IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received May 8, 2020, accepted May 15, 2020, date of publication May 20, 2020, date of current version June 4, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2996146

Plagiarism Detection in Computer Programming Using Feature Extraction From Ultra-Fine-Grained Repositories

VEDRAN LJUBOVIC AND ENIL PAJIC
Faculty of Electrical Engineering, University of Sarajevo, Sarajevo 71000, Bosnia and Herzegovina
Corresponding author: Vedran Ljubovic (vljubovic@etf.unsa.ba)

ABSTRACT Detecting instances of plagiarism in student homework, especially programming homework, is an important issue for practitioners. In the past decades, several tools have emerged that are able to effectively compare large corpora of homeworks and sort pairs by degree of similarity. However, those tools are available to students as well, allowing them to experiment and develop elaborate methods for evading detection. Also, such tools are unable to detect instances of ‘‘external plagiarism’’ where students obtained unethical help from sources not among other students of the same course. One way to battle this problem is to monitor student activity while solving their homeworks using a cloud-based integrated development environment (IDE) and detect suspicious behaviours. Each editing event in program source can be stored as a new commit to create a form of ultra-fine-grained source code repository. In this paper, the authors propose several new features that can be extracted from such repositories with the purpose of building a comprehensive profile of each individual developer. Machine learning techniques were used to detect suspicious behaviours, which allowed the authors to significantly improve upon the performance of more traditional plagiarism detection tools.

INDEX TERMS Repository mining, plagiarism detection, ultra-fine-grained repositories, machine learning, feature extraction.

I. INTRODUCTION

Academic dishonesty in computer science education is a considerable problem for any course that is based on independent assignments (homework). Detecting such behavior has been a subject of active research for almost 40 years. Several tools for plagiarism detection have reached a level of maturity that have allowed them to be publicly released and used in the academic setting [1]–[3].

Changes in the cultural and social environment in the past decade present a new challenge for practical plagiarism detection. Students increasingly publish their homework solutions on social networks, sometimes along with elaborate instructions on how to modify them in a way that avoids detection. Also, many students turn to ‘‘rent-a-coder’’ type websites to obtain unique, but dishonest, homework solutions.

This paper presents an attempt to address these problems by logging all activities that students perform while solving homework in a cloud-based integrated development environment (IDE). This log is treated as a source code repository, and repository mining techniques are used to build ‘‘developer profiles’’ allowing detection of suspicious behaviours. The paper is focused on plagiarism detection, but a similar approach could be used for other purposes in an educational setting, such as detecting students who might need special attention or additional instruction in certain areas.

A typical problem with source repositories is their low resolution. In this paper, the authors propose creating a form of ultra-fine-grained repository by using the IDE ‘‘autosave’’ feature to record minor changes, down to individual keystrokes. ‘‘Autosave’’ is usually implemented in a way that reduces the impact to system performance, such that a document is saved after a certain period of inactivity (typically 1 second).

File system monitoring tools are then used to automatically commit these changes to a Subversion (SVN) shadow repository that is invisible to the user. This allows researchers to use well-known tools for analysis and debugging of SVN

The associate editor coordinating the review of this manuscript and approving it for publication was Joey Tianyi Zhou.

repositories. This approach is also compatible with all known IDEs and editors that support autosave. Further, since ''Run program'' and ''Test'' buttons also result in the creation of various output files (executable, core dump, etc.), these can be analyzed to detect IDE interactions and log unit test results.

The method proposed in this paper is a form of Change Recording or Change Logging [4]; however, the system stops short of inferring semantics of change and instead records concrete, atomic changes to the source code text with the expectation that this high-resolution log will be further analyzed using machine learning methods.

To demonstrate the usefulness of this approach, ultra-fine-grained repositories were created for 300 students working on programming homework in an introductory university programming course. A set of features was extracted from these repositories and used to improve plagiarism detection. The hypothesis presented in this paper is that students who plagiarize homework use their IDE differently than those who work on their own.

Note that the solution presented here is cloud-based, so no tools were installed on student computers, especially tools that would monitor activity outside the IDE code editor window (i.e. keylogger), since that would be unethical. Students were asked to sign a consent form that allows use of their anonymized usage history for research purposes, and data for those who refused to sign this form were excluded from the dataset.

The rest of this paper is structured as follows: Chapter 2 provides a review of related work, with an emphasis on the current state of plagiarism detection. Chapter 3 describes research methodology used, primarily the comparison metrics. Chapter 4 gives a detailed description of the new dataset developed as part of authors' ongoing research. The topic of Chapter 5 is the new approach to plagiarism detection proposed in this paper. First, an explanation of feature extraction process and the semantics of extracted features is provided in subsection 5.A. Subsection 5.B details the machine learning approach based on neural networks used to classify submitted programs. Chapter 6 gives a comparison with existing tools for plagiarism detection, while Chapter 7 presents conclusions and possible future research directions.

II. RELATED WORK

This paper relies on research in two areas: software repository mining and plagiarism detection. To the best of the authors' knowledge, the sole paper that has combined these areas to date is [5].

One of the promises of repository mining is the extraction of information from source control systems such as Concurrent Versioning System (CVS) that would allow researchers to create comprehensive profiles of each individual developer, in order to predict their future contribution and possible problems. Such methods could also be useful in academic settings [6].

Experimental results in these application areas often show limited usefulness. The problem is that individual commits in common source control systems, as used in professional or open-source development, are too coarse-grained to extract information that would be useful for many purposes [7].

To address this problem, researchers have developed tools that preprocess repository data to increase its resolution [8]. This results in a type of fine-grained repository where units of data are no longer commits, but *Source Code Changes (SCC)*.

However, developers are notoriously lazy in doing frequent commits and commits that reflect semantic changes. We cannot force developers to change their habits; therefore, a recent research direction is to use the IDE itself as source of information about developer activities [4].

