# A Fuzzy-based Approach to Programming Language Independent Source-Code Plagiarism Detection

Giovanni Acampora and Georgina Cosma
School of Science and Technology
Nottingham Trent University
Nottingham, U.K. NG11 8NS
e-mail: (giovanni.acampora@ntu.ac.uk, georgina.cosma@ntu.ac.uk).

***Abstract*—Source-code plagiarism detection in programming, concerns the identification of source-code files that contain similar and/or identical source-code fragments. Fuzzy clustering approaches are a suitable solution to detecting source-code plagiarism due to their capability to capture the qualitative and semantic elements of similarity. This paper proposes a novel Fuzzy-based approach to source-code plagiarism detection, based on Fuzzy C-Means and the Adaptive-Neuro Fuzzy Inference System (ANFIS). In addition, performance of the proposed approach is compared to the Self- Organising Map (SOM) and the state-of-the-art plagiarism detection Running Karp-Rabin Greedy-String-Tiling (RKR-GST) algorithms. The advantages of the proposed approach are that it is programming language independent, and hence there is no need to develop any parsers or compilers in order for the fuzzy-based predictor to provide detection in different programming languages. The results demonstrate that the performance of the proposed fuzzy-based approach overcomes all other approaches on well-known source code datasets, and reveals promising results as an efficient and reliable approach to source-code plagiarism detection.**

## I. Introduction

Plagiarism of source-code is a growing problem due to the growth of source-code repositories, and digital documents found on the Internet. Plagiarism is considered as one of the most severe problems in academia, due to the availability of source-code found on-line and also due to sharing of source-code solutions among students. Essentially, plagiarism in computer programs occurs when a person reuses source-code authored by someone else and fails to acknowledge the author [1]. Plagiarism detection in software programs, concerns the identification of source-code files that contain similar and/or identical source-code fragments. When two programs have been written for solving the same problem in the same programming language, it is very likely that these source-code solutions will contain code which is similar. For this reason, ample consideration and scrutiny must take place prior to classifying two programs as similar. Importantly, within plagiarised files, similarity does not occur by coincidence; the similar source-code fragments must share source-code similarity which is significant enough to classify the two programs as similar [2]. This particularly applies to student solutions, where a programming problem is assigned to a class of students who are required to provide their solution written in a particular programming language. In essence, when a set programs are scrutinised for plagiarism, evidence must be based on the code which is unique in the files under scrutiny, and all the coincidental similarity among program solutions should only be complimentary to the evidence. Hence, the challenge in plagiarism detection systems is to retrieve files which have significant code, and not to overwhelmingly detect files which contain several small and insignificant fragments occurring in several files (this is considered as noise in the data), as this will add the extra burden of time on the academic, searching through a large number of files to detect the ones which contain proof of plagiarism. Similar source-code fragments which are common across many files, essentially add noise to the problem of plagiarism detection [2]. State-of-the art string matching plagiarism detection systems, are not capable of dealing with such noise and inevitably (and falsely) detect as similar a large number of files which share commonly occurring source-code fragments (or substring sequences). Source-code plagiarism detection using string-matching algorithms is based on structural characteristics (common substring sequences), and for this reason such algorithms fail to detect similar files that contain significant code shuffling, and are prone to the problem of *local confusion* [3]. String-matching based systems use parsers to convert source-code files into tokens and this makes them programming language-dependent. For example, string-matching algorithms developed for detecting plagiarism among Java files, cannot be used to detect plagiarism among programs written in the C++ programming language as they will not be able to be parsed. In addition, files which contain compilation errors and do not compile are excluded from the comparison process.

In order to address these issues, we are proposing a fuzzy-based algorithm suitable for plagiarism detection. The proposed algorithm uses the statistical approach of Singular Value Decomposition to remove noise from the source-code files, and thereafter uses computational intelligence approaches of FCM [4] and ANFIS [5] to predict the similar files. In particular, the comparison process commences by clustering files based on their similarity using the Fuzzy C-Means clustering approach, and thereafter, optimising the performance using the Adaptive Neuro-Fuzzy Inference System. The proposed algorithm, searches for significant similarity among files, and is capable of including all files in the comparison process since it does not rely on any parsers and files are not compiled. In addition, similarity is not computed on a pair-wise basis but rather files are clustered based on their degree of membership to the particular cluster of similar files. The performance of the proposed system is compared against the Fuzzy C-Means (FCM) and Self-Organising Map (SOM) [6] computational intelligence algorithms; and it is also compared against the state-of-the-art Running Karp-Rabin Greedy-String-Tiling (RKR-

GST) plagiarism detection algorithm implemented within the JPlag [7] plagiarism software. The paper is structured as follows: Section II provides a literature review on source-code plagiarism algorithms; Section III discusses the framework for the proposed Fuzzy Clustering System and its algorithms; Section IV describes the methodology for experimentation including the datasets. Section IV-A2 discusses the performance evaluation measures and how these have been adapted for the plagiarism detection problem when using a clustering approach; Section IV-B presents a comparative discussion of the performance of our proposed system and other clustering approaches; and finally V summarises the findings and future work.

