A Machine Learning Based Tool for Source Code Plagiarism Detection

Upul Bandara, and Gamini Wijayarathna

Abstract— **Source code plagiarism is a severe problem in academia. In academia programming assignments are used to evaluate students in programming courses. Therefore checking programming assignments for plagiarism is essential. If a course consists of a large number of students, it is impractical to check each assignment by a human inspector. Therefore it is essential to have automated tools in order to assist detection of plagiarism in programming assignments.**

Majority of the current source code plagiarism detection tools are based on structured methods. Structural properties of a plagiarized program and the original program differ significantly. Therefore it is hard to detect plagiarized programs when plagiarism level is 4 or above by using tools which are based on structural methods.

This paper presents a new plagiarism detection method, which is based on machine learning techniques. We have trained and tested three machine learning algorithms for detecting source code plagiarism. Furthermore, we have utilized a meta-learning algorithm in order to improve the accuracy of our system.

Index Terms— **k-nearest neighbor, machine learning, naïve bayes classifier, plagiarism detection, source code**

I. INTRODUCTION

Detection of source code plagiarism is equally valuable for both academia and industry. Zobel [1] has pointed out that "students may plagiarize by copying code from friends, the Web or so called "private tutors". Most programming courses in universities evaluate students based on the marks of programming assignments. If a programming course consists of a large number of students, it is impractical to check plagiarism by human inspectors. Moreover Liu and et al [2] have mentioned that "A quality plagiarism detector has a strong impact to law suit prosecution". Therefore there is a huge demand for accurate source code plagiarism detection systems from both academia and industry.

Woo and Cho [3] have mentioned two methods for plagiarism detection.

1. *Structured Based Method*: this method considers structural characteristics of documents when developing plagiarism detection algorithms.

2. *Attribute Counting Method*: this method extracts various measurable features (or metrics) from documents. Extracted metrics use as input for similarity detection algorithms.

Presently most of the source code plagiarism detection algorithms are based on the structured method [3, 4, 2]. In addition to that there are few attempts which are based on the attribute counting approach [5, 6].

Faidhi and Robinson [7] have mentioned a spectrum of six levels in program plagiarism. Level 0 is the lowest level of plagiarism. Level 0 represents copying someone else's program without modifying it. Level 6 represents highest level of plagiarism. It represents modifying program's control logic in order to achieve the same operation. It is to be noted that when moving from level 0 to level 6, structural characteristics of a plagiarized program varies from the original program. Moreover Arwin and Tahaghoghi [4] have mentioned that structured based systems rely on the belief that the similarity of two programs can be estimated from the similarity of their structures. Since structured properties of plagiarized documents vary from its original document it is difficult to detect plagiarism when plagiarism level is 4 or higher.

On the other hand plagiarism detection systems which are based on attribute counting techniques are not relying on structural properties of the source program. Therefore they are not suffering from the problem mentioned above. But presently systems which are based on attribute counting techniques are not accurate enough for practical applications [5, 6].

Therefore we have proposed a new system that is based on attribute counting technique. Moreover we have used machine learning approach in order to detect similarity between source codes. Ethem Alpapaydin [8] has pointed out that "there is no single learning algorithm that in any domain always induces most accurate learner". Moreover, he has mentioned that, by combining multiple base learners in a suitable way prediction performance can be improved. Therefore instead of using just one learning algorithm, we have used three learning algorithms for training our system. We tested our system with source codes belonging to ten developers. During the training period we found out that not a single algorithm was capable of identifying the source code files belonging to all the developers with adequate accuracy. But one interesting observation was that, the results generated by three algorithms were complementing each other. Therefore we have decided to use a meta-learning algorithm in order to combine the results generated by three learning algorithms

The rest of the paper is organized as follows. Section II we will be presenting plagiarism detection methods based on the attribute counting techniques. Section III we will be discussing the machine learning algorithms for plagiarism detection. Section IV, we will discuss on implementation, training and testing our system. We will conclude our paper by discussing the final results and future works of our system in Section V.

Manuscript received September 06, 2011, revised September 8, 2011.

U Bandara is with the *Virtusa* Corporation, Sri Lanka (e-mail: upulbandara@ gmail.com).

G. Wijayarathna is with the Faculty of Science, University of Kelaniya, Sri Lanka(e-mailgamini@kln.ac.lk).

II. Related Work

Few research papers have been written on source code plagiarism detection using the attribute counting techniques. Steve Engels and et al [5] describe code metrics and feature based neural network for detecting plagiarism. As the first step this method extracts source code metrics from software source codes. Extracted metrics are used as the input for a feature-based neural network. Output of the feature-based neural network was presented in terms of precision and recall. Precision for cheating cases was 0.6464 and recall for cheating cases was 0.4158.

