An Approach of Suspected Code Plagiarism Detection Based on XGBoost Incremental Learning

Qiubo Huang[1, a], Guozheng Fang[1, b] and Keyuan Jiang[2, c]

[1]Donghua University, Shanghai, China Shanghai, China

[2]Purdue University Northwest, USA Hammond, USA

[a]huangturbo@dhu.edu.cn, [b]hellofangguozheng@163.com, [c]kjiang@pnw.edu

Abstract. Code plagiarism is a serious problem in the teaching evaluation process, and the programming assignment is related to the student's grades. Therefore, it is especially important to detect code plagiarism submitted by students. As all the codes submitted are kept in the database, and the data are gradually accumulated day by day. In this case, we propose a detection approach based on relevant features and XGBoost incremental learning. First, we describe the definitions of the relevant features of the code submission record in the Online Judge system, as well as the algorithm details such as calculating code similarity, code style similarity and the level of concentration of plagiarism targets, etc. Then, we use information gain to filter out some irrelevant features, and use the performance metrics such as Accuracy, Macro F1-Score, AUC and ROC curve to select the learning model. Finally, the XGBoost incremental learning algorithm is used to optimize the system implementation, and the accuracy of the model is up to 97.9% during evaluation test.

Keywords: Code Plagiarism Detection, Relevant Features, XGBoost, Incremental Learning.

1. Introduction

Nowadays, with the rapid development of computer technology, a large number of online judge systems are pouring into the market, such as UVA, POJ, ZOJ and so on. The majority of them are set up for students with independent learning to prepare for the competition, and do not have the function of code plagiarism detection. However, in the teaching process, it is very important to evaluate the coding ability of students. Therefore, it seems extremely essential to introduce the approach of code plagiarism detection into the online judge system.

For these large amounts of source code plagiarism, it is difficult to detect without proper tool support. At present, various source code similarity detection systems have been developed to help detect source code plagiarism at home and abroad. These systems need to recognize many lexical and structural source code modifications. For example, the modification of the control structure, the modification of the data structure or the structural redesign of the source code [1]. When these structural modifications are applied to the source code, most existing code similarity detection systems may not recognize them. At the moment, in the academic research about the detection of suspected code plagiarism, the data-centric attribute counting methods are less studied [2, 3]. These methods use various measurable features extracted from the data. The extracted features are used as input to the source code plagiarism detection algorithm [4]. On the other hand, the code plagiarism detection system based on the data-centric attribute counting method does not depend on the structure of the source code. Therefore, they are not affected by the above problems. There are some papers that introduce the application of machine learning algorithms to the plagiarism detection system of attribute counting methods, but the accuracy is not satisfying [5].

Therefore,we propose a code plagiarism detection method based on relevant features and XGBoost incremental learning. It describes the definition of the relevant features of the code submission record in OJ (Online Judge) system, and the algorithm details. For example, when calculating the code similarity, we make use of the k-gram hash algorithm, and optimize the algorithm flow when calculating the hash value, so as to reduce the time complexity of the algorithm.When calculatingthe code style similarity, we take five features into account,

including spaces, indents, blank lines, braces' locations and comments in the code. We use edit distance to calculate the similarity of spaces, indents and blank lines, and use the longest common subsequence algorithm to calculate the similarity of the comments and the braces. The most important thing is that we proposed the degree of concentration of plagiarism objects. Finally, the XGBoost incremental learning algorithm elegantly solves the situation where the sample data is gradually added, and optimizes the final system implementation.

The remainder of the paper is organized as follows. We will first give the definition of the relevant features of the submitted code in OJ system, and the proposed algorithm details in Sec.2. We then use information gain scheme to filter out some weaker features, and adopt four indices including Accuracy, Macro F1-Score, AUC and ROC curve to evaluate the model in Sec.3. The implementation of the code plagiarism detection method based on XGBoost incremental learning are illustrated in Sec.4. Finally, we summarizes our work and puts forward some future research directions in Sec.5.

