Kaliuzhna, Tetiana; Kubiuk, Yevhenii

Article

Analysis of machine learning methods in the task of searching duplicates in the software code

Reference: Kaliuzhna, Tetiana/Kubiuk, Yevhenii (2022). Analysis of machine learning methods in the task of searching duplicates in the software code. In: Technology audit and production reserves 4 (2/66), S. 6 - 13. http://journals.uran.ua/tarp/article/download/263235/260162/608329. doi:10.15587/2706-5448.2022.263235.

UDC 004.85:004.032.2
DOI: 10.15587/2706-5448.2022.263235
Article type «Original research»

Tetiana Kaliuzhna,
Yevhenii Kubiuk

ANALYSIS OF MACHINE LEARNING METHODS IN THE TASK OF SEARCHING DUPLICATES IN THE SOFTWARE CODE

The object of the study is code in the Python programming language, analyzed by machine learning methods to identify clones.

This work is devoted to the study of machine learning methods and implementation of the decision tree machine learning model in the problem of finding clones in the program code. The paper also analyzes existing machine learning approaches for detecting duplicates in program code. During the comparison, the advantages and disadvantages of each algorithm were determined, and the results were summarized in the corresponding comparison tables. As a result of the analysis, it was determined that the method based on the decision tree, which gives the best result in the task of finding clones in the program code, is the most optimal both from the point of view of accuracy and from the point of view of implementation.

The result of the work is a created model that, with an accuracy of more than 99 %, classifies cloned and non-cloned codes on an automatically generated dataset in a minimal amount of time. This system has several open questions for future research, the list of which is presented in this work. The proposed model has the following ways of further development:

– recognition of clones rewritten from one programming language to another;

– detection of vulnerabilities in the code;

– improvement of model performance by creating more universal datasets.

The perspective of the work lies in training a decision tree model for accurate and fast detection of code clones, which can potentially be widely used for plagiarism detection in both educational institutions and IT companies.

Keywords: *clone detection, machine learning methods, decision tree, Support Vector Machine, TECCD, dataset.*

Received date: 29.06.2022
Accepted date: 12.08.2022
Published date: 26.08.2022

How to cite
Kaliuzhna, T., Kubiuk, Y. (2022). Analysis of machine learning methods in the task of searching duplicates in the software code. Technology Audit and Production Reserves, 4 (2 (66)), 6–13. doi: http://doi.org/10.15587/2706-5448.2022.263235

1. Introduction

The detection of duplicates in the program code, or the detection of clones, is a very active field of research in recent years. Code duplication has been recognized as a potentially serious problem that negatively affects the maintainability, understandability, and development of software systems.

Source code that is used in an identical form multiple times in software is called a duplicate code or source code clone. An automated process that helps find clones in source code is called clone detection. Similar sections or pieces of code are also considered duplicates, and even code sequences that are only functionally identical can be considered duplicate code. Duplicate code occurs most often when existing functions are copied from one place in the program code to another.

Cloning is considered defect-prone because inconsistent changes to code clones can lead to unexpected program behavior. It is important to understand that clones do not directly cause errors, but inconsistent changes to clones can lead to unexpected program behavior. Inconsistent bug fixing is a particularly dangerous type of change to cloned code.

Many software engineering tasks, such as plagiarism detection, code quality analysis, bug detection, vulnerability detection, etc., may require the detection of semantically or syntactically similar blocks of code. This makes clone detection an effective and useful part of software analysis [1].

Code cloning can occur in any software project. One problem is that code clones come in many different types, making them difficult to detect using standard patterns. Due to the diversity in the structure and form of semantically similar clones, machine learning methods are required to detect them.

2. The object of research and its technological audit

The object of research is code in the Python programming language, analyzed by machine learning methods to identify clones.

Duplicate code makes the overall code extremely difficult to maintain and the codebase becomes unnecessarily large, leading to technical debt. In work [2] it was determined that the main reasons for eliminating duplicates in the program code are:

1. Reduction of maintenance costs.

A clone of the source code must be found and processed at each point of use.

2. To facilitate error correction.

Source clones may go unnoticed, and with them, copied errors. This leads to uncoordinated changes in the future.

3. Minimization of memory requirements.

Cloning increases the amount of code and therefore the amount of memory required. Well-written code without duplication ensures that the program will take up less space.

4. Ease of code readability.

5. Code execution speed.

Code clones increase the time it takes to compile the code. Every millisecond of delay will contribute to greater latency and greater memory requirements on the user's local machine as well as on the production servers.

It is important to note that clones, depending on the degree of changes in the code, are divided into 4 types [3]:

– Type-1 clone: the code snippets are identical except for small changes in spaces and comments.

– Type-2 clone: code fragments are structurally and syntactically identical, only user-defined identifiers such as variable, type, or function names and comments change.

– Type-3 clone: based on Type-1 and Type-2, there are other changes to the copied segment, such as modification, insertion or deletion of operators.