Dian and Samadzadeh [9] have published a paper on identification of source code authors using source code metrics. This technique is based on extracting a large number of source code metrics. They have used 56 source code metrics. The extracted source code metrics are subjected to various statistical techniques in order to identify the authors of each source code.

Lange and Mancoridis [6] have proposed a source code plagiarism detection method, which uses source code metric histogram and genetic algorithm. First source code metrics were generated from software source code files. Then the normalized histograms were generated for each source code metric. The normalized histograms were used as the input for the nearest neighbor classifier. One interesting feature of this research is that, they have used a genetic algorithm in order to identify the optimized set of source code metrics. According to their research paper, the system is capable of identifying the true author of each source code file with 55 percent accuracy.

III. Algorithms for Source Code Author Identification

Any algorithm which is suitable for pattern recognition can be used for source code author identification without modifications or with minor modifications. In this section we will discuss three such algorithms, that we used for our research.

A. *Naïve Bayes Classifier*

This classifier is based on Bayes theorem. When C with small number of classes or outcomes, conditional on several features denoted by $F_1, F_2, \dots, F_n$, using Bayes theorem:

$$p(C|F_1, \dots, F_n) = \frac{p(C)p(F_1, \dots, F_n|C)}{p(F_1, \dots F_n)} \qquad (1)$$

Using conditional probability:

$$p(C, F_1, \dots, F_n) = p(C)p(F_1, \dots, F_n|C) \qquad (2)$$

Using chain rule:

$$p(C, F_1, \dots, F_n) = p(C)p(F_1|C)p(F_2|C, F_1) \dots p(F_n|C, F_1, F_2, F_3, \dots, F_{n-1}) \quad (3)$$

Using naïve property:

F_i is conditionally independent of every other F_j for all $i \neq j$ this means

$$p(F_i|C, F_j) = p(F_i|C) \qquad (4)$$

Therefore Naïve Bayes model can be written as:

$$p(C|F_1, F_2, \dots F_n) = \frac{1}{Z} p(C) \prod_{i=1}^{n} p(F_i\,|C) \qquad (5)$$

Z is a constant which is dependent only on $F_1, \dots, F_n$

We can directly use the Naïve Bayes classifier for our research. In our research C means the number of authors in the experiment and $F_1, F_2, \dots, F_n$ means a set of source code metrics extracted from the source codes.

Christopher and et al [10] have mentioned two methods to construct the Naïve Bayes classifier as enumerated below.

1. Multinomial Naïve Bayes classifier
2. Bernoulli Naïve Bayes classifier

The main difference between the above two variations is the way they calculate posterior probability or $p(F_i|C)$. Multinomial naïve bayes classifier calculates posterior probability as given in equation (6).

$$p(F_i|C) = \frac{T_{F_i} + 1}{\sum_{t' \in V} T_{F_{i'}} + B} \qquad (6)$$

Where T_{F_i} is the number of occurrences of F_i in the training documents of class c. $V$is the vocabulary (or unique metrics) of the training dataset and $B = |V|$

Bernoulli naïve bayes classifier calculates posterior probability as given in equation (7).

$$p(F_i|C) = \frac{N_{F_i} + 1}{N_c + 2} \qquad (7)$$

Where $N_{F_i}$is the number of documents in class c that contain F_i. N_c is the number of documents in class c.

B. *k-Nearest Neighbor (kNN) Algorithm*

The k-Nearest Neighbor Algorithm is one of the simplest machine learning algorithms, suitable for pattern recognition. The k-Nearest Neighbor (kNN) algorithm often performs well in most pattern recognition applications [11].

k is a parameter in the kNN algorithm. It is necessary to select the correct k value for the kNN algorithm by conducting several tests with various k values. kNN algorithm is based on the "Euclidian Distance"

Let X_i is a source code file with p features (or metrics) $(x_{i1}, x_{i2}, \dots, x_{ip})$.Euclidian distance between source code file X_i and X_j is defined as given below.

$$d(X_i, X_j) = \sqrt{\left(\left(x_{i1} - x_{j1}\right)^2 + \dots + \left(x_{ip} - x_{jp}\right)^2\right)} \qquad (8)$$

The kNN assigns test source code file to majority class of its k nearest neighbors in training data set.

kNN simply memorize all the documents in the training dataset and compares the test document against the training dataset. For this reason kNN is also known as "memory-based learning" or "instance-based learning.