2. Candidate Features

Table 1 Description of Features

Feature Name	Detailed Description
MSR	The maximum similarity between a student's submitted code and others' codes
CPMS	Whether MSR exceeds the similarity threshold
CPMSPC	A category value for the distance between MSR and threshold
ASSR	Similarity of the code style between two source codes
WPSR	The similarity of spaces, indents, and blank lines between two source codes
BSR	The similarity of the braces between two source codes
CSR	The similarity of code comments between two source codes
PCR	The level of concentration of plagiarism targets
DL	The difficulty level of the problem
PR	The plagiarism rate of a student
RRB	Ranking of the student in an exam

According to the submitted code of the student and the related problem's information, we can ext ract the features described in Table 1. The separation of code and comments is performed when pro cessing each piece of code, then we can calculate the similarities between two source codes. The det ailed description of the source code feature attributes is shown in Table 1. The relevant definitions o f the feature attributes are elaborated in the following.

2.1 Feature MSR

The codes submitted by all the students of one problem are processed according to the following scheme:

We are currently processing the source code S_0 . All the submitted codes by others are in set $S = \{S_1, S_2, \cdots, S_n\}$, where S_i represents the code submitted by the ith student. $\mathrm{sim}(S_0, S_i)$ represents the similarity between two codes S_0 and S_i. When use a k-gram hash algorithm [6] to calculate $\mathrm{sim}(S_0, S_i)$ following the steps:

Step 1: Remove the line breaks, tabs, and extra spaces in each code, and replace the identifiers with simpler forms. For S_i, we can obtain S_i'

Step 2: Generate the k-gram collection KG_i from S_i'. A k-gram is a substring of length k for S_i'. If the length of S_i' is n, then KG_i will have $n - k + 1$ elements.

Step 3: Calculate the hash value of each element of KG_i and store the results in set U_i. The Karp-Rabin String Matching algorithm is used to calculate the hash value. Compared with the traditional hash function, it can reduce the time complexity from $O(kn)$ to $O(n)$, thus accelerating the calculation speed of the whole process[7]. The first element of KG_i contains a string $\{c_1 \cdots c_k\}$, and $H(c_1 \cdots c_k)$ is the hash function based on the constant$b$:

$$H(c_1 \cdots c_k) = c_1 * b^{k-1} + c_2 * b^{k-2} + \cdots + c_{k-1} * b + c_k \quad (1)$$

We will not use equation (1) to calculate the hash value $H(c_2 \cdots c_{k+1})$of the second string, and we will use the following formula as the substitute:

$$H(c_2 \cdots c_{k+1}) = \left(H(c_1 \cdots c_k) - c_1 * b^{k-1}\right) * b + c_{k+1} \quad (2)$$

For S_i', we can calculate $n - k + 1$ hash values, so set U_i has $n - k + 1$ elements.

Step 4: Calculate the similarity between S_0' and S_i':

$$sim(S_0, S_i) = sim(S_0', S_i') = \frac{|U_0 \cap U_i|}{|U_0 \cup U_i|} \quad (3)$$

Step 5: The calculation formula of MSR of S_0 can be described as:

$$MSR = \max(sim(S_0, S_1), sim(S_0, S_2), \cdots, sim(S_0, S_n)) \quad (4)$$

2.2 Feature CPMSPC

First we define CPMS: Whether the maximum similarity (MSR) of the student's code to others' codes exceeds the similarity threshold (SRT).

The similarity threshold (SRT) is set by the author of the problem according to the difficulty level of the problem, and the threshold can also be dynamically adjusted by a method of machine learning. Therefore, the algorithm logic of the CPMS is as follows. If the maximum similarity (MSR) is less than the similarity threshold (SRT), the CPMS is set to 0; otherwise, it is set to 1.

We also define CPMSP: A measure of the distance between CPMS and SRT. It can be calculated by:

$$CPMSP = \frac{|MSR - SRT|}{SRT} \quad (5)$$

Discretize CPMSP, then we get a category value, which is called CPMSPC. It can be calculated as following:

If CPMS equals 0, then:

If 0.5<CPMSP≤1, then CPMSPC=1;

If 0.2<CPMSP≤0.5, then CPMSPC=2;

If CPMSP≤0.2, then CPMSPC=3;

If CPMS equals 1, then:

If CPMSP≤0.2, then CPMSPC=4;

If 0.2<CPMSP≤0.5, then CPMSPC=5;

If 0.5<CPMSP≤1, then CPMSPC=6.

We can know that there are six kinds of CPMSPC category values.

2.3 Feature ASSR

Code style similarity (ASSR): Separate the code style information of two codes, such as spaces, indents, blank lines, curly braces, comments, etc. Then we can calculate the code style similarity ASSR.