## II. Literature Review on Source-Code Plagiarism Algorithms

This section summarises the different source-code plagiarism detection tools that exist within the literature with emphasis on the most recent approaches. A history and explanation of on early detection tools is provided by [8]. Most of the source-code plagiarism detection tools are based on string-matching algorithms. With string-matching based systems, the first stage is called tokenization, where each source-code file is replaced by predefined and consistent tokens, for example different types of loops in the source-code may be replaced by the same token name regardless of their loop type (e.g. while loop, for loop). Each source-code file is then represented as a series of token strings. The string of tokens found in file pairs are compared using a token matching algorithm to identify to determine matching substring sequences. Matches of substrings are called tiles, and tiles whose length is below a minimum-match length threshold are excluded from the comparison process. The Running Karp-Rabin Greedy-String-Tiling algorithm (RKR-GST) is a well-known token matching algorithm developed initially within the YAP3 plagiarism detection tool [9]. Several tools have been developed, which essentially propose approaches to tokenisation of programs written in various languages and then use the RKR-GST algorithm, or another substring matching algorithm, to compare the token string representations of programs. Among them, JPlag [7] is a well-known and widely used on-line source-code plagiarism detection tool for detecting programs written in the Java, C#, C, C++, and Scheme programming languages, as well as natural language text. JPlag [7] uses the same comparison algorithm as YAP3 [9], but with optimized run time efficiency. JPlag computes the similarity as the percentage of token strings covered. JPlag is implemented as a web service and has a simple but efficient user interface which comprises of two main pages: 1) a page listing the similar file pairs along with their degree of similarity; and 2) a facility for comparing the detected similar files by highlighting their matching blocks of source-code fragments. Kustanto and Liem [10] proposed a tool for detecting source-code plagiarism among programs written in the LISP and Pascal programming languages. Their approach is a token-based approach which essentially comprises of two steps: firstly, it parses the source code and transforms it into tokens, and thereafter compares each pair of token strings obtained in the first step using the RRKR-GST algorithm. More recently, Muddu et al. [11] propose a token representation approach for programs written in the Java programming language, and then use the RKRGST algorithm to detect code similarity. They compared their approach to other plagiarism detection tools, namely the Copy Paste Detector (CPD), Sherlock [12], CCFinder [13] and Plaggie [14] and found that their approach outperformed the other approaches. One of the problems encountered by all, including the abovementioned, token based string matching approaches, is that files must parse to be included in the comparison for plagiarism, and this can cause similar files that were not parsed to be missed. In addition, the parameter of minimum-match length is user-defined and changing this number can alter the detection results (for better or worse). Furthermore, to alter this number one may need an understanding of the RKR-GST algorithm. The approaches by Moussiades and Vakali [15] and Cosma and Joy [2] are based on information retrieval algorithms. Moussiades and Vakali [15] have developed a plagiarism detection system called PDetect which is based on the standard vector-based information retrieval technique. Initially, the PDetect tool represents each source-code program as an indexed set of keywords and their frequencies, and thereafter computes the pair-wise similarity between programs. The program pairs that have similarity greater than a given cut-off value are grouped into clusters. Their experiments revealed that PDetect and JPlag are sensitive to different types of attacks and the authors suggest that JPlag and PDetect complement each other. Cosma and Joy [2] proposed the PLAGATE algorithm, which combines the Latent Semantic Analysis [16], information retrieval techniques with string-matching in order to improve detection. Their results revealed that hybrid approaches returned better plagiarism detection results than when using LSA or string-matching approaches alone. Other hybrid approaches include that of Ajmal et al [17] who proposed a source-code plagiarism detection system which also utilises a greedy string tiling algorithm. In summary, their system initially selects a seed source code file (or the original file) from the dataset and detects the top k similar files using source code metrics techniques (such as McCabe's Cyclomatic Complexity, counts of logical, physical, comment, and blank lines; and counts of attributes and methods). In the second step, the detected files are compared to the seed file through a Greedy String Tiling algorithm. The files, whose contents match to the seed file with similarity more than the threshold value, are detected as similar. Kilgour et al. [18] combined structure metric techniques with Fuzzy logic. Initially, information about each source-code file was obtained using existing numerical software metrics (proportion of blank lines, proportion of lines that are or include comments, average length of identifiers, use of templates, statements per function/method, use of underscores in identifiers, use of capitalization in identifiers) and thereafter fuzzy-logic linguistic variables were used which capture more subjective elements of authorship, such as the degree to which comments match the actual source code's behaviour. The authors emphasize that metrics alone are not sufficient for detecting plagiarism due to the difficulty with capturing certain types of information and measuring their similarity. An exhaustive literature review revealed that Fuzzy approaches have not been applied to source-code plagiarism detection as much as structure metric and string matching approaches. However, Fuzzy logic approaches have been successfully applied to cluster source-code for the recovery of source-code design patterns [19], source-code mining [20], [21], to assess similarity within program samples [22], to derive rules in order to detect security defects in programs [23], to find traceability links between reports and source-code [24].

## III. A Fuzzy Clustering based Source-Code Plagiarism Detection System

This section introduces an innovative computational intelligence framework for the purpose of analysing source-code in the context of source-code plagiarism detection (see Fig. 1).

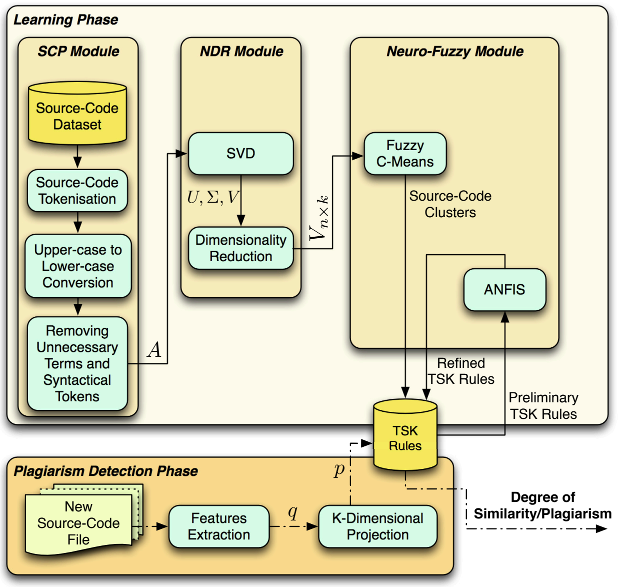


Fig. 1: The Architecture of the Fuzzy-based Source-Code Plagiarism Detection System