C. AdaBoost Meta-learning Algorithm

Boosting is used to boost the accuracy of any given learning algorithm [12]. There are many different kinds of boosting algorithms are available in the research literature. For this research, we used the AdaBoost meta-learning algorithm. According to Russell and Norvig [11], daBoost shows very important property: if learning algorithms are weak llearningalgorithms, AdaBoost will classify training dataset perfectly for large enough weak learners.

Suppose training set contains data points, $(x_1, y_1), \dots, (x_n y_n)$ where, x_i belongs to instance space X and C_i belongs to small number of classes for example $\{1,2,\dots,K\}$. Goal of the AdaBoost is to find out a classifier $C^*(x)$ that minimizes misclassification error rate. Assuming training data are independently and identically distributed, misclassification error rate is given by following equation.

$$\begin{aligned} E_X, C^{\pi}(x) \neq C = {} & E_x Prob(C(X) \neq C|X) \\ = {} & 1 - E_x Prob(C(X) = C|X) \\ = {} & 1 - E_x Prob(C(X) = C|X) \\ = {} & 1 - \textstyle\sum_{k=1}^{K} E_x[C^{\pi}(X) = x Prob(C = k|X)] \quad (9) \end{aligned}$$

From equation (9) it is clear that,

$$C^*(x) = \arg\max k \ Prob(C = k \mid X = x) \quad (10)$$

will minimize the misclassification error rate.

In AdaBoost, each data point in the dataset is associated with a weight. Initially, all the data points in the dataset are assigning an equal weight. First iteration starts with the first weak leaner. After first iteration, weights of misclassified data points are increased. Second iteration is stared with the next weak learner and it is carried out with newly calculated weights. This process continues for all the weak learners. During the training process, a score is given for each classifier, and the final classifier is the liner combination of weak classifier used in iterations.

AdaBoost algorithm can be implemented in several different ways. Different algorithms calculate weights associated with each data point in different ways. For this research we used the AdaBoost algorithm described by Russell and Norvig [11].

IV. Implementation, Training and Evaluation

Ding and Samadzadeh [9] have mentioned that not all source code metrics contribute equally for source code author identification. According to Ding and Samadzadeh [9] layout metrics perform a much important role than other metrics. In addition to that Lange and Mancoridis [6] have identified and listed source code metrics that perform well for source code identification. Therefore we have identified the following nine metrics shown in Table 1, which perform well in source code identification.

TABLE 1: SOURCE CODE METRICS USE FOR SOURCE CODE AUTHOR IDENTIFICATION

Metrics Name	Description
LineLengthCalculator	This metric measures the number of characters in one source code line.
LineWordsCalculator	This metric measure the number of words in one source code line.
AccessCalculator	Java uses the four level access controls: public, protected, default and private. This metrics calculates the relative frequency of these access levels used by programmers.
CommentsFrequencyCalculator	Java uses three types of comments. This metrics calculate the relative frequency of those comment types used by the programmers.
IndentifiersLengthCalculator	This metric calculates the length of each identifier of Java programs.
InLineSpaceInlineTabCalculator	This metric calculates the whitespaces that occurs on the interior areas of non-whitespace lines.
TrailTabSpaceCalculator	This metric measures the whitespaces and tabs occurring at the end of each non-whitespace line.
UnderscoresCalculator	This metric measures the number of underscore characters used in identifiers.
IndentSpaceTabCalculator	This metric calculates the indentation of whitespaces used at the beginning of each non-whitespace lines.

In order to extract some of the metrics, it was essential to parse source codes files according to the syntactic rules of the programming language, which was used to write that source code. Since our dataset consisted only Java [13] source code files, we used the ANTLR [14] parser generator in order to extract some of the source code metrics.

For each code metric we gave a unique code as shown in Table 2. Next we converted each source code file into a set of tokens together with token frequencies. This process is explained in the next section.

A. Generating a Set of Tokens from the Source Code Files

As we discussed in previous section, for each source code metric we assign a three letter unique code as shown in Table 2.

TABLE 2. CODING SYSTEM OF SOURCE CODE METRICS

Code Metric	Coding System
LineLengthCalculator	LLC
LineWordsCalculator	LWC
AccessCalculator	ACL
CommentsFrequencyCalculator	CFC
IndentifiersLengthCalculator	ILC
InLineSpaceInlineTabCalculator	INT
TrailTabSpaceCalculator	TTS
UnderscoresCalculator	USC
IndentSpaceTabCalculator	IST

For example consider "LineLengthCalculator" code metric generates the metric as shown in Table 3 for a particular source code file.