– Type-4 clone: code fragments are semantically similar, i. e. perform similar functions, but syntactically differ.

Among the four types of clones, Type-1, Type-2, and Type-3 are syntax-based code clones. Type-4 is a semantic clone of the code, which indicates that the code performs similar functions, but the syntactic structures are different.

Early detection and removal of duplicate lines of code simplifies code structure and reduces file size, also increases code maintainability and reduces technical debt over time. Removing duplicates keeps the code clean, which in turn helps deliver feature support and updates faster. Unpatched code clones represent hidden bugs, and for critical security issues, hidden vulnerabilities, so it's important to detect them quickly. Also, to prevent violation of the rules of academic integrity and violation of copyright, the topic of finding duplicates is very important.

3. The aim and objectives of research

The work aims is to find duplicates in the program code.

The objectives of this research are:

1. Review of existing methods and algorithms of machine learning, which are used in the task of finding duplicates in the program code, and conducting their comparative analysis.

2. Development of a machine learning model, which will be used for further implementation in the task of detecting clones in the code.

3. Verification of the developed software product in practice.

4. Research of existing solutions of the problem

In this work, three methods of machine learning for detecting clones were considered:

1) decision tree model of machine learning [4];

2) method using Support Vector Machine (SVM) [5];

3) method using A Tree Embedding Approach for Code Clone Detection (TECCD) [6].

The working principles of the studied methods:

1. Method using a decision tree machine learning model (Decision Tree).

A decision tree is a supervised learning algorithm that generates decision nodes using the information obtained from the value of each function [7]. It can be represented as a tree-like graph model consisting of several levels of nodes representing a decision rule. Classification is performed by passing data through the tree from the top to the end node. At each decision node, a branch is selected based on the value of the corresponding attribute. A significant advantage of the decision tree model over other models is that it provides an interpretable result [8].

While investigating the decision tree machine learning model, the authors considered an approach to improve existing code clone detection tools using machine learning techniques. The paper investigates 19 clone class metrics to capture various characteristics of code clones and use them to train a decision tree model. The trained decision tree model is then used as a filter to remove false clone classes from the cloning result.

The authors of the paper noted that applying a decision tree cloning filter trained on Java clones to Python clones showed that the filter was not effective in another language, and further work is needed on this issue.

2. Method using Support Vector Machine.

SVM in this approach provides clone class identification by classifying code clones into one of the appropriate classes.

In the study using SVM, the main focus was on the technique of finding similar blocks of code and quantifying their similarity (Type-3 clones). Code clones are detected in two stages. At the first stage, a parser is used to create sets of functions. In this work, SVM is the machine learning tool and *.c files are the input. Feature sets are converted to Libsvm format by assigning 0 and –1 to the class labels of sorting and unsorting algorithms, respectively. In the second step, the feature sets used as input are passed through the LibSVM tool to classify the code fragment. Scaling is performed on both the training and test datasets. The output of the second stage is the label of the test data set and the accuracy obtained by the tool.

The authors noted that the accuracy increases with the number of instances. The accuracy for 45 instances is 93.182 %.

3. A Tree Embeddings Approach for Code Clone Detection.

The approach is to convert the source code into an abstract syntax tree (AST) [9], which contains information about the structure of the code. The AST is then mapped to a vector based on machine learning techniques and the Euclidean distance of the vectors is compared to detect code clones.

The TECCD technique is based on tree embedding for code clone detection. This approach first performs an embedding tree to obtain a node vector for each intermediate

ISSN 2664-9969

node in the AST. Embedding nodes capture information about the context/structure of the AST. Then a tree vector is created from its node vectors using a simplified method [10]. Finally, Euclidean distances between tree vectors are measured to identify code clones.

This approach is implemented in a tool called TECCD, and the evaluation was carried out with 7 large Java projects, as well as BigCloneBench [11]. The results showed that this tool is quite effective in terms of accuracy and completeness [12].

4. Comparison of methods.

The following is a comparative analysis of the considered algorithms (Tables 1, 2).

Table 1

Comparative characteristics of methods

Method	Precision	Data pre-processing
Decision Tree	98 %	Small (remove spaces)
Support Vector Machine	93.182 %	Average (transformation of feature sets into LibSVM format)
A Tree Embedding Approach for Code Clone Detection	88 %	Large (Generation AST)

Table 2

The main advantages and disadvantages of the methods

Method	Advantages	Disadvantages
Decision Tree	– short learning time; – take into account every possible outcome of the decision and trace each node to the conclusion accordingly; – can process large-dimensional data with good accuracy; – small data preparation; – simple and reliable	– sensitive to noise in input data; – spaces in the data are difficult to maintain; – small data changes can significantly change the constructed decision tree
Support Vector Machine	– sensitive to noise in input data; – spaces in the data are difficult to maintain; –small data changes can significantly change the constructed decision tree	– unstable to noise in the source data; – when trying to use it in multi-class classification, the quality and speed of work decrease; – not suitable for large datasets
A Tree Embedding Approach for Code Clone Detection	– fairly high accuracy of results on large volumes of data; – demonstrates good indicators of accuracy and completeness; – consistently high ability to detect Type-1 and Type-2 clones	– requires complex preliminary data preparation; – requires a large set of training data; – the algorithm is quite difficult to implement; – a lot of time is spent on AST generation

After a detailed comparison with Tables 1, 2, it can be seen that the method based on the decision tree is optimal, both from the point of view of accuracy and from the point of view of implementation.