### A. Source-Code Pre-processing Module

The purpose of the Source-code Pre-processing (SCP) module is to pre-process the source-code files in such way that it removes unnecessary and meaningless terms and characters in order to reduce the size of the data to more efficiently capture the semantic representation of each source-code file. The first step towards SCP is called *tokenisation*, the lexical analysis process of breaking a stream of text up into words, called tokens. Thereafter the following pre-processing parameters are also applied: conversion of upper-case letters to lower case, removing terms that occur in one file because such terms hold no information about relationships among terms across files, removing terms solely composed of numeric characters, removing syntactical tokens (e.g. semi-colons, colons), removing terms consisting of a single letter. The following source-code specific pre-processing parameters are also applied: removal of comments, removal of language reserved terms, removal of obfuscation parameters found in terms (e.g. those found within terms, i.e student_name, _ is an obfuscation parameter); source-code identifiers containing multiple words are treated as a single identifier, i.e., obfuscators that join two words together are removed such that the two words are treated as a single word. Listing 1 and 2 show an example of what a source-code fragment looks like before and after applying pre-processing, respectively.

Listing 1: Example Java Source-Code before Pre-processing

```
public class ReadAndPrintNumbers{
public static void main(String[] args){
try{
Scanner s=new Scanner(new File(``num.dat''));
while(s.hasNextInt()){
System.out.println( s.nextInt() );}}
catch(IOException e){
System.out.println(e);}}
}
```

Listing 2: Example Java Source-Code after Pre-processing

```
public class readandprintnumbers
static void main string args
try scanner new scannernew file num dat
while hasnextint system out println
nextInt catch ioexception
```

Once pre-processing is applied, the SCP Module creates the *Vector Space Model* (VSM) representation of the source-code dataset. In the VSM represents a term-by-file matrix $A_{m \times n} = [a_{ij}]$, in which each row $i$ holds the frequency of refined dictionary terms (i.e. terms found in source-code files after pre-processing) across source-code files, and each column $j$ represents a source-code file. Hence, each cell $a_{ij}$ of A contains the frequency at which a dictionary term $i$ appears in a source-code file $j$. Thereafter, the SCP Module normalises the term frequency in matrix $A$ by applying a *global weighting* function to adjust the frequency of terms in respect to the entire collection of source-code files. In the same way, a *document length normalisation* is applied to tune the frequencies based on the length of each file. In particular, the SCP Module uses the *normal global weighting* function named $g_i$ and the *cosine document length normalisation* named $n_j$:

$$\begin{aligned} g_i &= \frac{1}{\sqrt{\sum_j a_{ij}^2}} \\ n_j &= (\textstyle\sum_i (g_i \cdot a_{ij})^2)^{-1/2} \end{aligned} \tag{1}$$

where $a_{ij} = A[i, j]$.

After the SCP Module computes the normalisation step, each entry of the matrix A is updated as follows:

$$a_{ij} = a_{ij} \times g_i \times n_j$$

with $i = 1, \ldots, m$ and $j = 1, \ldots, n$.

The role of normalisation enables the framework to capture information about the importance of each term in describing each source-code file.

### B. Noise and Dimensionality Reduction Module

The aim of the Noise and Dimensionality Reduction (NDR) Module is to further reduce the database size and hence space complexity by removing noise and irrelevant data. This task is accomplished by using the Singular Value Decomposition (SVD) and the Dimensionality Reduction Statistical techniques, both described in [25]. SVD is able to expose the underlying relationships among the original data items. SVD can be used for data reduction because it identifies and orders the dimensions of data based on variation which makes it a useful technique for tasks requiring representation of data in a reduced dimensional space. Using SVD, the best approximation of the original data points using fewer dimensions can be achieved.

Given a $m \times n$ matrix $A$, let $r$ be the rank of matrix $A$, the singular value decomposition of A is defined as

$$A_{m\times n} = U_{m\times r} \cdot \Sigma_{r\times r} \cdot V^T_{r\times n}$$

where $U$ is an $m \times r$ orthogonal matrix ($U^T U = I_r$) term-by-dimension matrix, whose columns hold the left singular vectors; $\Sigma$ is an $r \times r$ diagonal matrix containing the singular values of matrix A, ordered in decreasing order such that $\sigma_1 \geq \sigma_2 \geq \ldots \sigma_r > \sigma_r + 1 = \ldots = \sigma_r = 0$, where $r = rank(A)$; and $V^T$ is an $n \times r$ orthogonal matrix ($V^T V = I_r$) file-by-dimension matrix. After SVD is completed, a rank-k approximation to matrix A can be obtained, where *k* represents the number of dimensions (or factors) retained, and $k \leq r$. The rank-k approximation process is known as *dimensionality reduction*, which involves truncating all three matrices to $k$ dimensions. After SVD and dimensionality reduction are applied, a semantic space representation of the source-code programs (i.e. source-code files) is created, where semantic similarity (or relatedness) among the programs can be measured by the relative closeness of files (or other artefacts e.g. methods, terms) in the similar, it means that these terms have the same meaning in the context being used (i.e. synonyms, terms that have been renamed). In the context of plagiarism detection, files that are more similar than others in the corpus are of interest, i.e. it is those files that are considered suspicious of plagiarism [2].

## C. Neuro-Fuzzy Module

Let $V_{n\times k}$ be the reduced source-code file-by-dimensions matrix containing only the selected features which resulted after applying SVD and Dimensionality reduction. This module takes as input matrix $V_{n\times k}$ and applies a neuro-fuzzy learning algorithm. The learning algorithm works in two sequential steps.