Using the coding scheme introduced in Table 2.0 we can generate the following tokens and token frequencies for the 'LineLengthCalculator" for the metric shown in Table 3.

TABLE 3. OUTPUT OF LINELENGTHCALCULATOR METRIC

Length of the Line	Number of Occurrences
5	12
8	20
15	13
20	9
32	11
38	3
40	2

TABLE: 4. TOKEN FREQUENCIES OF LINELENGTHCALCULATOR METRIC

Token	Token Frequency
LLC_5	12
LLC_8	20
LLC_15	13
LLC_20	9
LLC_32	11
LLC_38	3
LLC_40	2

As described above, we can represent each source code file as a set of tokens together with token frequencies. Those tokens and token frequencies are used as inputs for our learning algorithms. This process is almost identical to the use of word and word frequencies in document classification problems.

In order to get a better understanding of how these tokens and token frequencies are used in classifier algorithms, consider the following simplified training and testing data set shown in Table 5.

TABLE: 5. SIMPLIFIED TRAINING AND TESTING DATASET

	DocId	(Token, Token Frequency)	Class
Training	1	(LLC_5, 2), (LLC_8, 12), (LLC_7,2)	Author 1
	2	(LLC_5, 3), (LLC_9 ,2), (LLC_12,1)	Author 2
	3	(LLC_5, 1), (LLC_6, 1), (LLC_8,1)	Author 3
Testing	4	(LLC_5, 2), (LLC_8, 1), (LLC_12,2)	?

First we apply multinomial naïve bayes classifier, using equations (5) and (6). Prior probabilities are $P(Author1) = \frac{1}{3}, P(Author2) = \frac{1}{3}, P(Author3) = \frac{1}{3}$. Conditional probabilities of each token are given in Table 6.

TABLE: 6. CONDITIONAL PROBABILITY CALCULATION USING EQUATION (6)

	X=1	*X=2*	*X=3*
$P(LLC\_5 \mid Author\ X)$	$\frac{3}{21}$	$\frac{4}{11}$	$\frac{2}{8}$
$P(LLC\_8 \mid AuthorX)$	$\frac{13}{21}$	$\frac{1}{11}$	$\frac{2}{8}$
$P(LLC\_12 \mid AuthorX)$	$\frac{1}{21}$	$\frac{2}{11}$	$\frac{1}{8}$

Using equation (5)

$$P(Author1 \mid DocId = 4) \propto \left(\frac{1}{3}\right)\left(\frac{3}{21}\right)^2\left(\frac{13}{21}\right)\left(\frac{1}{21}\right) \approx 0.0002$$

$$P(Author2 \mid DocId = 4) \propto \left(\frac{1}{3}\right)\left(\frac{4}{11}\right)^2\left(\frac{1}{11}\right)\left(\frac{2}{11}\right) \approx 0.0007$$

$$P(Author3 \mid DocId = 4) \propto \left(\frac{1}{3}\right)\left(\frac{2}{8}\right)^2\left(\frac{2}{8}\right)\left(\frac{1}{8}\right) \approx 0.0006$$

Hence we can deduce that, according to the multinomial naïve bayes classifier, DocId = 4 belongs to Author2. Same way using equation (5) and (7), we can apply the Bernoulli naïve bayes classifier to the above dataset. Application of kNN, using equation (2), for the above dataset is shown is Table 6.

TABLE: 7. KNN CALCULATION USING EQUATION (8)

	DocId =4
DocId =1	$\sqrt{((2-2)^2 + (1-12)^2 + (2-0)^2)} = 11.18$
DocId =2	$\sqrt{((2-3)^2 + (1-0)^2 + (2-1)^2)} = 1.73$
DocId =3	$\sqrt{((2-1)^2 + (1-1)^2 + (2-0)^2)} = 2.34$

According to the calculation given in Table 7, we can deduce that, DocId=4 belongs to Author2.

Above calculations show how we can use classification algorithms in order to classify source code authors. Our tool uses the same procedure for classifying source code files to respective authors.

B. Training Dataset

We used the same dataset used by Lange and Mancoridis [6]. This dataset consists of Java Source code files belonging to 10 developers. We divided the dataset into two parts called the "training dataset" and "validation dataset". The training dataset consists of 904 source code files, with at least 40 source code files per developer. Validation dataset used as a "hide-out" date set and used only for final evaluation of the system. The validation dataset consisted of 741 source code files.