IDE usage patterns have proven useful in various application areas. In particular, data obtained from IDEs was used to increase the fidelity of code changes and thus detect unknown change patterns [9].

IDE interactions have been used to build recommendation systems [10]. Another possible use for developer interactions collected from IDEs is change prediction [11]. The most commonly used platforms for collecting IDE interactions are Eclipse Mylyn and Usage Data Collector (UDC). Data collected with Mylyn and UDC was still not sufficiently fine-grained to answer many research questions, leading to the development of new data collection platforms [7], [12].

Early literature was focused on analysis of software versioning systems such as SVN or Git; however, they are in essence textual tools that record no semantic or syntactic meaning. Changes to software systems operate directly on abstract syntax tree (AST) entities [4].

Literature reviews on plagiarism detection [13]–[17] attempt to classify plagiarism detection methods and algorithms. Commonly algorithms are compared based on the type of source code analysis performed and retrieval methods.

The most popular plagiarism detection tools treat source code as simple text and try to find longest matching substring using algorithms such as Running Karp-Rabin Matching and Greedy String Tiling (RKR-GST). Trivial source code manipulations that students use to avoid detection are eliminated through preprocessing, which often encodes years of teaching experience and knowledge of typical transformations used by cheaters [1], [14]. Real-world performance of these tools is usually sufficient, however, even the most efficient algorithms have an exponential time complexity in relation to the number of compared documents, and thus research into more efficient methods is justified [18].

Most other methods can be classified into two groups, often named structure-based and attribute-based methods [19]. Attribute-based methods are based on extracting a set of numerical features (called *fingerprints* or *metrics*) from source code and comparing them using a distance function (e.g. [20]) or machine learning (e.g. [21]). Optimal selection of features is discussed in a number of papers, however the performance results are often weak.

Structure-based methods attempt to encode relationship between various programming elements, often using *abstract syntax tree (AST)* from code and finding subtree or substring matches (e.g. [22]), or performing an alignment (e.g. [23]). MOSS tool [3] extracts fingerprints from code using hashed ''engrams'' of code, thus encoding a fine structure that can usually be obtained only from comparing code strings directly. Several recent papers [18], [24] use information retrieval techniques (i.e. feature vector extraction and comparison) on syntax trees.

Some authors argue that neither an attribute-based nor a structure-based approach is sufficient on its own, since each is effective to avoid different types of source code manipulations, and therefore results from both should be combined and presented to human supervisor for final decision [23].

Papers on plagiarism cited above focus on the final result of programmers work i.e. program code. It is possible to obtain further information from the process of creation of this code which could be useful for plagiarism detection. For example, plagiarism detection for in-class assignments can be improved by providing hints in the form of seating plan and order in which the code is authored [25]. Similar to this paper, snapshots of student code are taken periodically. It is assumed that, at a certain point in time, the code between two students who plagiarise would be highly similar, and then diverge over time as the students attempt to obfuscate their plagiarism.

Vamplew and Dermoudy [26] propose an Anti-Plagiarism Editor which prohibits copy & paste operation from outside the editor itself. In addition, a list of timestamps is generated for each time the file was saved. Comparing these lists can hint towards plagiarism. Tahaei and Noelle [27] describe a programming course during which students are allowed to submit their solutions to each assignment multiple times before the deadline. These individual submissions represent a form of coarse-grained repository. Metrics such as number of submissions and Levenshtein distance were successfully used in plagiarism detection.

The most similar work to the one presented in this paper is [5], which uses the Flourite Eclipse extension [12] to detect fine-grained changes. Flourite logs are then processed to create histograms of different change types. Instances of plagiarism are detected by a combination of maximum correlation between two logs (using Pearson correlation coefficient on a subset of events), which suggests that both are instances of plagiarism, and by detecting outliers (students having unusual distributions of histogram values).

Another commonly discussed topic in literature is code clone detection. Code clone is defined as a fragment of code that is repeated in the same file or across several files [28], [29]. Detecting such clones is a useful task in the process of code quality improvement. While clone detection uses similar methods and algorithms to plagiarism detection, there are differences that require the use of specialized tools. In plagiarism detection, more preprocessing is typically performed, especially transformations that are specific to plagiarism and arise from teachers' experience. Plagiarism detection tools compare whole files and do not look for similarities within the same file. Additionally, plagiarism detection tools output pairs of similar homeworks with a similarity score (usually a percentage) for human review. Output from a clone detection tool would have to be extensively preprocessed before using in plagiarism detection setup.

III. RESEARCH METHODOLOGY

How to evaluate a plagiarism detection tool? One approach would be to observe classification error or accuracy (see e.g. [15]).

Most plagiarism detection tools give output in the form of an ordered list of document pairs with their similarities (in percents). So, to obtain a binary classification one must use a threshold. The issue of what threshold to use is rarely discussed in the literature.

While experimenting with tools based on substring matching found in literature [1], [2], an attempt was made to determine an optimal threshold for each collection with a view that there should be some relationship between similarity threshold and average document length or complexity. The conclusion was that optimal threshold shows a very large variance and that no such relationship could be established.

Typically, plagiarism detection tools are used such that a human supervisor scrolls trough the sorted list of documents and reviews each result, similar in a way to browsing through search results. During such use, false positive results are easily noticed and discarded, while real cases of plagiarism that are given low similarity (false negatives) are a far greater problem since a human supervisor will eventually give up after several pages of results.

This suggests that metrics known from information retrieval would be better suited for comparison. Reference [30] proposes Mean average precision (MAP) as the typical metric used in evaluation of IR algorithms. However, MAP is better suited for IR applications where false positive and false negative results are equally bad outcomes. Authors believe that a greater penalty should be assigned for false negative classification, as false positive results can be easily eliminated by the human supervisor (as discussed above).