- **Step 1:** Fuzzy C-Means (FCM) clustering [4] is applied to generate a collection of clusters where each cluster contains the source-code file characterised by a similar collection of identifiers (i.e. the terms found in source-code files after pre-processing). Formally, let $V_{n\times k}$ me be the $n \times k$ source-code file by dimension matrix containing source-code file vectors $v_i$ such that $V = [v_1, v_2, v_3, \ldots, v_n]$, where each $v_i$ file vector contains $k$ number of features selected after applying SVD and dimensionality reduction. Let $c$ be the number of clusters and $n$ be the total number of source-code files such that $2 \leq c < n$. Matrix $V$ is input into the FCM algorithm which returns a list of cluster centres $X = x_1, \ldots, x_c$ and a membership matrix $U = \mu_{i,k} \in [0,1],\ i = 1, \ldots, n,\ k = 1, \ldots, c$, where each element $\mu_{ik}$ holds the total membership of a source-code file $v_k$ belonging to cluster $c_i$. FCM updates the cluster centres and the membership grades of each source-code file using the objective function shown in equation (2)

$$J(U, c_1, \ldots, c_c) = \sum_{i=1}^{c} \sum_{k=1}^{N} \mu_{ik}^m ||v_k - x_i||^2, 1 \leq m \leq \infty \tag{2}$$

where $\mu_{ik}$ represents the degree of membership of a source-code file $v_i$ in the $ith$ cluster; $x_i$ is the cluster centre of fuzzy group $i$; $|| * ||$ is the Euclidean distance between $ith$ cluster and $jth$ data point; and $m \in [1, \infty]$ is a weighting exponent.
Hence, FCM clustering is applied to initially classify the data into clusters and generate a Sugeno-type Fuzzy Inference System (FIS) structure containing a single rule for each cluster. The number of clusters determines the number of rules and membership functions in the generated FIS. These initial rules are thereafter used and tailored to the input data using ANFIS.

- **Step 2:** This step uses the collection of TSK rules as input to the ANFIS [26] algorithm. ANFIS uses a hybrid learning algorithm to identify the membership function parameters of single-output Sugeno-type fuzzy inference systems. Applying the FCM algorithm (described in Step 1) prior to applying the ANFIS algorithm in order to reduces the complexity of the system by means of the number of rules generated by ANFIS. The architecture of a type-3 ANFIS, which is the ANFIS used in our proposed model, is explained in [5]. Briefly, ANFIS tunes the FIS parameters using the input/output training data in order to optimise the model. ANFIS terminates the training process stops when the designated epoch number is reached or the training error goal is achieved. The performance of ANFIS is evaluated using the array of root mean square training errors, which essentially computes the difference between the FIS output and the training data output at the end of each epoch. The membership degree of a source-code file determines how close a source-code file is to the next cluster. Hence, let $c_a$ and $c_b$ be two clusters, a source-code file $v_k$ can belong to cluster $c_a$ such that $v_k \in c_a$, or it can belong in the intersection area between two clusters such that $v_k \in c_a \wedge v_k \in c_b$.

## IV. Experiments: Methodology and Results

### A. Methodology

The proposed Fuzzy-based system was tested on two Java source-code datasets. These datasets are described in subsection IV-A1. The performance of the proposed algorithm is evaluated against the FCM, SOM, and the RKR-GST (JPlag) algorithms using the evaluation measures described in subsection IV-A2. JPlag is a well-known tool, accessible online, and a fast implementation of the Running Karp-Rabin Greedy String Tiling algorithm, and for these reasons it was selected for experimentation as opposed to the other string-matching algorithms.

*1) Datasets:* Two corpora comprising of Java source-code files created by undergraduate students on a Programming module were used. Table I shows the characteristics of each dataset, where *Number of files* is the total number of files in a corpus, and *Number of terms* is the number of terms found in an entire source-code corpus after pre-processing is performed. *Number of suspicious files* is the total number of files that were detected in a corpus and judged as suspicious by academics. The last row of the table indicates the number of files excluded from comparison by JPlag. JPlag reads and parses the files prior to comparing them. Files that cannot be read or do not parse successfully are excluded from the comparison process by JPlag.

TABLE I: The Datasets

| | Dataset A | Dataset B |
|---|---|---|
| Number of files | 191 | 176 |
| Number of terms | 481 | 640 |
| Total number of suspicious files | 35 | 32 |
| Total number of files not parsed by JPlag | 7 | 6 |

Table II shows the identified classes, and this information will be used for evaluating the clustering performance of each algorithm. In Table II, classes 1-8 and 1-7 of Dataset A and B respectively, contain similar files, and class 0 contains all files which are not similar and hence could not be placed in the other clusters. When forming the classes of similar files, the aim was to identify those files which contain source-code fragments which are distinct in files. Suspicious source-code fragments must not be "short, simple, trivial (unimportant), standard, frequently published, or of limited functionality and solutions" because these may not provide strong evidence for proving plagiarism among programming solutions written by multiple authors [2]. Hence, small source-code fragments that are likely to be similar in many solutions can be used to examine further the likelihood of plagiarism, but alone they may not provide sufficient evidence for proving plagiarism [2].

TABLE II: Identified Classes (Targets)

| Dataset A | | | Dataset B | | |
|---|---|---|---|---|---|
| 0 | 156 | 81.675 | 0 | 144 | 81.818 |
| 1 | 2 | 1.047 | 1 | 5 | 2.841 |
| 2 | 5 | 2.618 | 2 | 7 | 3.977 |
| 3 | 8 | 4.188 | 3 | 6 | 3.409 |
| 4 | 2 | 1.047 | 4 | 3 | 1.705 |
| 5 | 6 | 3.141 | 5 | 3 | 1.705 |
| 6 | 5 | 2.618 | 6 | 6 | 3.409 |
| 7 | 5 | 2.618 | 7 | 2 | 1.136 |
| 8 | 2 | 1.047 | | | |