C. Implementing the System

The system was completely developed using Java [13] programming language. Furthermore, several source code metrics were generated with the help of ANTLR parser generator library [14]. Figure 1 shows the high-level architecture of the system.

Each sub-system in the Figure 1 is mapped into a Java package except, source code sub-system. It represents the Java source codes, which are used for training and testing the system. Metrics sub-system generates source code metrics from Java source code files.

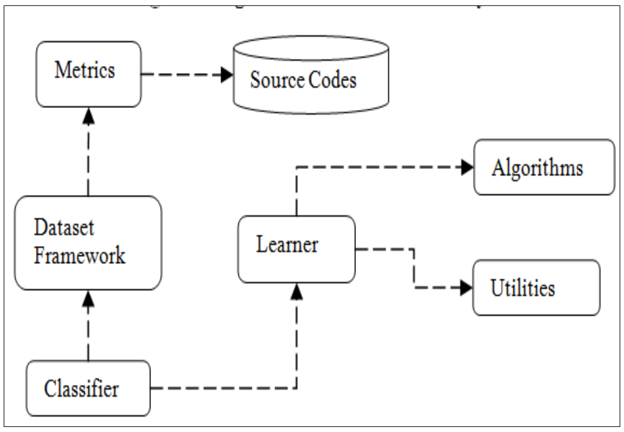


Figure 1.High-level Architecture of the System

Dataset Framework layer provides a high-level interface for the classifier sub-system. Classifier sub-system requests source code metrics via the Dataset Framework layer. Classification algorithms discussed, in Section 3 are implemented in the Algorithms sub-system. Naïve Bayes classifier and kNN classifier are implemented based on the pseudocode as described by Christopher and et al [10]. Furthermore, AdaBoost algorithm is implemented according to the description given by described by Russell and Norvig [11]. Additional functionalities are implemented in the Utilities sub-system. Learner sub-system works as a high-level abstraction layer. Classifier sub-system requests functionalities of Algorithms and Utilities, sub-systems via the interfaces provided by Learner sub-system.

D. Training the System

Initially the system was trained by using the multinomial naïve Bayes classifier. We started with 100 source code files with 10 files per developer. We increased the number of files by 100 source code files in each of the iteration. Moreover in each of the iteration, we calculated the "confusion matrix" for testing dataset. We continued this process until the training dataset consisted of 800 source code files. After completing this process, we found out that multinomial naïve Bayes classifier performed well when the training set consists of 800 source code files.

The "Confusion matrix" was generated for multinomial naïve Bayes classifier for the hideout dataset. According to the generated "confusion matrix" the multinomial naïve Bayes classifier's prediction performance is poor for Author 5, 6 and 8.

As the second step we trained the Bernoulli naïve Bayes classifier. As we trained the multinomial naïve Bayes classifier we increased training dataset by 100 source code files at a time. Moreover we have found out that bernoulli naïve bayes classifier perform well when training dataset consists of 800 source code files.

According to the generated "confusion matric" for the bernoulli naïve Bayes classifier we have gained some improvement for Author number 8 and 5. But Bernoulli naïve bayes is lacking of classifying the source code files of Author number 6.

Next we investigated the k-Nearest Neighbor (kNN) classifier with the intention of improving the accuracy when classifying Author number 6. Since k is a parameter in the kNN learner, we tested the kNN classifier with several k values in order to obtain the best k value for our experiment. Figure 2 shows the variation of success rates for various k values.