Precision is defined as the number of correctly retrieved documents (true positives) among the top n results:

$$P_n = \frac{TP}{n}$$

while *recall* is the ratio of correctly retrieved documents (among the top n) to the total number of relevant documents in the dataset:

$$R_n = \frac{TP}{P}$$

The remaining issue is choice of n. Document collections in dataset vary greatly both in the total number of documents and in the number of known instances of plagiarism. To account for that, we choose n to be the number of documents that are known to be plagiarized ($n = P$), such that a tool that correctly ranks all plagiarized documents before those non-plagiarized would have a recall of 1, while a null-algorithm described earlier would have recall of 0. Thus, the chosen metric represents both precision and recall.

A shortcoming of precision-recall as a comparison metric is that it cannot be used on collections in which there are no known instances of plagiarism.

IV. DESCRIPTION OF DATASET

Given the discussion on homework plagiarism in the 21st century provided in the Introduction, dataset construction is a challenging problem. Several papers use the output from existing known tools as a "jury" for measuring new proposed algorithms [14], [31]. However, such approach was not applicable in this paper since existing tools based on code similarity are unable to detect cases of external plagiarism such as using "rent-a-coder" websites. Another common approach is constructing artificial datasets that reflect various realistic situations [15], [16], [30]. Overall, a lack of standard, openly discussed datasets is detected in literature [30].

The new dataset presented here is constructed from homeworks submitted by students during a one-semester university module on introductory programming in C. All final source codes submitted by students are available for comparison using traditional plagiarism detection tools. In addition, starting from 2015, students are required to use a cloud-based IDE to solve their homework, which allows to construct ultra-fine-grained repositories for further analysis.

Some statistics on the dataset are given in Table 1. Unlike some other similar courses, homework in this course consists of a large number (15-20) of relatively simple assignments. For some of these assignments, there are not many different ways to solve the problem correctly, and so existing plagiarism detection tools tend to report a very large number of false positive results. Anonymized dataset is available as open access dataset [32].

TABLE 1. Some statistics for the dataset.

Course	A2016	A2017
Students enrolled	607	488
Number of assignments	18	20
Submitted solution files	5655	5733
Files per assignment	41-503	125-444
- Average	314.17	286.65
Average file size (bytes)	1567.08	1317.23
Changes per file	1-13821	1-7740
- Average	710.57	675.11
Plagiarised solutions	746 (13.2%)	699 (12.2%)

Homework participation ranges from cca. 90% in early homeworks, but towards the end of semester it drops as low as 10% or less. There are several reasons for this. Homework assignments are fairly difficult, and it is possible to pass the course and even obtain a good grade without doing them. Also, most students at this school are not interested in achieving a high GPA due to lack of good scholarship options. Long term trend points towards increased participation and decreased plagiarism, suggesting that the applied measures were effective.

The method to fight plagiarism used to date was to require up to 20% of students to deliver an oral defense of their homework. The choice of students for this oral defense is based on code similarity, but also on other criteria such as suspicious behaviour (copy-pasting homework, suddenly solving homework at the last minute, etc.) and also past success on the course. Some false positive and false negative results were manually removed and added. A tool developed in-house was used for detecting source code similarity, but results were later reevaluated using Sim and JPlag.

The ground truth file was constructed as follows: Student who failed to defend their homework were marked as plagiarised in the ground truth file, as well as students with extremely high degree of similarity (99% or greater). In addition, when two or more homeworks are found to be similar, it is expected that one of those students will succeed in their oral defense (the original author who shared their homework). Most plagiarism tools can't distinct between original and copy, so this original likewise remained in the "ground truth" file.

Further, instances of very short (unfinished) homeworks that were nonetheless submitted to the grading system were removed entirely from the dataset. Even though proper classification of such homeworks is certainly a useful feature, there is a large difference in how various tools handle such cases: some allow to specify a threshold for homework length, other tools use some undisclosed heuristic, and some tools will simply mark all such homeworks as plagiarized and let a human supervisor unmark them.

This basic ground truth represents a sufficiently hard task for plagiarism detection.

To avoid overfitting, the dataset is divided into a training set and a test set. Further, two additional versions of ground truth were created for training static and dynamic detection methods respectively. Static ground truth includes only homeworks that are known to be similar, while dynamic ground truth excludes "original authors" of such homeworks but keeps those who have no similar pair but failed to defend their homework.

The structure of the ground truth file is such that some homeworks are grouped in similar pairs (triplets, quadruplets, etc.), while others are listed alone. Therefore, when evaluating a plagiarism detection tool, we do not verify if such tool correctly identified pairs of similar documents but simply count false positive and false negative results including both sides of their detected pairs.

V. PROPOSED ALGORITHM

A. FEATURE EXTRACTION

The choice of features that are extracted from student homeworks is a difficult task. Common approach is to construct a feature vector from source code metrics and compare it using standard distance functions from information retrieval. If two programs have a small distance, this is suspected to be a case of plagiarism.

A number of authors have conceded that, even after carefully selecting features for comparison, the plagiarism detection performance is still inferior to substring matching approach (for a recent example, see [30]). In previous paper [33], authors have argued that different features have varying contribution to the likelihood that homework is plagiarised, even after normalization, and proceed to use a Modified version of Weighted Euclidean distance (MWED) with weights optimized using Genetic Algorithm (GA). This gave a marked improvement over other metric-based approaches, but still inferior to RKR-GST approach.

The authors' research proceeds by extracting a second type of feature vector. In addition to the *static feature vector* (extracted from source code directly), another vector called a *dynamic feature vector* is extracted from a fine-grained repository. Feature vectors were then compared using MWED.