*2) Performance evaluation measures for plagiarism detection:* This section describes the performance evaluation measures for comparing the performance of the proposed Fuzzy clustering approach to Fuzzy C-Means, the Self-Organising Map (SOM), and a variation of the Running Karp-Rabin Greedy-String-Tiling algorithm (RKR-GST) implemented by JPlag. Due to the fact that the similarity values provided by these algorithms are not directly comparable, since they use a different measure of similarity, the Recall, Precision, Fscore, Specificity, and Accuracy evaluation techniques, and the Pearson Correlation Coefficient have been utilised. In addition, the Wilcoxon rank sum test has been applied to determine any statistical significant difference among the performance of the algorithms. Let $c_i = \{sc_1, sc_2, \ldots, sc_n\}$ and $pc_i = \{sc_1, sc_2, \ldots, sc_n\}$ denote a known class $c_i$ and a predicted cluster $pc_i$ respectively, each containing $n$ source-code files where $n \leq N$, and $N$ is the total number of source-code files in a corpus. In order to be able to compare the performance of all approaches, the known class sets were constructed after two academics scrutinised the outputs of all algorithms and devised subsets of similar files for each dataset. Hence, given a set of predicted clusters $pc_i$ and known classes $c_i$, Precision, Recall,and Fscore (or F-measure), Accuracy, and Specificity can be calculated. *Precision* of cluster $pc_i$ (also called positive predictive value) is the fraction of retrieved instances that are relevant. *Recall*, of cluster $pc_i$ (also known as sensitivity) is the fraction of relevant instances that are retrieved. For a given cluster $c_i$, true positives, $tp$, is the number of source-code files correctly labelled as belonging to the cluster; false positives, $fp$, is the number of files incorrectly labelled as belonging to the cluster; false negatives, $fn$, is the number of files not labelled as belonging to the correct cluster (but should have been); and false positives, $fp$, is the number of files incorrectly classified as belonging to the cluster. *Accuracy* is the proportion of files labelled as true (both true positives and true negatives), and *Specificity* is the true negative rate (i.e. files correctly identified as negative). The functions of the above mentioned evaluation measures are found below. The closer the value of the evaluation measures is to 1.0, the better the performance of the system.

$$Specificity = \frac{\sum_{i=1}^{c}(tn_i)}{\sum_{i=1}^{c}(fp_i + tn_i)}, \in [0,1]. \quad (3)$$

$$Accuracy = \frac{\sum_{i=1}^{c}(tp_i + tn_i)}{\sum_{i=1}^{c}(tp_i + tn_i + fp_i + fn_i)}, \in [0,1], \quad (4)$$

$$Precision = \frac{\sum_{i=1}^{c}(tp_i)}{\sum_{i=1}^{c}(tp_i + fp_i)}, \in [0,1], \quad (5)$$

$$Recall = \frac{\sum_{i=1}^{c}(tp_i)}{\sum_{i=1}^{c}(tp_i + fn_i)}, \in [0,1], \quad (6)$$

where $c$ be the total number of classes which contain similar source-code files. $Fscore_\beta$ is computed as function (7)

$$Fscore_\beta = \frac{(\beta^2 + 1.0)(Precision \times Recall)}{\beta^2 \times (Precision + Recall)}, \in [0,1], \quad (7)$$

where $AF \in [0,1]$, and the $\beta$ coefficient provides a means of biasing Fscore towards Precision or Recall. In particular setting $\beta = 0.5$ biases it toward precision; setting $\beta = 1.0$ weights Precision and Recall equally; and setting $\beta = 2.0$ biases toward Recall). In the experiments, all three cases have been tested to determine the relative performance of the various clustering algorithms, when emphasizing Recall or Precision, and both. Hence, to penalise false negatives more strongly than false positives by selecting a value $\beta > 1$, thus giving more weight to Recall.

## B. Experiment Results

This section describes the results from the experiments performed on two datasets. Datasets A and B consist of Java files written for the purpose of solving a particular programming task. For this reason a lot of the files contain similar source-code fragments that are not actually suspicious.

TABLE III: Datasets A and B: SOM cluster results at c=20 clusters

| Dataset A | | | | Dataset B | | | |
|---|---|---|---|---|---|---|---|
| CI | #Files | %of Files | Sim? | CI | #Files | %of Files | CSim? |
| 1 | 2 | 1.047 | N | 1 | 11 | 6.250 | N |
| 2 | 4 | 2.094 | N | 2 | 5 | 2.841 | Y |
| 3 | 7 | 3.665 | N | 3 | 5 | 2.841 | Y |
| 4 | 2 | 1.047 | N | 4 | 2 | 1.136 | Y |
| 5 | 2 | 1.047 | N | 5 | 16 | 9.091 | N |
| 6 | 2 | 1.047 | N | 6 | 6 | 3.409 | Y |
| 7 | 5 | 2.618 | Y | 7 | 7 | 3.977 | N |
| 8 | 11 | 5.759 | Y | 8 | 3 | 1.705 | Y |
| 9 | 6 | 3.141 | Y | 9 | 7 | 3.977 | N |
| 10 | 9 | 4.712 | N | 10 | 19 | 10.795 | N |
| 11 | 56 | 29.319 | Y | 11 | 15 | 8.523 | N |
| 12 | 55 | 28.796 | N | 12 | 4 | 2.273 | Y |
| 13 | 3 | 1.571 | N | 13 | 5 | 2.841 | N |
| 14 | 4 | 2.094 | N | 14 | 25 | 14.205 | N |
| 15 | 3 | 1.571 | N | 15 | 9 | 5.114 | N |
| 16 | 5 | 2.618 | Y | 16 | 16 | 9.091 | N |
| 17 | 7 | 3.665 | N | 17 | 6 | 3.409 | N |
| 18 | 3 | 1.571 | N | 18 | 6 | 3.409 | Y |
| 19 | 3 | 1.571 | N | 19 | 7 | 3.977 | N |
| 20 | 2 | 1.047 | Y | 20 | 2 | 1.136 | Y |

This kind of similarity can be considered as noise and can return false positives. For this reason, applying dimensionality reduction allows for such noise to be removed and the files which contain similar but distinct code to be clustered together. After applying SVD, the value of dimensionality was set to $k = 50$ for Dataset A, and $k = 15$ for Dataset B, and these values were selected experimentally. Thereafter, each matrix V was used for applying the FCM clustering using a suitable number of clusters.