5. Methods of research

In view of the above comparison of methods, the task of this research is the implementation of the machine learning method using the decision tree model, which is the most optimal from the point of view of the considered characteristics of the methods. So, let's develop an authentic implementation of this method to improve its effectiveness in the task of finding duplicates in the program code. The data supplied to the input of the model are presented in the form of a numerical vector, with the metrics described in Table 3.

Table 3

Information about metrics

The name of the metric	Sense of metrics	Formula
volume_ratio	ratio of code volumes	$\frac{n\_tokens_1}{n\_tokens_2}$
max_fragment_ratio	the ratio of the volumes of the maximum code blocks	$\frac{\max\_fragment\_lines_1}{\max\_fragment\_lines_2}$
min_fragment_ratio	the ratio of volumes of minimal blocks of codes	$\frac{\min\_fragment\_lines_1}{\min\_fragment\_lines_2}$
clone_tokens_to_min_tokens_ratio	the ratio of the number of common tokens to the number of tokens of a smaller code	$\frac{P(tokens_1 \cap tokens_2)}{\min\left(P\left(tokens_1\right), P\left(tokens_2\right)\right)}$
lines_ratio	the ratio of the number of common tokens to the number of smaller code tokens	$\frac{n\_lines_1}{n\_lines_2}$
operators_overlap_ratio	relation of intersections of operators	$1-\frac{\sum_{i=1}^{n}\left\|n\_op_1^i - n\_op_2^i\right\|}{\max\left(n\_op_1, n\_op_2\right)}$

A decision tree or classification tree (DT) is a supervised learning technique that can be used for both classification and regression problems, but is mostly better for solving classification problems. It is a tree-like classifier where internal nodes represent features of the data set, branches represent decision rules, and each leaf node represents an outcome. The goal is to create a model that can predict the value of the target variable while learning simple decision rules derived from the characteristics of the data.

In Fig. 1 [13] shows the diagram of the general structure of the decision tree.

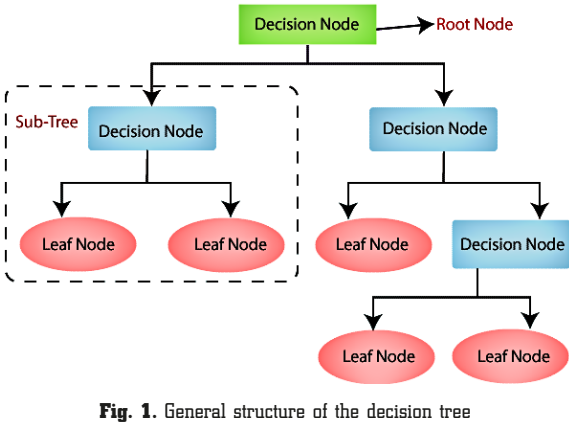

Fig. 1. General structure of the decision tree

DT creates a classification model by constructing a decision tree. Each tree node defines an attribute check, each branch descending from this node corresponds to one of the possible values for this attribute [14].

In Fig. 2 shows the High-Level Diagram (HLD) of our software product. HLD demonstrates the structure of the developed software product:

1) Code Samples – Python scripts from which a dataset is generated;
2) Data Generating Script – data generating script;
3) Dataset – actual dataset;
4) Training Script – a script that trains the model;
5) Model – decision tree model – binary file;
6) Classification Script – a script that loads the model and classifies the received codes using it (takes two codes, extracts metrics, submits to the model, receives 1 or 0 (clones or not clones));
7) Flask App – a web application that receives files, transfers them to Class Script and shows the result to the user (6 and 7 are one docker container);
8) Code – are code files that we want to classify as clones or non-clones;
9) Classification Results – received results.