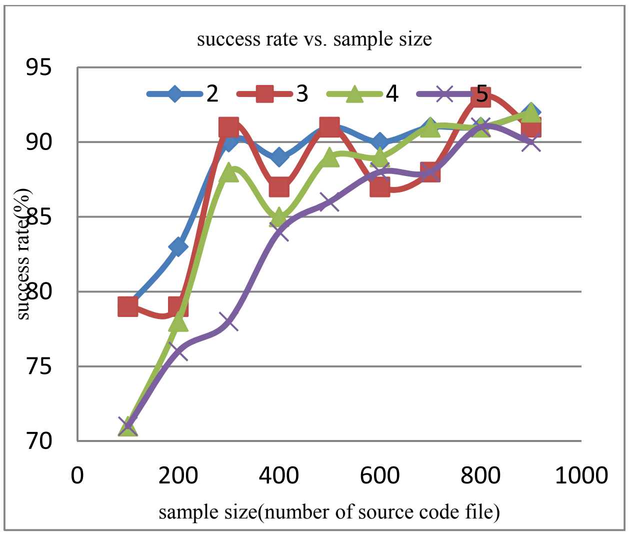


Figure 2. Success rate vs. sample size of kNN algorithm for training dataset

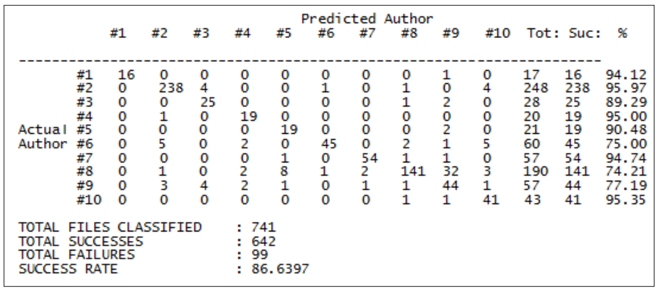
According to Figure 1, optimum training data set size is 800 documents and k value is 3.

Finally we tested the kNN learning algorithm with the above configuration against the hideout data set. According to the final confusion matrix the kNN learner performed well for all the developers except for developer number 8.

After training the system with three different learning algorithms we concluded that, no single algorithm satisfactorily classifies source code files of all 10 authors. But above three learning algorithms can satisfactory classify a subset of the authors in the 10 authors set. Furthermore the above three learning algorithms complement each other when classifying authors. Since three algorithms complement each other we used ensemble learning method to improve the overall accuracy of our system. Therefore we used the AdaBoost [12] algorithm for improving the accuracy of our system.

For training the individual weak-learner we used 800 source code files with not less than 40 source code files for each developer and also for training the AdaBoost algorithm we used 800 source code documents again with not less than 40 documents per author. Figure 3 shows the confusion matrix of running the AdaBoost for our validation dataset.

According to the final output of the AdaBoost algorithm, our system is capable of classifying authors with 86.64 percent accuracy. Testing dataset consists of 741 source code files belonging to 10 authors. Out of 741 files, 642 files were successfully classified by the system. Therefore, from plagiarism detection perspective, we should inspect incorrectly classified 99 files for plagiarism. For example, testing dataset consists of 17 files belonging to Author 1. Out of these 17 files, 16 files were classified correctly. However, one file classified incorrectly under Author 9. Therefore, we can suspect that incorrectly classified file might have some relationship with Author 9. In order to investigate this further, we can manually check that file for plagiarism. Since the level of accuracy of the system relatively high, number of source code files that should be investigated by human inspectors are relatively low. Therefore, this tool will not replace human inspectors completely, but it will assist human inspectors to find out plagiarized source codes. Moreover, quality of the plagiarism detection process would be high, since a human inspector is involving in the plagiarism detection process.



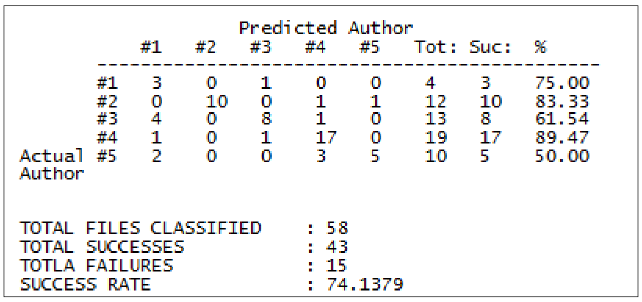
```
                                Predicted Author
             #1    #2    #3    #4    #5    #6    #7    #8    #9    #10   Tot: Suc:  %

     -----------------------------------------------------------------------------
        #1   16    0     0     0     0     0     0     0     1     0     17   16   94.12
        #2   0     238   4     0     0     1     0     1     0     4     248  238  95.97
        #3   0     0     25    0     0     0     0     1     2     0     28   25   89.29
        #4   0     1     0     19    0     0     0     0     0     0     20   19   95.00
Actual  #5   0     0     0     0     19    0     0     0     2     0     21   19   90.48
Author  #6   0     5     0     2     0     45    0     2     1     5     60   45   75.00
        #7   0     0     0     0     1     0     54    1     1     0     57   54   94.74
        #8   0     1     0     2     8     1     2     141   32    3     190  141  74.21
        #9   0     3     4     2     1     0     1     1     44    1     57   44   77.19
        #10  0     0     0     0     0     0     0     1     1     41    43   41   95.35

TOTAL FILES CLASSIFIED      : 741
TOTAL SUCCESSES             : 642
TOTAL FAILURES              : 99
SUCCESS RATE                : 86.6397
```