As shown in Table II, the Dataset a was split into $c = 9$ classes of which 8 contained similar source-code files, and one cluster all the rest of the files (i.e. non-similar). Regarding Dataset B, a total of $c = 8$ classes were formed, of which 7 classes contained similar source-code files, and one class contained the rest of the files (i.e. non-similar). Each algorithm was run on each dataset by setting the number of clusters to $c$, and then the files returned by each algorithm in each cluster were scrutinised for similarity by academics prior to computing the evaluation measures. The reason was to ensure that any files missed by academics but returned by the algorithms were added to the sets of target files. The final sets of target files (found in Table II) were used for the purpose of evaluating the clustering performance of the algorithms. Setting SOM to $c = 10$ or $c = 15$ clusters did not return good clustering results on either datasets, and for this reason $c = 20$ was selected. As shown in Table III with $c = 20$ clusters SOM was able to identify clusters of similar files, however, it also returned a number of clusters which did not contain any similar files. Essentially, the number of SOM clusters was increased to allow the algorithm to reach its best performance. The experiment results for Datasets A and B are shown in Tables IV and V respectively.

Table III shows the number of files returned in each cluster which resulted after applying the SOM algorithm when using 20 clusters, and whether the clusters identified contained any similar files. Table VI shows the average performance of all algorithms across only those clusters which contained similar files. In addition the Ratio of correctly Detected Clusters (Ratio DC), is defined as follows:

TABLE IV: Datasets A Results

| | **Dataset A Performance** | | | |
|---|---|---|---|---|
| | **Our Approach** | **FCM** | **SOM** | **RKR-GST** |
| Accuracy | 1.000 | 0.957 | 0.988 | 0.982 |
| Specificity | 1.000 | 0.972 | 0.996 | 1.000 |
| Ratio DC | 1.000 | 0.750 | 0.625 | 0.125 |
| Precision | 1.000 | 0.208 | 0.793 | 1.000 |
| Recall | 1.000 | 0.314 | 0.657 | 0.200 |
| $\text{Fscore}_{1.0}$ | 1.000 | 0.250 | 0.719 | 0.333 |
| $\text{Fscore}_{0.5}$ | 1.000 | 0.223 | 0.762 | 0.556 |
| $\text{Fscore}_{2.0}$ | 1.000 | 0.285 | 0.680 | 0.238 |
| FNR | 0.000 | 0.686 | 0.343 | 0.800 |
| FPR | 0.000 | 0.028 | 0.004 | 0.000 |
| TP | 35.000 | 11.000 | 23.000 | 7.000 |

TABLE V: Dataset B: Results

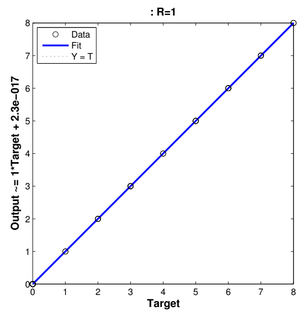
| | **Dataset B Performance** | | | |
|---|---|---|---|---|
| | **Our Approach** | **FCM** | **SOM** | **RKR-GST** |
| Accuracy | 0.997 | 0.963 | 1.000 | 0.981 |
| Specificity | 0.998 | 0.976 | 1.000 | 0.999 |
| Ratio dc | 1.000 | 0.571 | 1.000 | 0.429 |
| Precision | 0.938 | 0.356 | 1.000 | 0.900 |
| Recall | 0.938 | 0.500 | 1.000 | 0.281 |
| $\text{Fscore}_{1.0}$ | 0.938 | 0.416 | 1.000 | 0.429 |
| $\text{Fscore}_{0.5}$ | 0.938 | 0.377 | 1.000 | 0.625 |
| $\text{Fscore}_{2.0}$ | 0.938 | 0.462 | 1.000 | 0.326 |
| FNR | 0.063 | 0.500 | 0.000 | 0.719 |
| FPR | 0.002 | 0.024 | 0.000 | 0.001 |
| TP | 30.000 | 16.000 | 32.000 | 9.000 |

$$\text{Ratio DC} = \frac{|\text{number of clusters correctly detected}|}{|\text{number of known classes}|} \quad (8)$$

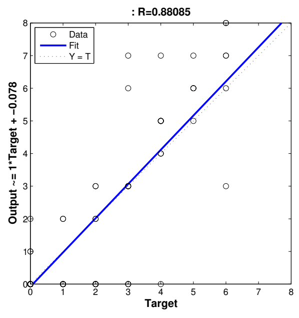
The evaluations in Table VI only show the performance of the algorithms for the correctly identified clusters, and hence the algorithms that returned a few clusters containing similar files have high Precision but low Recall values. For this reason we consider Recall to be more important than Precision, and also *Ratio DC* should be taken into consideration. In summary Table VI shows that the proposed algorithm outperformed all other algorithms, followed by the SOM, FCM, and finally by the RKR-GST (JPlag). To conclude, we consider $Fscore_2$ which gives more bias towards Recall our proposed approach outperformed: FCM by $0.595$ points, SOM by $0.129$ points, and RKR-GST(JPlag) by $0.687$ points. Figures 2 and 3 show the correlations among the actual values returned by each algorithm and the target values. In order to determine whether there exist any statistically significant difference in performance, the Wilcoxon rank sum test was applied. This tests the null hypothesis that the target and actual values have equal medians,