Static features are sufficiently described in previous paper [33]. On the other hand, *dynamic features* are extracted directly from the repository, and they are describing developers' behaviour and habits. For example, how often is the developer compiling code? How often does compiling end with errors? Is compilation triggered after every code change (even the minor ones)? Does the developer use copy-paste, and how often?

These features can be extracted from the repository, and they are dynamic in the sense that they reflect how things are done and not the final result (source code). For example, if a developer has plagiarized code by pasting it, then dynamic features tell us that there is one big paste, possibly only one compile try (and successful), and maybe one unit test run. The final code is the same as the original one (static), but developer profiles (their dynamic features) are different.

So, the idea is to group similar dynamic features (like one large paste followed by one successful compilation), and if such a group tends to appear in plagiarized homeworks, any homework with this group of features could be detected as plagiarized.

All dynamic features are presented in Table 2. Due to scarcity of related work, the choice of dynamic features is presented in this paper for the first time. In particular, features extracted from coarse-grained repositories (such as [27]) were not found to be useful on the evaluated dataset.

1) PASTE FEATURES

This group consists of features related to code paste. Paste is defined as simultaneous insertion of more than 5 effective lines of code. Through log analysis it was found that sometimes, 2-3 lines of code can suddenly appear for legitimate reasons, such as network lag and usage of "code snippet" feature of IDE. Allowing for some gray area, the most efficient approach was to specify a limit of 5 lines beyond which code insertion is described as code paste.

"Effective lines" excludes empty lines, lines with just white-space and lines with only curly braces. Also, multiple lines can be added or removed by applying code refactoring tools within IDE, and they too are excluded by some preprocessing.

TABLE 2. Dynamic features.

Feature	Description
ADDED	Lines added (how many lines are added)
DELETED	Lines deleted (how many lines are deleted)
MODIFIED	Lines modified (how many lines are modified and how many times modification occurred)
AVG PASTE	Average length of paste
MAX PASTE	Maximum paste length (if similar to code size, then this activity is very suspicious)
PASTES	Number of pastes
COMP	Number of total compilations
COMP S	Number of successful compilations
TEST RUNS	Number of unit test runs
AVG TEST	Average test score (score is calculated as number of test passed / total number of tests)
CPS	Characters per second (coding speed, rarely changes for one developer especially for tasks in same assignment)
TIME	Time spent for effective coding (with smart ignore pause events, pastes, write-and-delete, etc.)
SM BREAKS	Number of small breaks (when coding stops for small amount of time, 300 seconds – 5 minutes)
LO BREAKS	Number of long breaks, paused coding for 900 seconds (15 minutes) or more

The `PASTES` feature simply counts the number of paste events.

The `MAX PASTE` feature is calculated as the maximum paste length in characters (not lines). The `AVG PASTE` feature is calculated as the sum of characters in all pastes divided by the number of pastes. These features can be further analyzed. If `AVG PASTE` is similar to `MAX PASTE`, but there are few pastes (not one), that is an alert for suspicious code.

2) TEST FEATURES

This group of features is related to unit testing of written code. Every task has five or more unit tests, although the number of unit tests is very often larger than 10, and sometimes even larger than 30.

A unit test can fail for various reasons:

- Program does not compile,
- Program compiles but outputs wrong results,
- Program outputs correct results, but ends with crash,
- There are memory leaks and
- There is memory violation (reading uninitialized variables, accessing out-of-scope array elements...).

So, for every task there is a number of unit tests that can be run unlimited times. The number of unit test runs is `TEST RUNS`. This feature is very important. For example, if only one test run occurred, that probably means that all tests passed (otherwise, the developer will try to repair errors in the code and re-run tests). If all test pass in the first attempt (one test run), that indicates a perfect program (and perfect developer), so this is an alert for suspicious activity.

It is deduced (by watching multiple test runs) that the number of test runs heavily depends on the developer profile. Some developers tend to run tests with every code change (and watch how the test results are changing), and others tends to test code by themselves before hitting that "Run tests" button. It is possible to describe other behaviours, such as copying one unit test into source code directly, and then debugging that test by running the program (in that way, tests are not run with every change).

On the other hand, it can be very useful to record the success ratio of test runs. For example, if the first run passes 10 tests of 14, then the success ratio is 10/14; the second test passes 12/14, that means that the source code is improving. If the success ratio is declining (e.g. 7/14), that is a sign of heavy code refactoring, developer frustration, etc. The `AVG TEST` feature represents the average success ratio of all test runs (e.g in the case above it is $(10/14 + 12/14 + 7/14)/3 = 0.69$).

3) HITS-OF-CODE FEATURES

These features represent the developers' activity. We call them "hits of code" because they indicate code modification. Adding, deleting and modifying lines means that the developer is "hitting" them, making effort and spending time to write or delete them.

A common debate among authors on this subject, with too many examples to cite, is how to treat text with no semantic value in a given programming language, such as comments, whitespace, common "boilerplate" code (preprocessing directives etc.) However, the particular cloud IDE used in this research allows students to quickly reformat code with a single key combination i.e. to K&R style, OTB style etc. Furthermore, as user enters code, the IDE attempts to add indentation and move braces according to users selected style, which sometimes leads to contradictory results. Another IDE feature is "refactoring" which allows to quickly rename a variable or function in all places where it is used. The consequence of this behavior is that all code changes that are presumed to have come from IDE, such as reformat events, moving a brace up or down etc. are ignored.

Adding lines means that the code is being developed. Counting of added lines is not as straightforward as it may seem. All code refactoring tools add or remove lines, and code styling tools do the same. Some developers write the opening curly brace on the same line but others write them on a new line, etc. All of this has to be considered before counting a line as an added line. Some preprocessing is done, ignoring all of these irrelevant additions. The count of added lines represents the `ADDED` feature.