1) Feature WPSR

The similarity of spaces, indents and blank lines (WPSR) is calculated using the edit distance ED, and includes the following steps:

Step 1: Separate the spaces, indents, blank lines and other information in the code, we can get WP, such as “blank line 1, space 3, space 4, indent 1”. Indicates that the code contains 1 consecutive blank line, 3 consecutive spaces, 4 consecutive spaces, and 1 consecutive tab character.

Step 2: Calculate the edit distance ED according to the spaces, indents and blank lines separated by the two codes. For example, the conversion results of the two codes are as follows.

WP1 = [blank line 1, space 3, space 4, indent 1]

WP2 = [blank line 1, space 4, indent 1, indent 1]

Because it changes from three spaces to four spaces, you need to "insert" a space. It also changes from four spaces to one indent, you need to "delete" three spaces, as well as "replace" a space with indent. The total operations required are one insert, three deletions and one replacement, so the edit distance is 5.

Step 3: Calculate the similarity using the edit distance ED.

$$\mathrm{WPSR} = 1 - \frac{\mathrm{ED}}{\max(\mathrm{SC}_1, \mathrm{SC}_2)} \tag{6}$$

Among them, SC_1 is the total amount of all the numbers in WP1. In the example, SC1=1+3+4+1=9, and SC2=1+4+1+1=7.

So WPSR between WP1 and WP2 is 1-5/(9+7)=11/16.

2) Feature BSR

The similarity of the braces in the codes (BSR) represents the similarity of two strings formed by the braces and their states in the two codes. Among them, we give four state definitions for braces. The first one shows that the brace is at the far left of a line of code, the second indicates that the brace is at the far right of a line of code, the third indicates that the brace is in the middle of a line of code, and the fourth one shows that the brace is on a separate line. The above four states are represented by 1, 2, 3, and 4, respectively. Therefore, we give the following calculation examples.

Step 1: We extract all the braces and their state from the two codes to form two strings. For example, a string extracted by a code is: "{2{2}1{2}4}4";

Step 2: For the above string, it is denoted by $\mathrm{S} = \{\mathrm{s}_1, \mathrm{s}_2, \cdots, \mathrm{s}_\mathrm{n}\}$, and then convert it into a sequence $\mathrm{X} = \{\mathrm{x}_1, \mathrm{x}_2, \cdots, \mathrm{x}_\mathrm{m}\}, \mathrm{m} = \frac{\mathrm{n}}{2}$, where $\mathrm{x}_\mathrm{i} = \{\mathrm{s}_{2*\mathrm{i}-1}, \mathrm{s}_{2*\mathrm{i}}\}$ and $\mathrm{i} \in \{1,2, \cdots, \mathrm{m}\}$. Next, calculate the length of the longest common subsequence of the sequences after the conversion of the two strings, denoted as C_L;

Step 3: Record the number of braces in the first string as C_1, and the number of braces in the second string as C_2. Then, give the formula for calculating the similarity of the braces in the codes as:

$$\mathrm{BSR} = 2 * \frac{\mathrm{C}_\mathrm{L}}{\mathrm{C}_1 + \mathrm{C}_2} \tag{7}$$

3) Feature CSR

The similarity of code comments (CSR) represents the similarity of two strings consisting of the status category values of the comments at each position in the two codes. We give three state definitions for comments. The first one indicates that this comment is on a separate line, and the second indicates that this comment is at the far right of a line of code, and the third is the other categories, which is represented by 1, 2, and 3 respectively. For the same reason, we use the same logic algorithm as BSR to calculate CSR.

After calculating WPSR, BSR and CSR, their average value can be obtained as ASSR:

$$\mathrm{ASSR} = \frac{\mathrm{WPSR}+\mathrm{BSR}+\mathrm{CSR}}{3} \tag{8}$$

2.4 Feature PCR

The level of concentration of plagiarism targets (PCR) is described as follows. Let n be the number of codes (to the problems) submitted by a student, then the set $\mathrm{R} = \{\mathrm{r}_1, \mathrm{r}_2, \cdots, \mathrm{r}_\mathrm{n}\}$ represents all the targets of the plagiarisms, where r_i represents the target (a student) of the i^{th} code plagiarized from. As a student can plagiarize more than one code from one student, so there will be duplications in R. After getting rid of duplications in R, we can get a new set $\mathrm{S} = \{\mathrm{s}_1, \mathrm{s}_2, \cdots, \mathrm{s}_\mathrm{m}\}$, $\mathrm{m} \leq \mathrm{n}$. Then the formula of PCR can be expressed as:

$$\mathrm{PCR} = 1 - \frac{\mathrm{m}-1}{\mathrm{n}} \tag{9}$$

If m equals to 1, which means the students copied all of his codes from one student, then PCR equals to 100%. This is the highest level of the concentration of plagiarism targets.