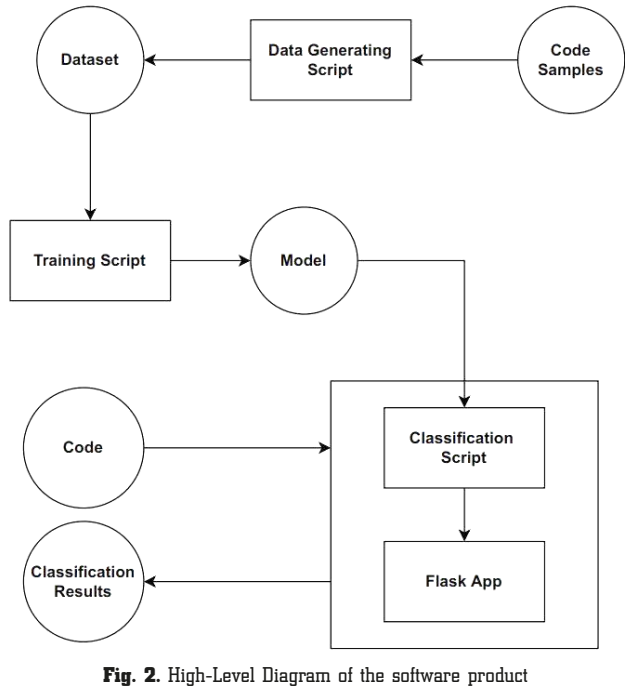

Fig. 2. High-Level Diagram of the software product

The implementation of the decision tree is developed in the Python programming language. This programming language is widely used in many areas, as there are many modules and libraries for it that can be used to create any software product. The implementation process follows the following algorithm:

– pre-processing of data;
– model training on the training sample;
– checking the accuracy of the result (creating a matrix of confusion matrix);
– visualization of the result of the test set.

5.1. Dataset creating. Since there were no marked datasets in public access, and their manual marking would take a long time, a decision was made to automatically generate the dataset.

Another 80 clones were generated from 20 Python scripts.

The generation took place as follows: changes are made to the code 35 times. The change is renaming a random variable with a probability 0.2, also changing the sequence of a random block of code with a probability 0.2, and removing or adding a random line with a probability 0.6.

A total of 100 scripts were received, 5 variants of each of 20 scripts. In Fig. 3, 4 show variants of clone codes.

As a result, it could be and was generated $\left(C_5^2+5\right)\cdot 20=$ $=300$ pairs of clone codes (in the formula add 5 more options since a pair of codes with itself is also suitable for the dataset).

For the balance of the dataset (equal numbers of representatives of the False and True classes), 300 pairs of codes that are not clones were also selected.

The following metrics (Table 3) were selected from pairs of codes for logical and intuitive reasons. In formulas:

n_tokens_i – the number of tokens (words) in code i, $i=\{1,2\}$;
$\max\_fragment\_lines_i$ – the number of lines in the largest block of code i, $i=\{1,2\}$;
$\min\_fragment\_lines_i$ – the number of lines in the smallest block of code i, $i=\{1,2\}$;
$tokens_1$ – set of code tokens (words) i, $i=\{1,2\}$;
n_lines_i – the number of lines in code i, $i=\{1,2\}$;
$n\_op_j^i$ – the number of type operators i (+, – ,...), $i=\{1,$ number of operator types$\}$ in the code j, $j=\{1,2\}$;
n_op_i – the total number of statements in code i, $i=\{1,2\}$.

The above metrics + the Clone field (True, if yes; False, if not) make up the dataset (Table 4).

```python
from django.db import models
from django.utils import timezone
# from datetime import date
# Create your models here.
class Todo(models.Model):
    description = models.CharField(max_length=200) #task description
    day = models.DateField(default = timezone.now()) #day user selected for this task
    user = models.ForeignKey('auth.User', related_name='todos', on_delete=models.CASCADE) #reference to user
    status = models.BooleanField(default=False) #done or not done
    def __str__(self):
        return self.description

class ItemList(models.Model):
    title = models.CharField(max_length=300)
    user = models.ForeignKey('auth.User', related_name='lists', on_delete=models.CASCADE) #reference to user
    def __str__(self):
        return self.title

class Item(models.Model):
    description = models.CharField(max_length=200) #task description
    user = models.ForeignKey('auth.User', related_name='item_lists', on_delete=models.CASCADE) #reference to user
    status = models.BooleanField(default=False) #done or not done
    list_rel = models.ForeignKey('J2D_api.ItemList', related_name='items', on_delete=models.CASCADE)
    def __str__(self):
        return self.description
```

Fig. 3. Source code

ISSN 2664-9969

```python
class Item(models.Model):
from rxwbvzmf.db import models
    def __str__(self):
from rxwbvzmf.utils import lrknacihqbayrmw
# from datetime import date
# evrnncxuzthcpvjjaos your models here.
class Item(models.Model):
    description = models.CharField(max_length=200) #task description
    user = models.ForeignKey('auth.User', vdukgwbidkip='item_lists', on_delete=models.oxrervbbiwsdn) #reference to user
    ajumsfrcr = models.BooleanField(default=False) #done or not done
    ajumsfrcr = models.BooleanField(default=False) #done or not done
    ajumsfrcr = models.BooleanField(default=False) #done or not done
    list_rel = models.ForeignKey('J2D_api.rvtzppiaihc', vdukgwbidkip='items', on_delete=models.oxrervbbiwsdn)
    description = models.CharField(max_length=200) #task description
    day = models.DateField(default = lrknacihqbayrmw.now()) #day user selected for this task
    user = models.ForeignKey('auth.User', vdukgwbidkip='todos', on_delete=models.oxrervbbiwsdn) #reference to user
    ajumsfrcr = models.BooleanField(default=False) #done or not done
    def __str__(self):
        return self.description
        return self.description
        return self.title
        return self.title
    title = models.CharField(max_length=300)
    user = models.ForeignKey('auth.User', vdukgwbidkip='lists', on_delete=models.oxrervbbiwsdn) #reference to user
    def __str__(self):
    def __str__(self):
        return self.description
class Todo(models.Model):
class Item(models.Model):

class rvtzppiaihc(models.Model):