Removing or deleting lines means refactoring or deleting unused code, but can also mean intended code breaking in a way that genuine code passes all or almost all tests but plagiarized code passes some amount of tests (say, 80%) where the plagiarist is satisfied with 80% of credits (grade, etc.) without actually doing the task. Counting removed lines also requires the same level of preprocessing as counting added lines. The `DELETED` feature represents the number of effectively deleted lines. Some developers delete lines character by character (using backspace), some selecting a line and then deleting, and some by selecting multiple lines. On the other hand, a minority just comment lines and deletes them later (or even never). The strategy for removing lines is somewhat unique to each developer's profile.

The most challenging feature to calculate in this group is `MODIFIED`. It represents line modification, but as the repository records changes in added-deleted fashion (similar to other repositories), some additional work is required. First, repositories store modifications as a deletion of one line and addition of another, even in case of minor changes (e.g. changing variable name from `a` to `b`). So, these added and deleted lines that appear within very small intervals and that are consecutive in the change log are considered as modifications. Preprocessing has to be done as in cases of line addition and deletion. This feature is the "real" hit-of-code feature, and it represents how much time and effort the developer spent on one line, and if that amount is somewhat large, that means that the line modified is very important. This is also similar to the "characters in diff" and "modifies" features described in [6].

4) COMPILATION FEATURES

This group consists of two features: `COMP` and `COMP S`. The first feature represents the total number of compile attempts, and the second one represents the number of successful compilations (with no compile error).

As with the unit tests, some developers tend to compile their program with every minor code change, whereas some prefer to write the code and then compile. An interesting fact is that the first group of developers has a subgroup, those who try to compile and if the compilation is not successful with multiple errors, they fix the first error and try to compile again. The second subgroup are developers who try to repair as many compile errors as they can between compilation tries. This is not the same as fixing code errors (e.g. wrong output, program crash, etc.), because, for example, incorrect output can be fixed with multiple modifications and compilations, but multiple compile errors, on the other hand, can usually be fixed all at once (instead of fixing one at a time, as in the first subgroup).

The number of compile fails can be easily calculated as a difference of these two features, but it is not included as a feature because it is linearly dependent on the other two and will be entirely redundant.

5) TIME FEATURES

This group has very important features for plagiarism detection and developer profiling.

The first feature, `TIME`, represents the overall amount of time the developer spent on a specific task. Time is smartly and accurately calculated, and this feature can not be easily "fooled".

Events that happen within 15 seconds or less are considered as consecutive, and the time is calculated between first and last such event. However, if the delay is longer than 15 seconds, time counting stops. In this way, the TIME feature doesn't increase when the developer is running programs, compiling, or unit testing (these actions are represented by separate features), nor is it counted when the developer takes a break, or writes and deletes the same line repeatedly. The constant of 15 seconds was determined through observation of actual programmer behaviours in comparison to log entries.

Some plagiarists try to bypass the paste detection feature and plagiarize code with the help of third-party software that "types" code for them. This is a type of software that receives source code (possibly the original) as input, and writes that code to a specific editor (with currently active focus), character by character. The plagiarist sets up a "coding interval" (e.g. 0.8 seconds), and leaves the software to re-write code to the IDE that records every key-stroke. Such code will have `TIME` feature that is highly proportional to code length and typing speed.

To catch this form of cheating, a new feature is introduced, `CPS` – characters per second. Every micro-commit is recorded, time difference is calculated and then CPS is calculated. The `CPS` feature represents the average typing speed, but this approach can be used to detect cheating attempts. If typing speed has very low variation between micro-commits, that indicates that a "robot" is typing. Human typing speed will typically vary within small amount. Every letter cannot be written and reached at the same time, plus the developer needs time to think what to code. So, "robot-typing" plagiarism is detected as a feature of typing speed. Also, with two other features, long breaks and small breaks, this can be checked because humans take breaks while coding, whereas software does not.

As previously mentioned, developers need a break; to drink a cup of coffee, to think about code or a problem, to sketch program flow on paper. So, as a feature, the number of breaks is used. But it can be useful to separate small breaks (mentioned above) and longer breaks (e.g. sleep). The `SM BREAKS` feature represents the number of small breaks (5 minutes or less). Similarly, the `LO BREAKS` feature represents the number of long breaks (15 minutes or more).

Correlation of typing speed, time spent, and number of breaks can be made with individual developers. Some tend to complete a task in single try, while others complete the task within days. Some are typing and thinking very fast, and other do not.

These features should not vary for one developer within one larger task (e.g. homework assignment). If a developer has a typing speed of 2 CPS in one task, the same developer cannot have typing speed of 5 CPS (or 0.7 CPS) in another one. Typing speed can increase, but not in such a small time frame. It is very suspicious when these features vary between consecutive tasks.

B. USING NEURAL NETWORKS TO IMPROVE PLAGIARISM DETECTION

Supervised learning is essential when using an IR approach in plagiarism detection [18]. In this paper, a simple back-propagation neural network is trained using only dynamic features described in chapter II, thus creating a binary classifier that outputs probabilities in the range [0, 1] for class 0 (not plagiarized) and class 1 (plagiarized). A threshold value k_1 is applied, and all documents for which artificial neural network (ANN) output is greater are marked as plagiarized without further processing.

After this initial filtering step, a second ANN trained in the same way is used to improve similarity ranking obtained from a reference plagiarism detection tool based on substring matching [2]. A weighted sum is used to produce such improved similarity value P:

$$P = P_{nn} \cdot k_2 + P_{static} \cdot (1 - k_2)$$

where P_{nn} is ANN output and P_{static} is the output from the reference plagiarism detection tool.

Both ANNs were built using the Fast Artificial Neural Network (FANN) library [34]. This library uses by default a modified version of the RPROP training algorithm called iRPROP [35], which does not require setting a learning rate since it is determined automatically.