2.5 Feature DL

The difficulty level of the problem (DL): When creating a new problem, the teacher establishes the difficulty level of the problem or attaches a corresponding label according to his teaching experience. In the OJ system, three levels including easy, medium, and difficult are used to indicate the difficulty of the problem. In the subsequent data processing, the difficulty values are expressed in the form of 1, 2, and 3, respectively.

2.6 Feature PR

The plagiarism rate of the student (PR): We count the number of codes that a student is labeled to be plagiarized, and it is denoted as TNPE. Therefore, we can calculate the plagiarism rate of the student. $PR = TNPE/n$, where n is the number of codes he ever submitted.

2.7 Feature RRB

Ranking of the student (RRB): Each time the student completes a solution (to a problem), the system will calculate his current total score in real time, and then update his rank according to his total score. Since students' rank were in a descending order, we can take advantage of the basic idea of insertion sorting to implement the algorithm in order to reduce the complexity in the ranking process.

3. Data Processing Based on Information Gain

First, we extract some feature datasets from the raw data and their tag data in the database, into the initial sample dataset. Then, we use the filter selection method to filter out some weaker features from the initial sample set to reduce the difficulty of subsequent learner learning tasks. We use information gain to measure the importance of features. If the information gain based on a feature of the sample set is larger, it means that the feature contains more information that contributes to the algorithm. Thus, we calculate the information gain of each candidate feature based on the sample set, sorted by the value of the information gain. Finally, the feature whose information gain is greater than the threshold is selected.

Next, we use the dichotomy to discretize some continuous feature attribute values. The method is described below.

Assume we have a given sample data set D, and have a continuous attribute in the data set D, the values in this continuous attribute are sorted in ascending order and their values are combined into a set $C = \{c_1, c_2, \cdots, c_n\}$. By taking the average of any two adjacent elements c_i and c_{i+1} in the set C as the dividing point, the definition of the candidate dividing point set is given $T_c = \left\{\frac{c_i + c_{i+1}}{2} \middle| 1 \le i \le n-1\right\}$, and then the information gain of the data set after division is obtained. The information gain in all possible division cases are compared, so we choose the dividing point that maximizes the information gain. Then, we repeats the above process by selecting a set with the largest information entropy from the obtained data set, until the number of sets reaches the user-specified number or the specified termination condition.

"Information entropy" is the most commonly used indicator for measuring the purity of a sample set. Assuming that the proportion of the k^{th} label sample in the set D is $p_k (k = 1,2,\cdots,m)$, the information entropy of the sample D is defined as [8]:

$$Ent(D) = -\sum_{k=1}^{m} p_k \log p_k \tag{10}$$

Suppose there is a division point $t(t \epsilon T_c)$, which divides the sample set D into D_1^t and D_2^t, where D_1^t includes the samples whose continuous attribute C is not greater than t, and D_2^t includes others. According to the difference of the size of samples, each partitioned sample is given the corresponding weight $|D_i^t|/|D|, i \in \{1,2\}$. Then, the calculation formula of the maximum information gain of all the partitions of D is as following [8]:

$$Gain(D, C) = \max_{t \epsilon T_c} \left(Ent(D) - \sum_{i \in \{1,2\}} \frac{|D_i^t|}{|D|} Ent(D_i^t) \right) \tag{11}$$

4. Incremental Learning Based on Xgboost

In the actual database, the amount of data is gradually increasing. In particular, during an exam, a large amount of new data are generated. These new data often have great values, with recent user plagiarism information and behaviors that were not available in previous data. How to effectively use these new data, this will be solved in this section.