```

Fig. 4. Source code clone (automatically generated)

Table 4

Dataset metrics

volume_ratio	max_fragment_ratio	min_fragment_ratio	clone_tokens_to_min_tokens_ratio	lines_ratio	operators_overlap_ratio	Clone
1	1.024	1	0.5950920245	1.024	1	True
1	0.9926470588	1	0.6666666667	0.9926470588	1	True

5.2. Learning and tree building parameters. Decision tree learning options:

– *Quality criterion: Ginny.*

The separation quality measurement function works according to the Gini criterion:

$$H(X)=\sum_{k=1}^{K} p_k(1-p_k)=1-\sum_{k=1}^{K} p_k^2, \quad (1)$$

$$p_k=\frac{1}{|X|}\sum_{i\in X}[y_i=k], \quad (2)$$

where p_k – the probability of appearance of class k in the sample.

– *Maximum depth:* there was no limit to the maximum depth when building the tree.

– *The minimum number of objects in a tree leaf*: is 2 objects.

– *Separation strategy*: as already mentioned, the Gini test was used. The partitioning was chosen in such a way as to maximize the amount of information in the samples going into both subtrees.

6. Research results

As the result, a classifier model was trained (Fig. 5).

It was able to classify data from the test and training datasets with high precision.

A tree of depth 6 was obtained. The results of running the tree on the training dataset are checked using the confusion matrix.

The confusion matrix summarizes the classification efficiency of the classifier concerning some test data [15]. It is a two-dimensional matrix indexed in one dimension by the true class of the object and in the other by the class assigned by the classifier. Table 5 shows an example of a discrepancy matrix for a three-class classification problem with classes *A*, *B*, and *C*.

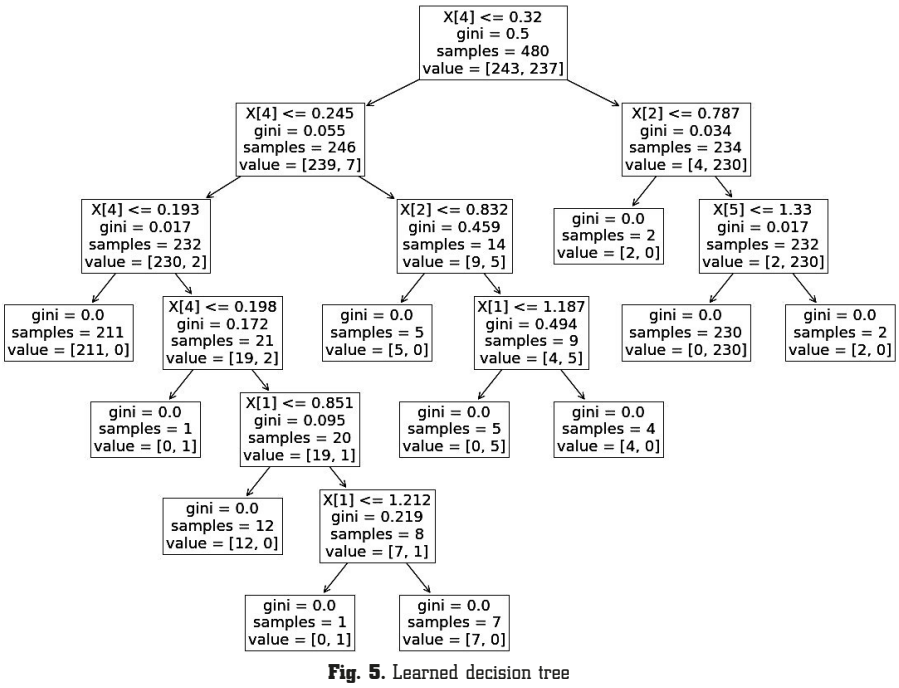

Fig. 5. Learned decision tree

Table 5

An example of a three-class confusion matrix

Classes		Assigned class		
		A	*B*	*C*
Actual class	*A*	10	2	1
	B	0	6	1
	C	0	3	8

The first row of the matrix indicates that 13 objects belong to class *A* and that 10 are correctly classified as belonging to *A*, two incorrectly classified as belonging to B and one as belonging to *C*.