Other hyper-parameters of ANNs as well as values for k_1 and k_2 were determined using cross-validation. The dataset was split into training, validation, and testing subsets, with sizes roughly in ratio 50:25:25 (Fig. 1). Firstly, cross-validation is performed on the second (similarity improving) ANN, then on the first (filtering) ANN. Each time, the validation subset was used to terminate training as soon as highest recall was obtained. Recall as performance metric chosen in this paper caused the results to have a high degree of variation between runs, so the process is repeated 10 times, and the ANNs with best performance are ultimately chosen. Dynamic ground truth was used both for training and cross-validation.

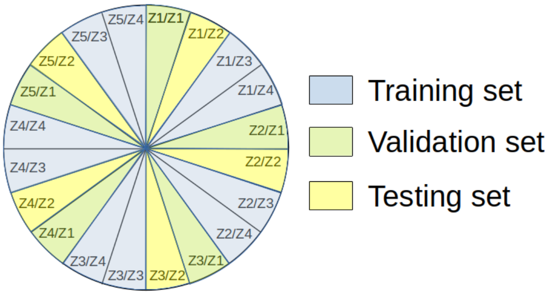

FIGURE 1. Organization of A2017 dataset by assignment (homework) into training, validatoin and testing set. Z1/Z1, Z1/Z2 and so on are labels for individual assignments (homeworks).

The best results (highest recall) were obtained with the following configuration: for filtering ANN, a single hidden layer with 192 neurons, sigmoid symmetric (tansig) activation function, and $k_1 = 0.79$. For similarity improving ANN, a single hidden layer with 64 neurons, sigmoid (logsig) activation function, and $k_2 = 0.36$.

Note that the neural network described in this paper attempts to model student behaviour as they solve their homework. This behaviour is likely to change gradually over time, in part in response to plagiarism detection. Therefore it is advisable to retrain the network periodically with latest data.

VI. COMPARISON WITH OTHER PLAGIARISM DETECTION TOOLS

A. RESULTS FIDELITY

Plagiarism detection results obtained with the described setup (labeled Sim+NN) were compared to reference tools from literature: JPlag [1], Sim [2], and MOSS [3], as well as an a tool developed by authors (labeled *fvpd*) based on static feature vectors extracted from code [33].

The full ground truth file is used to obtain recall and accuracy for each of the evaluated tools. Accuracy is calculated at such threshold that gives minimal classification error for each homework separately. Results are presented in Table 3.

TABLE 3. Mean recall and accuracy for all assignments in test set.

Plagiarism detection tool	Recall	Accuracy
fvpd	0.3805	0.8194
JPlag	0.4935	0.8202
MOSS	0.5452	0.8256
Sim	0.6411	0.8390
Sim+NN	0.6875	0.8728

It is clear from these results that adding neural network output to the output obtained from Sim resulted in significant improvement in both accuracy and recall. Further breakdown of recall per individual assignment can be seen in Fig. 2.

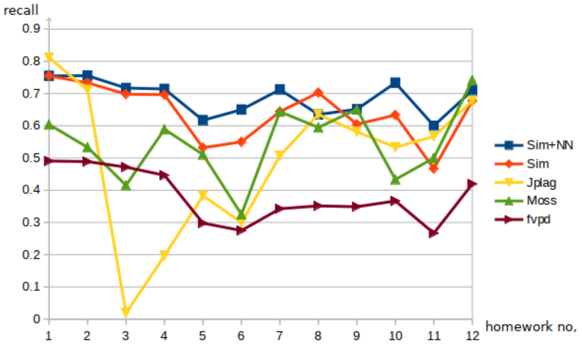

FIGURE 2. Comparison of recall for evaluated tools. Results for the tool presented in this paper are labeled Sim+NN and are given in blue.

It should be noted that JPlag gives exceptionally poor results in some of the early homeworks due to a very large number of false positive results. For example, for homework labeled 3 on Fig. 2, it reported approximately 150 pairs marked as 100% similar. JPlag has a tunable "sensitivity" parameter, but changing its value did not improve the results. The authors suspect that the preprocessing performed by JPlag is too extensive, as for some of the more elaborate homeworks it gives much better results but broadly on par with Sim. The use of neural networks consistently managed to improve Sim results for each homework.

A weakness of the proposed method is wrong classification (false positive) of exceptionally good programmers who managed to solve the assignments on the first try, with very little testing. This effect can be seen in homework 8 in Fig. 2. Therefore, in practical application of the proposed method, it is critical that the cutoff coefficient k_1 is not set too low and that all students marked as plagiarized in this way are allowed an oral defense.

B. TIME AND MEMORY PERFORMANCE

Tools based on substring matching have, in general, an exponential time complexity. A naïve implementation would have an $O(N^2n^3)$ complexity [14], where N is the total number of documents and n is their length in characters. However, clever programming and various tricks allow the tools presented here to perform very well in most realistic usage cases. The worst-case complexity of $O(N^2n^{1,12})$ has been claimed [14], [36].

The system presented here uses results obtained from one such tool (Sim) that has exceptionally good performance and further improves those results using a pre-trained ANN. Improvement consists of two stages: extracting features from code has time complexity of $O(Nn)$ where n is the average size of logs, while N is the total number of documents as before; testing those features against ANN has time complexity that is proportional only to the number of neurons, so here it can be assumed to be $O(N)$.

Using an efficient implementation of the system presented in this paper, time complexity should be dominated by the substring matching stage. Thus, for a sufficiently large corpus of homeworks, execution time for the described system should not be significantly greater than using substring matching alone.

Experiments indicate that, using the fann library, none of the ANNs that were built during the research presented here have used more than several hundred kB of RAM. An efficient feature extraction algorithm can be used that does not store repository data in RAM; thus, authors believe that this solution is highly memory efficient, as well.