In the field of machine learning, it is very difficult to use traditional batch learning techniques to get useful information from the ever-increasing amount of new data. As the size of the data

continues to increase, the demand for time and space will increase rapidly, and eventually the speed of learning will not keep up with the speed of data updates. It is a widely used intelligent data mining and knowledge discovery technology compared with Incremental Learning. It is also a learning system that can continuously acquire new knowledge from new sample data, and can save a large amount of the knowledge that has been learned before. As the sample data is gradually accumulated, the learning accuracy will also increase. Therefore, incremental learning is an effective way to solve this problem.

The XGBoost toolkit in the Python environment is used in this paper. We can do the work as following:

Step 1: Import the sample dataset into memory. The sample dataset will be divided into training and testing set.

Step 2: Convert the training set to XGBoost's own data structure, DMatrix, to speed up subsequent calculations.

Step 3: Adjust the corresponding model parameters and learn the training.

Step 4: Save the model generated by each training into memory, or a binary file. When the next new sample data are generated, we can train the new sample based on the original model, which can produce a good new model under the condition of less time.

5. Experimental Results and Analysis

Now, we have processed the sample data set, and then divided it into training set and testing set. In order to understand the distribution of training data intuitively, we use PCA to reduce the dimensionality of the feature data in the training set, and then use Python Matplotlib and mplot3d toolkits to visualize the data, as shown in Fig. 1.

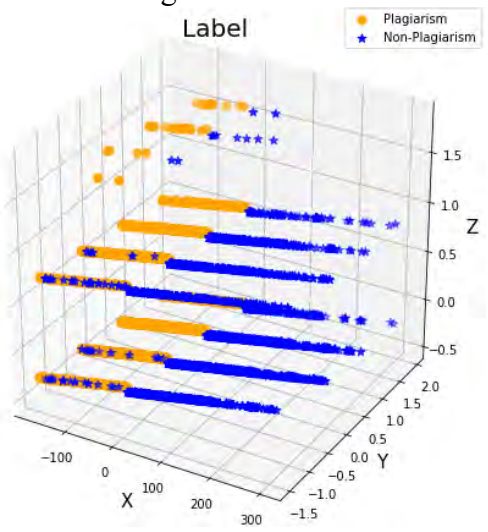

Fig. 1 Distribution map of training data

Furthermore, we use SVM, GBDT and XGBoost machine learning algorithms to learn the training set, then analyze the experimental results and evaluate the algorithm model. Their corresponding performance metrics are shown in Table 2.

Table 2 Evaluation of svm, gbdt and xgboost

Evaluation	Algorithm		
	SVM	GBDT	XGBOOST
Accuracy	0.9761	0.9780	0.9790
macro-P	0.9760	0.9781	0.9791
macro-R	0.9761	0.9781	0.9791
macro-F1	0.9761	0.9780	0.9790
AUC	0.9761	0.9781	0.9791

It is not difficult to know from Table 2 that the performance metrics of the XGBoost model are larger than the other two machine learning models. The results of Macro F1-Score are especially

important, so it is preliminarily concluded that XGBoost is better than the other two. In order to more accurately analyze the experimental results and select the optimal model, we draw the ROC curve and compare the machine learning model based on the ROC curve, as shown in Fig. 2 and 3. It is not difficult to know that the ROC curve of the XGBoost model has completely "encased" the other two models, so it is also basically concluded that the XGBoost model outperforms the other two models. However, some people think that there may be coincidences and intersections between them by looking at Fig. 3, then we will further calculate the area under the ROC curve, AUC (Area Under ROC Curve), as shown in Table 2. The AUC value of XGBoost is greater than the values of the other two models, so its performance is still better than the other two models.