TABLE VI: Datasets A and B: Average Performance

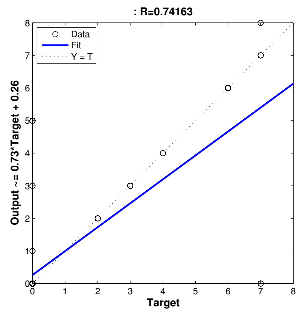
| | **Average Performance** | | | |
|---|---|---|---|---|
| | **Our Approach** | **FCM** | **SOM** | **RKR-GST** |
| Accuracy | 0.998 | 0.960 | 0.994 | 0.981 |
| Specificity | 0.999 | 0.974 | 0.998 | 1.000 |
| Ratio dc | 1.000 | 0.661 | 0.813 | 0.277 |
| Precision | 0.969 | 0.282 | 0.897 | 0.950 |
| Recall | 0.969 | 0.407 | 0.829 | 0.241 |
| $\text{Fscore}_{1.0}$ | 0.969 | 0.333 | 0.859 | 0.381 |
| $\text{Fscore}_{0.5}$ | 0.969 | 0.300 | 0.881 | 0.590 |
| $\text{Fscore}_{2.0}$ | 0.969 | 0.374 | 0.840 | 0.282 |
| FNR | 0.031 | 0.593 | 0.171 | 0.759 |
| FPR | 0.001 | 0.026 | 0.002 | 0.000 |
| TP | 32.500 | 13.500 | 27.500 | 8.000 |



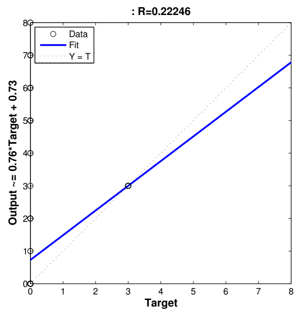
(a) **Our Approach**



(b) Fuzzy C-Means

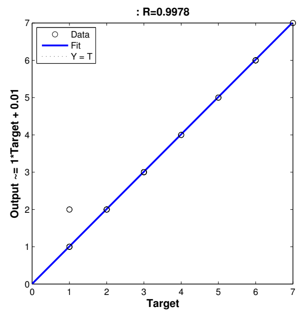


(c) SOM

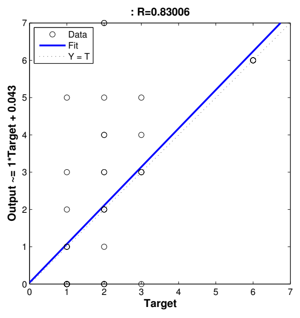


(d) RKR-GST(JPlag)

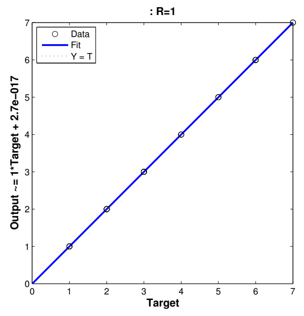
Fig. 2: Dataset 1 Experiment results: Comparison of values returned by each approach



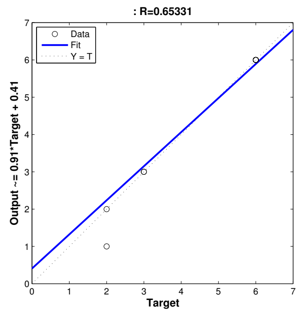
(a) **Our Approach**



(b) Fuzzy C-Means



(c) SOM



(d) RKR-GST(JPlag)

Fig. 3: Dataset 2 Experiment results: Comparison of values returned by each approach

against the alternative that they have not. The test assumes that the two samples are independent. For the experiment the level of significance of 5% ($\alpha = 0.05$) was selected. Hence, for each algorithm, the following hypotheses were tested:

- $H_0$: There is no significant difference among the results returned by algorithm x and target values y.
- $H_1$: There is a significant difference among the results returned by algorithm x and target values y.

In reporting the results below, note that the correlation r, along with the p value of the significance test, and the hypothesis $h$ value are reported, where, $h = 0$, implies that there are no significant differences among the performances of the two algorithms hence the null hypothesis $H_0$ is accepted, whereas a rejection of the null hypothesis implies that there are significant differences in the performance of the two tests, hence the hypothesis $H_1$ is accepted. **Dataset A:** our approach&targets(r=1.00, p=1.0, h=0); FCM&targets(r=0.881, p=0.087, h=0), SOM&targets(r=0.742, p=0.479, h=0), RKR-GST(JPlag)&targets(r=0.223, p=0.000, h=1). **Dataset B:** our approach-targets(r=0.998, p=0.986, h=0); FCM&targets(r=0.830, p=0.216, h=0), SOM&targets(r=1.000, p=1.000, h=0), RKR-GST(JPlag)&targets (r=0.653, p=0.000, h=1). In summary the results revealed that our proposed system outperformed all other systems, with significant differences detected among the RKR-GST(JPlag) algorithm and our approach.

## V. Conclusion

Source-code plagiarism detection in programming, concerns the identification of source-code files that contain similar and/or identical source-code fragments. Fuzzy clustering approaches are a suitable solution to detecting source-code plagiarism due to their capability to capture the qualitative and semantic elements of similarity. However, Fuzzy clustering approaches have not been evaluated on the source-code plagiarism detection problem as much as other approaches. This paper proposes a neuro-fuzzy based approach to clustering source-code for detecting clusters which could contain similar and hence plagiarised files. In particular, we demonstrate how the Fuzzy C-Means (FCM), Self-Organising Map(SOM), and our proposed approach which is based on FCM and the Adaptive Neuro-Fuzzy Inference System (ANFIS) can be used to detect source-code plagiarism. We compare the performance of these models with JPlag's implementation of the well known Running Karp-Rabin Greedy String-Tiling Algorithm (RKR-GST). Our proposed approach uses data mining techniques to represent the pre-processed source-code files into a Vector Space Model and removes noise from the data prior to applying the FCM clustering algorithm to cluster the data and then ANFIS to optimise the performance of the detection system. The proposed approach appears to overcome the several problems currently encountered by state-of-the-art plagiarism detection algorithms, these problems are: language dependency, misdetection due to code re-shuffling including local confusion, and many of the existing algorithms and tools also have the disadvantage of not being able to include files that do not parse in the comparison process. In particular, we have exploited well established clustering algorithms to the source-code plagiarism problem, and results from experiments demonstrate that the performance of the proposed approach overcomes all other approaches on the particular datasets. In future work we plan to evaluate our proposed algorithm using more datasets, and also datasets written in other programming languages.

