sovietov, P. Automatic Generation of Programming Exercises. In Proceedings of the 2021 1st International Conference on Technology Enhanced Learning in Higher Education (TELE), Lipetsk, Russia, 7–9 July 2021; IEEE: Pistacaway, NJ, USA, 2021; pp. 111–114.
24. Demidova, L.A.; Sovietov, P.N.; Gorchakov, A.V. Clustering of Program Source Text Representations Based on Markov Chains. *Vestn. Ryazan State Radio Eng. Univ.* **2022**, *81*, 51–64.
25. 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]
26. 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; pp. 472–483.
27. Pham, H.S.; Nijssen, S.; Mens, K.; Nucci, D.D.; Molderez, T.; Roover, C.D.; Fabry, J.; Zaytsev, V. Mining Patterns in Source Code using Tree Mining Algorithms. In Proceedings of the Discovery Science: 22nd International Conference, DS 2019, Split, Croatia, 28–30 October 2019; Springer International Publishing: New York, NY, USA, 2019; pp. 471–480.
28. Lin, J. Divergence Measures Based on the Shannon Entropy. *IEEE Trans. Inf. Theory* **1991**, *37*, 145–151. [CrossRef]
29. Nielsen, F. On the Jensen–Shannon Symmetrization of Distances Relying on Abstract Means. *Entropy* **2019**, *21*, 485. [CrossRef] [PubMed]
30. Sokal, R.R.; Michener, C.D. A Statistical Method for Evaluating Systematic Relationships. *Evolution* **1957**, *11*, 130–162.
31. Peveler, M.; Maicus, E.; Cutler, B. Comparing Jailed Sandboxes vs Containers Within an Autograding System. In Proceedings of the 50th ACM Technical Symposium on Computer Science Education, Minneapolis, MN, USA, 27 February–2 March 2019; pp. 139–145.
32. Wang, X.; Du, J.; Liu, H. Performance and Isolation Analysis of RunC, gVisor and Kata Containers Runtimes. *Clust. Comput.* **2022**, *25*, 1497–1513. [CrossRef]
33. Brailsford, S.C.; Potts, C.N.; Smith, B.M. Constraint Satisfaction Problems: Algorithms and Applications. *Eur. J. Oper. Res.* **1999**, *119*, 557–581. [CrossRef]
34. Mailund, T. *Introducing Markdown and Pandoc: Using Markup Language and Document Converter*; Apress: Berkeley, CA, USA, 2019; 139p.
35. 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]

36. Fowler, M.; Rice, D.; Foemmel, M.; Hieatt, E.; Mee, R.; Stafford, R. *Patterns of Enterprise Application Architecture*; Addison-Wesley Professional: Boston, MA, USA, 2002; Chapter 14.
37. Bayer, M.; Brown, A.; Wilson, G. SQLAlchemy. *Archit. Open-Source Appl.* **2012**, *2*, 20.
38. Python Software Foundation. AST—Abstract Syntax Trees. 2023. Available online: https://docs.python.org/3/library/ast.html (accessed on 28 March 2023).
39. Wang, Y.; Huang, H.; Rudin, C.; Shaposhnik, Y. Understanding How Dimension Reduction Tools Work: An Empirical Approach to Deciphering t-SNE, UMAP, TriMAP, and PaCMAP for Data Visualization. *J. Mach. Learn. Res.* **2021**, *22*, 9129–9201.
40. Demidova, L.A.; Gorchakov, A.V. Fuzzy Information Discrimination Measures and Their Application to Low Dimensional Embedding Construction in the UMAP Algorithm. *J. Imaging* **2022**, *8*, 113. [CrossRef] [PubMed]
41. 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.
42. Shahapure, K.R.; Nicholas, C. Cluster Quality Analysis Using Silhouette Score. In Proceedings of the 2020 IEEE 7th international conference on data science and advanced analytics (DSAA), Sydney, Australia, 6–9 October 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 747–748.
43. 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, University Town, Singapore, 14–16 November 2022; pp. 696–708.
44. Russell, R.L.; Kim, L.; Hamilton, L.H.; Lazovich, T.; Harer, J.A.; Ozdemir, O.; Ellingwood, P.M.; McConley, M.W. Automated vulnerability detection in source code using deep representation learning. In Proceedings of the 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.
45. Bogomolov, E.; Kovalenko, V.; Rebryk, Y.; Baccheli, A.; Bryksin, T. Authorship attribution of source code: A language-agnostic approach and applicability in software engineering. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, 23–28 August 2021; pp. 932–944.