A special case of the discrepancy matrix is often used with two classes, one representing the positive class and the other the negative class. In this context, the four cells of the matrix are designated as true positive (*TP*), false positive (*FP*), true negative (*TN*) and 7 false negative (*FN*), as indicated in Table 6.

Table 6

Structure of confusion matrix

Classes		Assigned class	
		Positive	Negative
Actual class	Positive	*TP*	*FN*
	Negative	*FP*	*TN*

The result of the classification of the decision tree in the task of finding clones in the program code is shown in the confusion matrix (Fig. 6).

A number of classification performance indicators are defined in terms of these four classification results:

$Specificity = True\ negative\ rate = TN/(TN+FP) = 54/(54+0) = 1.$

$Sensitivity = True\ positive\ rate = Recall = TP/(TP+FN) = 65/(65+1) = 0.985.$

$Positive\ predictive\ value = Precision = TP/(TP+FP) = 65/(65+0) = 1.$

$Negative\ predictive\ value = TN/(TN+FN) = 54/(54+1) = 0.982.$

That is, let's conclude that only 1 out of 120 examples were classified incorrectly under the condition of balanced classes (60 by 60). That is, the accuracy of forecasting on such a dataset is more than 0.99.

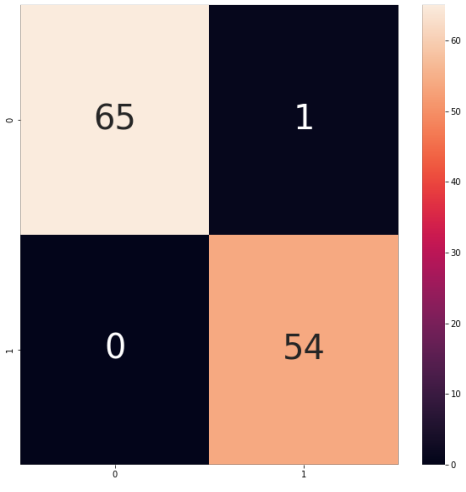

Fig. 6. Matrix of inconsistencies

Experiments

1. Detection of plagiarism

With the help of the trained model of the decision tree, let's conduct an experiment to detect plagiarism in real works of students (to preserve confidentiality, the surnames have been changed, any coincidences are accidental).

Let's upload to the program 20 laboratory works No. 1 of students from one discipline (Fig. 7).

After testing, the program showed that konoplianka.txt is a clone of the popov.txt code, while popov.txt is a clone of the zinchenko.txt code, but the konoplianka.txt code is not a clone of the zinchenko.txt code. This can be explained by the fact that the trained model did not learn transitive property.

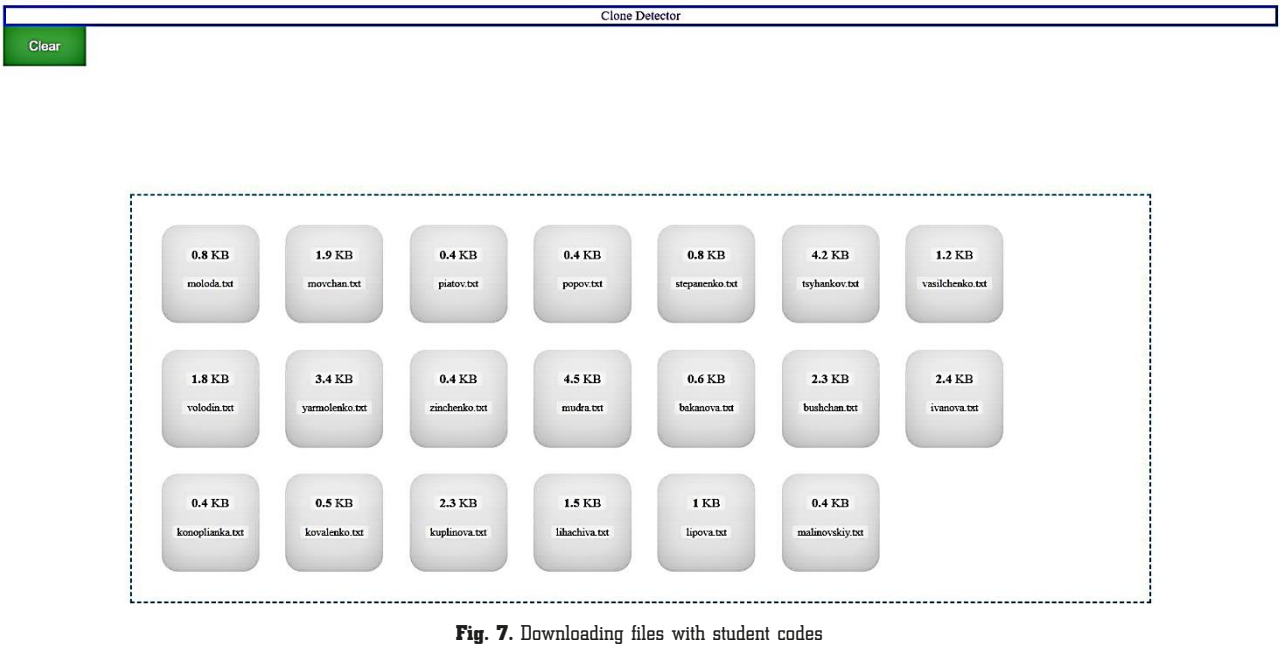

Fig. 7. Downloading files with student codes

ISSN 2664-9969

No more coincidences between laboratory works were found, which indicates a high level of compliance with academic integrity by students, because at least 17 out of 20 works are unique.