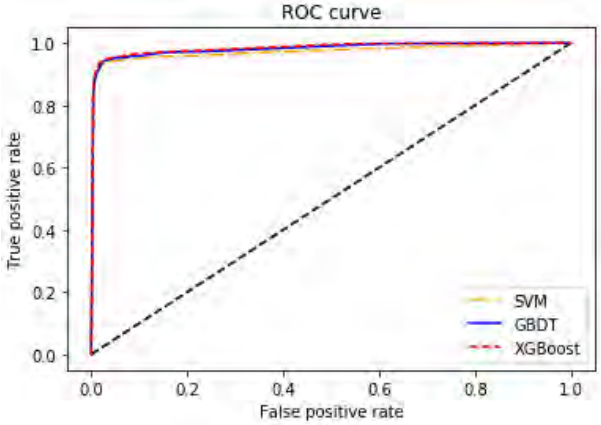

Fig. 2 ROC Curve

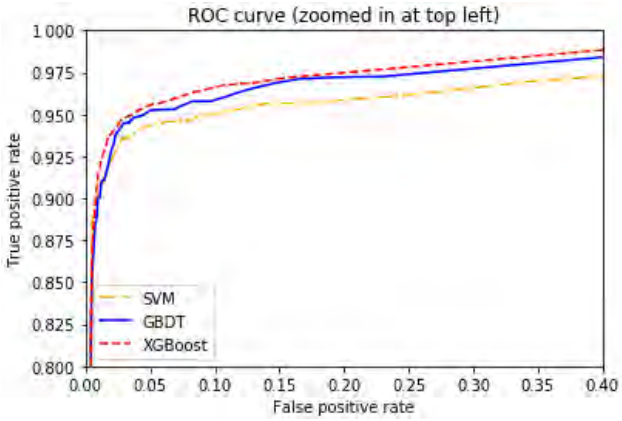

Fig. 3 ROC Curve (zoomed in at top left)

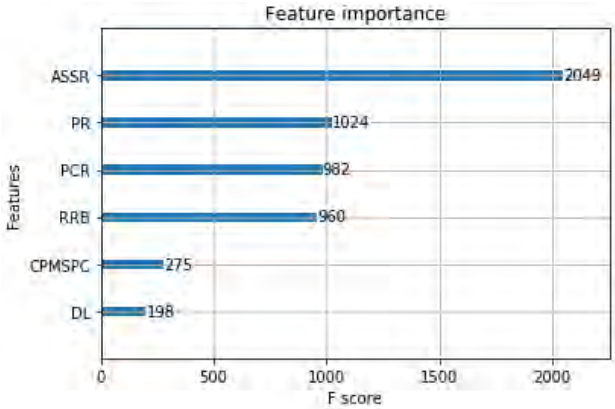

Fig. 4 XGBoost Feature Importance

Finally, the analysis shows that the XGBoost algorithm has the highest classification accuracy and the best performance compared with the other two models. Therefore, a comparison chart of the importance of features based on the XGBoost model is given, as shown in Fig. 4. XGBoost is a highly scalable, end-to-end tree boosting model. It also take advantage of a new sparsity-aware

algorithm for sparse data and weighted quantile sketch for approximate tree learning. More importantly, it also provides methods for cache access patterns, data compression and building of scalable tree boosting system in parallel [9]. Therefore, XGBoost is undoubtedly the best choice to solve our problem.

6. Conclusion

In this paper, we describe the definition of the relevant features of the submitted code in Online Judge system, and their calculation algorithm details. Such as code similarity, code style similarity and the level of concentration of plagiarism targets, etc. Respectively, they have adopted efficient algorithm to achieve high efficiency. Then, we use information gain to filter out some weaker features, and use the performance metrics such as Accuracy, Macro F1-Score, AUC and ROC curve to choose the model. Finally, the XGBoost incremental learning algorithm is chosen to optimize the system implementation. However, academic research has not been enough so far, and we still need to continue to strengthen the research on machine learning algorithms, and further advances in theoretical research.

Acknowledgments

We would like to thank all the graduate students who contributed much to the OJ system, and they help make this work possible.

References

[1]. Z. Djuric, and D. Gasevic, “A Source Code Similarity System for Plagiarism Detection”, The Computer Journal, vol. 56, pp. 70-86, 2013.

[2]. 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.

[3]. 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.

[4]. 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.

[5]. U. Bandara, G. Wijayarathna, “A machine learning based tool for source code plagiarism detection”, International Journal of Machine Learning and Computing, vol. 1, no. 4, 2011.

[6]. S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In ACM, editor, Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003, San Diego, California, June 09–12, 2003, pages 76–85, New York, NY 10036, USA, 2003. ACM Press.

[7]. Richard M. Karp and Michael O. Rabin. Pattern-matching algorithms. IBM Journal of Research and Development, 31(2):249–260, 1987.

[8]. Raileanu, L.E. and Stoffel, K. (2004) Theoretical comparison between the Gini Index and Information Gain criteria. Ann. Math. Artif. Intell. 41, 77–93.

[9]. T. Chen and C. Guestrin. XGBoost: A scalable tree boosting system. In Proceedings of the 22nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining, pages 785–794, San Francisco, CA, 2016.