Figure 3. Confusion matrix of AdaBoost algorithm for hideout dataset

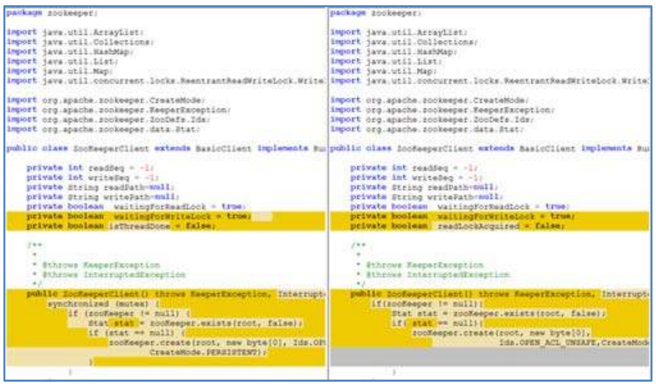
Furthermore, we tested the system with a small dataset, which consist of 133 Java source code files submitted by a group of five M.Sc. students for their programming assignments. The Result of the AdaBoost algorithm ran against the hideout dataset which consist of 58 files is shown in Figure 4. According to the output, AdaBoost algorithm classified 43 out of 58 source code files. Further investigation was carried out in order to find out plagiarized source code files, and we were able to find out two such incidences. One of them is shows in the Figure 5.



```
                 Predicted Author
           #1   #2   #3   #4   #5   Tot: Suc:   %
        ---------------------------------------------
       #1  3    0    1    0    0    4    3     75.00
       #2  0    10   0    1    1    12   10    83.33
       #3  4    0    8    1    0    13   8     61.54
       #4  1    0    1    17   0    19   17    89.47
Actual #5  2    0    0    3    5    10   5     50.00
Author


TOTAL FILES CLASSIFIED    : 58
TOTAL SUCCESSES           : 43
TOTLA FAILURES            : 15
SUCCESS RATE              : 74.1379
```

Figure 4. Confusion matrix of AdaBoost algorithm for samll hideout dataset



```
package zookeeper;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.concurrent.locks.ReentrantReadWriteLock.Write

import org.apache.zookeeper.CreateMode;
import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.ZooDefs.Ids;
import org.apache.zookeeper.data.Stat;

public class ZooKeeperClient extends BasicClient implements Ru

    private int readSeq = -1;
    private int writeSeq = -1;
    private String readPath=null;
    private String writePath=null;
    private boolean  waitingForReadLock = true;
    private boolean  waitingForWriteLock = true;
    private boolean isThreadDone = false;

    /**
     *
     * @throws KeeperException
     * @throws InterruptedException
     */
    public ZooKeeperClient() throws KeeperException, Interrupt
        synchronized (mutex) {
            if (zooKeeper != null) {
                Stat stat = zooKeeper.exists(root, false);
                if (stat == null) {
                    zooKeeper.create(root, new byte[0], Ids.OP
                            CreateMode.PERSISTENT);
                }
            }
```

```
package zookeeper;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.concurrent.locks.ReentrantReadWriteLock.Write

import org.apache.zookeeper.CreateMode;
import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.ZooDefs.Ids;
import org.apache.zookeeper.data.Stat;

public class ZooKeeperClient extends BasicClient implements Ru

    private int readSeq = -1;
    private int writeSeq = -1;
    private String readPath=null;
    private String writePath=null;
    private boolean  waitingForReadLock = true;
    private boolean  waitingForWriteLock = true;
    private boolean  readLockAcquired = false;

    /**
     *
     * @throws KeeperException
     * @throws InterruptedException
     */
    public ZooKeeperClient() throws KeeperException, Interrupt
        if(zooKeeper != null){
            Stat stat = zooKeeper.exists(root, false);
            if( stat == null){
                zooKeeper.create(root, new byte[0],
                                Ids.OPEN_ACL_UNSAFE,CreateMod

            }
```

Figure 5. Diff of two plagarized source code files

V. Conclusion and Future Works

In this paper we discussed machine learning based method for plagiarism detection. The main feature of our method is that, we have used a meta-learning algorithm in order to improve prediction accuracy of our system.