So, let's test our model for the ability to detect a typical vulnerability in the code. Let's write the code where the value of the environment variable is manually set (Fig. 8), which is a typical example of the disclosure of sensitive data. Let's also create an example code containing only this vulnerability.

Another common vulnerability is a buffer overflow. An example of this vulnerability can be seen in Fig. 9.

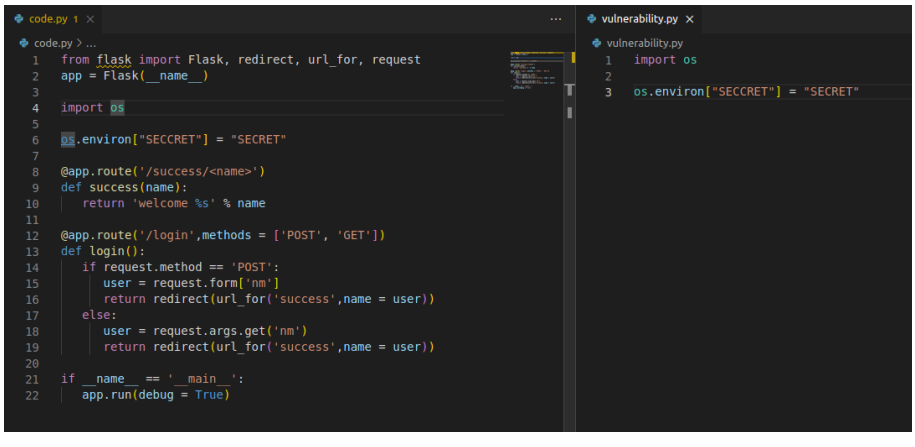

Fig. 8. Vulnerable code

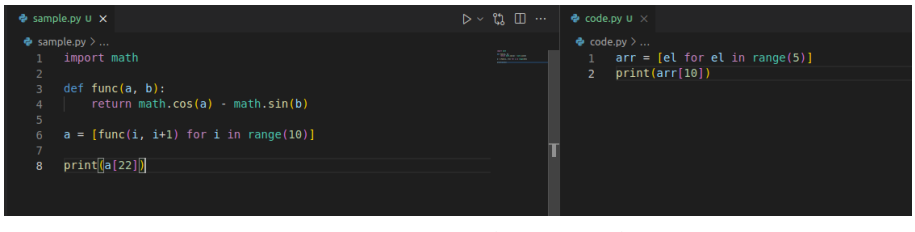

Fig. 9. Vulnerable code (buffer overflow)

Comparing codes with two types of vulnerabilities, it is possible to see that the model does not identify them as clones (Fig. 10, 11).

code.py is not clone of vulnerability.py

Fig. 10. The result of the program

sample.py is not clone of code.py

Fig. 11. The result of the program

This is because the model determined that it is more effective to rely on volume characteristics to search for duplicate codes, as suggested by the authors of the paper [16]. It is implied that the code, which is very different in structure and volume, has more «weight» for the model than the coincidence of a certain number of tokens.

7. SWOT analysis of research results

Strengths. The product presented in the paper is easy to use, shows the high accuracy of clone detection, takes up little memory, and can be used on any operating system because it is written in the Python language, which is cross-platform.

Weaknesses. This model is poorly suited for detecting vulnerabilities in the code.

Opportunities. In further research, it is possible to try to increase the emphasis of the model on comparing individual parts of the code and finding metrics that will be more based on individual types of clones. It is also possible to extend the language capabilities of the model (recognition of clones not only in Python) and train the model to recognize clones of code rewritten from one language to another if desired to improve the user interface.

Threats. Since this software product is not planned to be commercially implemented, there are no threats on the market.

8. Conclusions

1. The existing methods and algorithms of machine learning, which are used in the task of finding duplicates in the program code, are considered:
– decision tree;
– Support Vector Machine (SVM);
– TECCD.

A comparative analysis of the studied algorithms was carried out, according to the results of which the method using the decision tree machine learning model was the most optimal for implementation.

In order to improve the results of the already existing method, a proprietary software product was developed.

2. The decision tree classifier model was developed using the Python module «scikit-learn» [17].

6 metrics were selected for model training, including:
1) ratio of code volumes;
2) the ratio of the volumes of the maximum code blocks;
3) the ratio of volumes of minimal blocks of codes;
4) the ratio of the number of common tokens to the number of smaller code tokens;
5) ratio of the number of rows;
6) relation of intersections of operators.