VII. CONCLUSIONS AND FUTURE WORK

This paper presents a novel type of plagiarism detection system that uses artificial neural networks to improve upon the results obtained from classical tools based on substring matching.

The system uses an ultra-fine-grained code repository with a very high resolution of recorded changes. Such repository can be used to extract detailed information on each developer useful for building a developer profile.

A set of features that are descriptive of the types of behaviours that indicate plagiarism was extracted from dataset. Two neural networks were trained to classify all submitted homeworks as either plagiarized or non-plagiarized. First of those networks was used to filter those homeworks that strongly indicate plagiarism, while the other one was used to improve the plagiarism detection in remaining homeworks based on a similarity score using a weighted sum.

Results obtained show a significant improvement in both accuracy and recall of plagiarism detection. Therefore, we can conclude that there is a detectable difference in IDE usage patterns between students who plagiarise their homework and those who work on their own, and thus the authors' hypothesis is confirmed.

This is, of course, just an example of possible information that can be discovered from such repositories. In the future, the authors hope to use ultra-fine-grained repositories to discover and help students with difficulties in learning programming. Additionally, it could be possible to develop an intelligent assistant integrated in the cloud-based IDE, that detects certain typical mistakes and warns students to fix them. Such assistant can be trained on data from generations of students that is preserved in the authors' repository.

ANN might not be the optimal classification algorithm for this problem. A particular problem with ANNs is that it can be difficult to interpret their results [37], which in plagiarism detection is a required feature. Authors intend to perform further experiments using decision trees, random forest and naïve Bayes algorithms. Also, recent research in the area of visualisation and analysis of ANNs will be incorporated into an ''explain results'' feature.

The approach presented in this paper is to improve results obtained using Sim – a substring matching tool – using ANN. While such tools overall have a very good classification fidelity, the execution time increases exponentially with the number of processed documents, limiting their use in courses with very large number of participants such as MOOC. Authors hope to further improve performance of the fvpd tool which is based on feature extraction, and thus potentially scales far better, but in the research so far [33] it gave below average results (Fig. 2). ANN can be used to achieve this goal while maintaining overall O(nN) complexity.

REFERENCES

[1] L. Prechelt and G. Malpohl, ''Finding plagiarisms among a set of programs with JPlag,'' *J. Universal Comput. Sci.*, vol. 8, no. 11, pp. 1016–1038, Mar. 2003.

[2] D. Grune and M. Huntjens, ''Detecting copied submissions in computer science workshops,'' Inf. Faculteit Wiskunde Informatica, Vrije Universiteit, Amsterdam, The Netherlands, Tech. Rep., 1989. [Online]. Available: http://www.dickgrune.com/Programs/similarity_tester/Paper.ps

[3] S. Schleimer, D. S. Wilkerson, and A. Aiken, ''Winnowing: Local algorithms for document fingerprinting,'' in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2003, pp. 76–85.

[4] Q. D. Soetens, R. Robbes, and S. Demeyer, ''Changes as first-class citizens: A research perspective on modern software tooling,'' *ACM Comput. Surveys*, vol. 50, no. 2, pp. 1–38, Jun. 2017, doi: 10.1145/3038926.

[5] J. Schneider, A. Bernstein, J. V. Brocke, K. Damevski, and D. C. Shepherd, ''Detecting plagiarism based on the creation process,'' *IEEE Trans. Learn. Technol.*, vol. 11, no. 3, pp. 348–361, Jul. 2018.

[6] K. Mierle, K. Laven, S. Roweis, and G. Wilson, ''Mining student CVS repositories for performance indicators,'' *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, Jul. 2005.

[7] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig, ''Is it dangerous to use version control histories to study source code evolution?'' in *Proc. 26th Eur. Conf. Object-Oriented Program. (ECOOP)*, 2012, pp. 79–103

[8] E. Giger, M. Pinzger, and H. C. Gall, ''Comparing fine-grained source code changes and code churn for bug prediction,'' in *Proc. 8th Work. Conf. Mining Softw. Repositories*, 2011, pp. 83–92.

[9] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, ''Mining fine-grained code changes to detect unknown change patterns,'' in *Proc. 36th Int. Conf. Softw. Eng. (ICSE)*, 2014, pp. 803–813.

[10] W. Maalej, T. Fritz, and R. Robbes, ''Collecting and processing interaction data for recommendation systems,'' in *Recommendation Systems in Software Engineering*. Berlin, Germany: Springer, 2014, pp. 173–197.

[11] R. Robbes, D. Pollet, and M. Lanza, ''Replaying IDE interactions to evaluate and improve change prediction approaches,'' in *Proc. 7th IEEE Work. Conf. Mining Softw. Repositories (MSR)*, May 2010, pp. 161–170.

[12] Y. Yoon and B. A. Myers, ''Capturing and analyzing low-level events from the code editor,'' in *Proc. 3rd ACM SIGPLAN Workshop Eval. Usability Program. Lang. Tools*, 2011, pp. 25–30.

[13] J. Hage, P. Rademaker, and N. van Vugt, ''A comparison of plagiarism detection tools,'' Utrecht Univ., Utrecht, The Netherlands, Tech. Rep. UU-CS-2010-015, 2010.

[14] M. Mozgovoy, ''Enhancing computer-aided plagiarism detection,'' Ph.D. dissertation, Univ. Joensuu, Joensuu, Kuopio, 2007.

[15] S. Burrows, ''Source code authorship attribution,'' Ph.D. dissertation, RMIT Univ., Melbourne, VIC, Australia, 2010.

[16] V. T. Martins, D. Fonte, P. R. Henriques, and D. da Cruz, ''Plagiarism detection: A tool survey and comparison,'' in *Proc. 3rd Symp. Lang., Appl. Technol. (SLATE)*, vol. 38, Braganáa, Portugal, 2014, pp. 143–158.