## REFERENCES

[1] G. Cosma and M. Joy, "Towards a definition of source-code plagiarism," *Education, IEEE Transactions on*, vol. 51, no. 2, pp. 195–200, May 2008.

[2] C. G. and J. M., "An approach to source-code plagiarism detection and investigation using latent semantic analysis," *Computers, IEEE Transactions on*, vol. 61, no. 3, pp. 379–394, March 2012.

[3] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with JPlag," *Journal of Universal Computer Science*, vol. 8, no. 11, pp. 1016–1038, 2002.

[4] J. C. Bezdek, *Pattern Recognition with Fuzzy Objective Function Algorithms*. Norwell, MA, USA: Kluwer Academic Publishers, 1981.

[5] J. S. R. Jang, "Anfis: adaptive-network-based fuzzy inference system," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 23, no. 3, pp. 665–685, Aug. 2002.

[6] T. Kohonen, "Self-organized formation of topologically correct feature maps," *Biological Cybernetics*, vol. 43, no. 1, pp. 59–69, 1982. [Online]. Available: http://dx.doi.org/10.1007/BF00337288

[7] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with jplag," *j-jucs*, vol. 8, no. 11, pp. 1016–1038, nov 2002.

[8] M. Mozgovoy, "Desktop tools for offline plagiarism detection in computer programs," *Informatics in Education*, vol. 5, no. 1, pp. 97–112, 2006.

[9] M. Wise, "Yap3: improved detection of similarities in computer program and other texts," *SIGCSE Bulletin*, vol. 28, no. 1, pp. 130–134, 1996.

[10] C. Kustanto and I. Liem, "Automatic source code plagiarism detection," in *Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, 2009. SNPD '09. 10th ACIS International Conference on*, May 2009, pp. 481–486.

[11] B. Muddu, A. Asadullah, and V. Bhat, "Cpdp: A robust technique for plagiarism detection in source code," in *Software Clones (IWSC), 2013 7th International Workshop on*, May 2013, pp. 39–45.

[12] M. Joy and M. Luck, "Plagiarism in programming assignments," *IEEE Transactions on Education*, vol. 42, no. 2, pp. 129–133, 1999. [Online]. Available: http://eprints.dcs.warwick.ac.uk/81/

[13] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002. [Online]. Available: http://dx.doi.org/10.1109/TSE.2002.1019480

[14] A. Ahtiainen, S. Surakka, and M. Rahikainen, "Plaggie: Gnu-licensed source code plagiarism detection engine for java exercises." in *Koli Calling*, A. Berglund and M. Wiggberg, Eds. ACM, 2006, pp. 141–142.

[15] L. Moussiades and A. Vakali, "PDetect: A clustering approach for detecting plagiarism in source code datasets," *The Computer Journal*, vol. 48, no. 6, pp. 651–661, 2005.

[16] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[17] O. Ajmal, M. Missen, T. Hashmat, M. Moosa, and T. Ali, "Eplag: A two layer source code plagiarism detection system," in *Digital Information Management (ICDIM), 2013 Eighth International Conference on*, Sept 2013, pp. 256–261.

[18] R. Kilgour, A. R. Gray, P. J. Sallis, and S. G. Macdonell, "A fuzzy logic approach to computer software source code authorship analysis," in *Fourth International Conference on Neural Information Processing – The Annual Conference of the Asian Pacific Neural Network Assembly (ICONIP'97*. Springer-Verlag, 1997, pp. 865–868.

[19] S. Romano, G. Scanniello, M. Risi, and C. Gravino, "Clustering and lexical information support for the recovery of design pattern in source code," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, Sept 2011, pp. 500–503.

[20] Y. Liu and Y. Zhang, "An improved curd algorithm for source code mining," in *Fuzzy Systems and Knowledge Discovery, 2008. FSKD '08. Fifth International Conference on*, vol. 4, Oct 2008, pp. 335–339.

[21] X. Wang, "Kfcm algorithm based on the source code mining method study," in *Intelligent Systems Design and Engineering Applications (ISDEA), 2014 Fifth International Conference on*, June 2014, pp. 586–588.

[22] Y. Li, Y. Zhao, and B. Liu, "Similarity assessment of program samples based on theory of fuzzy," in *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*, Dec 2013, pp. 202–207.

[23] L. Yu, J. Zhou, Y. Yi, J. Fan, and Q. Wang, "A hybrid approach to detecting security defects in programs," in *Quality Software, 2009. QSIC '09. 9th International Conference on*, Aug 2009, pp. 1–10.

[24] N. Bettenburg, S. Thomas, and A. Hassan, "Using fuzzy code search to link code fragments in discussions to source code," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, March 2012, pp. 319–328.

[25] M. W. Berry, S. T. Dumais, and G. W. O'Brien, "Using linear algebra for intelligent information retrieval," *SIAM Rev.*, vol. 37, no. 4, pp. 573–595, Dec. 1995.

[26] J.-S. R. Jang, "Input selection for anfis learning," in *In Proceedings of the IEEE International Conference On Fuzzy Systems*, 1996, pp. 1493–1499.