A total of 600 code instances were randomly assigned to the training set (480 instances; 80 %) and the test set (120 instances; 20 %).

The performance of the model in the clone detection task was checked using the discrepancy matrix, from the indicators of which it can be seen that only 1 out of 120 examples was classified incorrectly under the condition of balanced classes (60 by 60). That is, the accuracy of forecasting on such a dataset is more than 0.99.

3. An experiment was conducted with real works of students, the result of which showed a high level of compliance with academic integrity by students, because at least 17 out of 20 works are unique.

From the result obtained in the experiments with the search for vulnerabilities, it can be concluded that this model is poorly suited for detecting vulnerabilities in the code since, usually, they make up a small part of the entire

code. This prevents them from being classified as a clone of other code with the same vulnerability. This problem can be considered a foundation for further research.

Conflict of interest

The authors declare that they have no conflict of interest about this research, including financial, personal, authorship, or any other nature that could affect the research and its results presented in this article.

References

1. Roy, C. K., Cordy, J. R., Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming, 74 (7)*, 470–495. doi: http://doi.org/10.1016/j.scico.2009.02.007
2. *Code Duplicate*. Available at: https://t2informatik.de/en/smartpedia/code-duplicate/
3. Roy, C. K., Cordy, J. R. (2007). A Survey on Software Clone Detection Research. *Computer and Information Science, 115 (541)*, 115.
4. Arammongkolvichai, V., Koschke, R., Ragkhitwetsagul, C., Choetkiertikul, M., Sunetnanta, T. (2019). Improving Clone Detection Precision Using Machine Learning Techniques. *2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP), 31–36*. doi: http://doi.org/10.1109/iwesep49350.2019.00014
5. Jadon, S. (2016). Code Clones Detection Using Machine Learning Technique: Support Vector Machine. *International Conference on Computing, Communication and Automation (ICCCA2016)*, 299–303. doi: http://doi.org/10.1109/ccaa.2016.7813733
6. Gao, Y., Wang, Z., Liu, S., Yang, L., Sang, W., Cai, Y. (2019). TECCD: A Tree Embedding Approach for Code Clone Detection. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 145–156. doi: http://doi.org/10.1109/icsme.2019.00025
7. Salzberg, S. (1994). C4.5: Programs for Machine Learning by J. Ross Quinlan. Morgan Kaufmann Publishers, Inc., 1993. *Machine Learning, 16 (3)*, 235–240. doi: http://doi.org/10.1007/bf00993309
8. Conforti, R., Leoni, M. D., Rosa, M. L., Aalst, W. V. D. (2013). Supporting risk-informed decisions during business process execution. *25th International Conference on Advanced Information Systems Engineering (CAiSE'13)*, 116–132. doi: http://doi.org/10.1007/978-3-642-38709-8_8
9. Kundel, D. (2020). *ASTs – What are they and how to use them*. Available at: https://www.twilio.com/blog/abstract-syntax-trees
10. Agerholm, S., Larsen, P. G. (1999). A Lightweight Approach to Formal Methods. *Lecture Notes in Computer Science*, 168–183. doi: http://doi.org/10.1007/3-540-48257-1_10
11. *BigCloneBench*. Available at: https://github.com/clonebench/BigCloneBench
12. Buckland, M., Gey, F. (1994). The relationship between Recall and Precision. *Journal of the American Society for Information Science, 45 (1)*, 12–19. doi: https://doi.org/10.1002/(SICI)1097-4571(199401)45:1<12::AID-ASI2>3.0.CO;2-L
13. *Decision Tree Classification Algorithm*. Available at: https://www.javatpoint.com/machine-learning-decision-tree-classification-algorithm
14. *Decision Tree Classifier*. Available at: https://www.sciencedirect.com/topics/computer-science/decision-tree-classifier
15. Bondarenko, O. (2021). *Matrytsia nevidpovidnostei*. Available at: https://oleghbond.medium.com/матриця-невідповідностей-329e7e4bf05e
16. Kubiuk, Y., Kyselov, G. (2021). Comparative analysis of approaches to source code vulnerability detection based on deep learning methods. *Technology Audit and Production Reserves, 3 (2 (59))*, 19–23. doi: http://doi.org/10.15587/2706-5448.2021.233534
17. *Scikit-Learn*. Available at: https://scikit-learn.org/stable/

✉ ***Tetiana Kaliuzhna,*** *Department of System Design, National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Kyiv, Ukraine, ORCID: https://orcid.org/0000-0002-0937-8988, e-mail: kalizhna.tania@gmail.com*

Yevhenii Kubiuk, *Department of System Design, National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Kyiv, Ukraine, ORCID: https://orcid.org/0000-0002-7086-0976*

✉ *Corresponding author*