[17] M. Agrawal and D. K. Sharma, ''A state of art on source code plagiarism detection,'' in *Proc. 2nd Int. Conf. Next Gener. Comput. Technol. (NGCT)*, Oct. 2016, pp. 236–241.

[18] D. Ganguly, G. J. F. Jones, A. Ramírez-de-la-Cruz, G. Ramírez-de-la-Rosa, and E. Villatoro-Tello, ''Retrieving and classifying instances of source code plagiarism,'' *Inf. Retr. J.*, vol. 21, no. 1, pp. 1–23, Feb. 2018.

[19] X. Chen, B. Francia, M. Li, B. McKinnon, and A. Seker, ''Shared information and program plagiarism detection,'' *IEEE Trans. Inf. Theory*, vol. 50, no. 7, pp. 1545–1551, Jul. 2004.

[20] J. A. W. Faidhi and S. K. Robinson, ''An empirical approach for detecting program similarity and plagiarism within a university programming environment,'' *Comput. Edu.*, vol. 11, no. 1, pp. 11–19, Jan. 1987.

[21] S. Engels, V. Lakshmanan, and M. Craig, ''Plagiarism detection using feature-based neural networks,'' *ACM SIGCSE Bull.*, vol. 39, no. 1, pp. 34–38, Mar. 2007.

[22] D. Gitchell and N. Tran, ''Sim: A utility for detecting similarity in computer programs,'' *ACM SIGCSE Bull.*, vol. 31, no. 1, pp. 266–270, Mar. 1999.

[23] M. El Bachir Menai and N. S. Al-Hassoun, ''Similarity detection in java programming assignments,'' in *Proc. 5th Int. Conf. Comput. Sci. Edu.*, Aug. 2010, pp. 356–361.

[24] O. Karnalim and Simon, ''Syntax trees and information retrieval to improve code similarity detection,'' in *Proc. 32nd Australas. Comput. Edu. Conf.*, Feb. 2020, pp. 48–55.

[25] A. Budiman and O. Karnalim, ''Automated hints generation for investigating source code plagiarism and identifying the culprits on in-class individual programming assessment,'' *Computers*, vol. 8, no. 1, p. 11, 2019.

[26] P. Vamplew and J. Dermoudy, ''An anti-plagiarism editor for software development courses,'' in *Proc. 7th Australas. Conf. Comput. Educ.*, 2005, pp. 83–90.

[27] N. Tahaei and D. C. Noelle, ''Automated plagiarism detection for computer programming exercises based on patterns of resubmission,'' in *Proc. ACM Conf. Int. Comput. Edu. Res.*, Aug. 2018, pp. 178–186.

[28] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, ''Comparison and evaluation of clone detection tools,'' *IEEE Trans. Softw. Eng.*, vol. 33, no. 9, pp. 577–591, Sep. 2007.

[29] C. K. Roy, J. R. Cordy, and R. Koschke, ''Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,'' *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, May 2009.

[30] O. Karnalim, S. Budi, H. Toba, and M. Joy, ''Source code plagiarism detection in academia with information retrieval: Dataset and the observation,'' *Informat. Edu.*, vol. 18, no. 2, pp. 321–344, 2019.

[31] Z. Duric and D. Gasevic, ''A source code similarity system for plagiarism detection,'' *Comput. J.*, vol. 56, no. 1, pp. 70–86, Jan. 2013.

[32] V. Ljubovic. (2020). *Programming Homework Dataset for Plagiarism Detection*. [Online]. Available: http://dx.doi.org/10.21227/71fw-ss32

[33] E. Pajic and V. Ljubovic, ''Improving plagiarism detection using genetic algorithm,'' in *Proc. 42nd Int. Conv. Inf. Commun. Technol., Electron. Microelectron. (MIPRO)*, May 2019, pp. 571–576.

[34] S. Nissen, "Implementation of a fast artificial neural network library (FANN)," Dept. Comput. Sci., Univ. of Copenhagen, København, Denmark, Tech. Rep., 2003.

[35] C. Igel and M. Hüsken, "Improving the Rprop learning algorithm," in *Proc. 2nd Int. Symp. Neural Comput.*, 2000, pp. 115–121.

[36] M. J. Wise, "Running Rabin-Karp matching and greedy string tiling," Basser Dept. Comput. Sci., Sydney, NSW, Australia, Tech. Rep., 1994.

[37] W. Samek, A. Binder, G. Montavon, S. Lapuschkin, and K.-R. Müller, "Evaluating the visualization of what a deep neural network has learned," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 28, no. 11, pp. 2660–2673, Nov. 2017.

VEDRAN LJUBOVIC was born in Sarajevo, Bosnia and Herzegovina, in 1978. He received the B.S. (Diploma Engineer) and M.S. degrees in computing and informatics from the Faculty of Electrical Engineering, University of Sarajevo, Bosnia and Herzegovina, in 2010, and the Ph.D. degree in technical sciences from the University of Sarajevo, in 2015.

From 2006 to 2015, he was a Teaching and Research Assistant with the Department for Computing and Informatics, University of Sarajevo. Since 2015, he has been an Assistant Professor with the Department for Computing and Informatics. His Ph.D. thesis and prior research were in the area of content-based image retrieval. In the recent years, his focus of interest is in applications of information retrieval techniques in source-code and repository analysis. He is also interested in applications of information technology in education. He is involved in several projects for development of educational information systems at his department. Also, he has authored a book and several papers at journals and conferences.

ENIL PAJIC was born in Konjic, Bosnia and Herzegovina, in 1993. He received the B.S. and M.S. degrees in computing and informatics from the Faculty of Electrical Engineering, University of Sarajevo, Bosnia and Herzegovina, in 2017.

During his studies, he was an Undergraduate Teaching Assistant at his department. Since graduation, he has employed as a Software Engineer at Monri Payments, Sarajevo.