We were able to achieve 86.64 percent accuracy by using the same dataset used by Lange and Mancoridis[6]. According to the research paper published by Lange and Mancoridis[6] their accuracy was 55 percent. Moreover, we have shown that this method works with adequate accuracy for small training datasets.

In our research we investigated only three learning algorithms. However it is interesting to see how other learning algorithms work in the source code author identification problem. Furthermore, the AdaBoost in not the only meta-learning algorithm we can use for combining several weak-learners. In the future we will be investigating other learning algorithms for combining our weak-learners.

Our system will not work correctly, if programmers follow some coding standards and source code formatting tools specified in their projects. Our main target was programs written by students in programming courses. Moreover, it is unlikely that students follow some coding standards and use source code formatting tools, when doing their programming projects. Therefore, at this point this might not be a huge issue. However, in the future we will be improving our system in order to overcome this issue.

References

[1] J. Zobel, “Uni Cheats Racket: A case study in plagiarism investigation,” *Proceedings of the sixth conference on Australasian computing education-Volume 30*, 2004, pp. 357–365.

[2] C. Liu, C. Chen, J. Han, and P.S. Yu, “GPLAG: detection of software plagiarism by program dependence graph analysis,” *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006, p. 881.

[3] J.H. Ji, G. Woo, and H.G. Cho, “A source code linearization technique for detecting plagiarized programs,” *ACM SIGCSE Bulletin*, vol. 39, 2007, p. 77.

[4] C. Arwin and S.M.M. Tahaghoghi, “Plagiarism detection across programming languages,” *Proceedings of the 29th Australasian Computer Science Conference-Volume 48*, 2006, p. 286.

[5] S. Engels, V. Lakshmanan, and M. Craig, “Plagiarism detection using feature-based neural networks,” *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, 2007, p. 38.

[6] R.C. Lange and S. Mancoridis, “Using code metric histograms and genetic algorithms to perform author identification for software forensics,” *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, 2007, p. 2089.

[7] J.A.W. Faidhi and S.K. Robinson, “An empirical approach for detecting program similarity and plagiarism within a university programming environment,” *Computers & Education*, vol. 11, 1987, pp. 11–19.

[8] E. Alpaydin, *Introduction to Machine Learning, Second Edition*, The MIT Press, 2010.

[9] H. Ding and M.H. Samadzadeh, “Extraction of Java program fingerprints for software authorship identification,” *Journal of Systems and Software*, vol. 72, 2004, pp. 49–57.

[10] M.C. D, R. Prabhakar, and S. Hinrich, *Introduction to Information Retrieval*, Cambridge University Press, 2008.

[11] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 2002.

[12] R.E. Schapire, “A brief introduction to boosting,” *International Joint Conference on Artificial Intelligence*, 1999, pp. 1401–1406.

[13] “Developer Resources for Java Technology.”, http://www.oracle.com/technetwork/java/index.html [Accessed: Jan 25, 2011]

[14] T. Parr, The Definitive Antlr Reference: Building Domain-Specific Languages, Pragmatic Bookshelf, 2007

Upul Bandara received the B.Sc in Civil Engineering and M.Sc.in IT from University of Moratuwa, Sri Lanka in 2004 and 2011. His major research interests include machine learning, information retrieval and compiler construction. Presently, he works for *Virtusa* Corporation, Sri Lanka as a senior engineer.

Gamini Wijayarathna received Dr. Eng. degree (specializing in Software Engineering) and M.Eng. degree from the University of Electro-Communications, Tokyo, Japan. He graduated from the Faculty of Science, University of Kelaniya with an Honors degree in 1984.

He has received special training in System Engineering by the Center of the International Cooperation for Computerization (CICC) in Tokyo, Japan and IBM System 34/36 by IBM Sri Lanka.

He has over twenty years of experience in professional software development for various business and scientific applications, conducting research in software engineering related fields, and teaching Information Technology related subjects. His client base includes private and public sector organizations in Sri Lanka, and few Japanese companies.

Dr. Wijayarathna worked as a Research Associate at the Software Design Laboratory, Graduate School of Information Systems, University of Electro-Communications, Tokyo, Japan before he joined the Department of Industrial Management as a senior lecturer in year 2001. During last few years, he has extended his services to other institutes in Sri Lanka. Currently he is working as a consultant for a Japan based software development company in Sri Lanka, an external member for the Faculty of Applied Sciences at Rajarata University, and a visiting lecturer for the Postgraduate Institute of Science, University of Peradeniya and the Postgraduate Institute of Medicine, University of Colombo. He is also permanent resource personnel for National Institute of Education.