Code Clone Detection Using Boosting Algorithms

M.V. Thanoshan
Department of Computing and Information Systems,
Sabaragamuwa University of Sri Lanka
Belihuloya, Sri Lanka
mvthanoshan@gmail.com

Kuhaneswaran Banujan
Department of Computing and Information Systems,
Sabaragamuwa University of Sri Lanka
Belihuloya, Sri Lanka
bhakuha@appsc.sab.ac.lk

B.T.G.S Kumara
Department of Computing and Information Systems,
Sabaragamuwa University of Sri Lanka
Belihuloya, Sri Lanka
btgsk2000@gmail.com

S. Prasanth
Department of Physical Sciences and Technologies,
Sabaragamuwa University of Sri Lanka
Belihuloya, Sri Lanka
sprasanth@appsc.sab.ac.lk

Zhenni Li
School of Automation,
Guangdong University of Technology
Guangzhou 510006, China
lizhenni2012@gmail.com

Incheon Paik
School of Computer Science and Engineering,
University of Aizu
Aizu-Wakamatsu, Fukushima, Japan
paikic@u-aizu.ac.jp

2023 3rd International Conference on Advanced Research in Computing (ICARC) | 979-8-3503-4737-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICARC57651.2023.10145638

***Abstract*— To increase programming productivity, developers often copy and paste the source code with or without changing it. However, they may also introduce significant downsides in the long run, including complicating the software and raising maintenance costs. The activity of duplicating the code is known as code cloning. They are classified into four types – Type-1, Type-2, Type-3, and Type-4. In this paper, the author presents a machine-learning approach for detecting code clones of all kinds except for Type-2. Abstract Syntax Trees are used to extract features from the methods. A distance combination approach combines two feature vectors of a pair of methods and their class labels. Once the dataset is finalised, a machine-learning approach is utilised to classify the clone type. Moreover, boosting classifiers like XGBoost, CatBoost, LightGBM, Gradient Boosting and AdaBoost are evaluated for the highest classification accuracy. From the results obtained, LightGBM outperformed all the other classifiers with the highest F1 score of 0.81. This study would motivate future researchers to focus on identifying the Type-2 clones and extracting novel features in determining the clone types.**

Keywords*— *Code Cloning, Software Engineering, Machine Learning, Abstract Syntax Trees.

I. INTRODUCTION

The two primary stages of the software development life cycle are initial development and active maintenance and evolution to fulfil constantly changing customer requirements. In the majority of other industries, development expenditures make up the lion's share of a project's overall cost [1]. However, 90% of the expense of software over its lifespan in the software development sector goes toward maintenance [2]. In software development, developers often tend to copy and paste the code with or without modifications [3]. This copied code (duplicated code) is known as a code clone. The activity of duplicating the code is known as code cloning [4]. The idea of code cloning is recognised as one of the bad smells and makes software maintenance more difficult [5]. Therefore, code clones make it more difficult to maintain software [5, 6].

There are four types of code clones: Type-1 (T1), Type-2 (T2), Type-3 (T3), and Type-4 (T4) [7, 8]. T1 clones are syntactically identical (exactly the same) code fragments besides differences in layouts, comments, and whitespaces. T2 clones are syntactically identical code fragments besides differences in literals, identifiers, and types, along with T1 differences. T3 clones are syntactically similar (not identical) code fragments beside differences in statements (statements can be added or changed, or removed) along with T1 and T2 differences. T4 clones are syntactically dissimilar code fragments that implement the same functionality [7, 8]. Meanwhile, considering the adverse effects of code clones in software, code clone detection is an active area of research [4]. Six code clone detection techniques are available based on internal source code representation. They are text-based, token-based, tree-based, Program Dependency Graph (PDG)-based, metrics-based, and hybrid approaches [8].

In the text-based technique, the target source code is considered a sequence of strings. Code clones are detected by comparing sequences of strings of two code fragments. In the token-based technique, the target source code is parsed into a sequence of tokens. To find duplicated subsequences of tokens, the sequence is scanned. Ultimately, actual code fragments representing duplicated subsequences are returned as code clones. The tree-based technique parses target source code into Abstract Syntax Trees (ASTs). Using tree-matching techniques, similar subtrees are searched in the parsed tree. Finally, actual code fragments representing identical subtrees are returned as code clones. PDG-based technique parses target source code into PDGs. Then, the isomorphic subgraph matching algorithm is utilised to discover similar subgraphs [8].

At last, code fragments representing similar subgraphs are returned as code clones. In the metrics-based technique, various metrics are extracted from code fragments. To detect code clones, vectors of metrics are compared. Hybrid approaches comprise one among two or both hybrid code representation and hybrid techniques. These approaches can, however, also be grouped under the earlier subcategories [8].

Fig. 1 shows that the original method and its T1 clone method are syntactically identical after removing the whitespaces, layouts, and comments (trimming). The original method and its T2 clone method are syntactically identical after trimming and normalising identifiers, literals, and types. Even though a new statement is inserted and an existing statement is modified, the original method is syntactically similar to its T3 clone method after trimming and normalising. It can be noticed that the T4 clone method is syntactically dissimilar from the original method. However, they perform the exact computation even though they are varied a lot in their shape. For example, when focusing on the original method, it uses for loop to calculate the factorial value of the given value of n, whereas its T4 clone method uses recursion to calculate the factorial value of the given value of n. Therefore, in terms of semantics, computation, and functionality, both are similar.

979-8-3503-4737-1/23/$31.00 ©2023 IEEE

In this paper, the author extracts features using ASTs to classify code clones as they require substantially minor effort to represent code patterns, scalable with a huge codebase and have well-defined syntax [9, 10]. Adaptive Boosting (AdaBoost), Gradient Boosting, CatBoost, and Light Gradient Boosting Machine (LightGBM) are the first to use in this research method to the best of the author's knowledge. Remaining of the paper is prepared as follows. In Section II, the author discusses the related works. Section III introduces the methodology. Section IV contains the results and discussion. In Section V, the paper is concluded with future works.

II. Related Works

Sheneamer and Kalita [10] developed machine-learning models to classify T3 and T4 clones. Both AST and PDG were used to extract features. The pair of code fragments are represented as a vector, and pairings are used to train supervised learning classifiers to recognise different sorts of clones. They made use of the IJaDataset 2.0 dataset [11]. They evaluated fifteen machine learning algorithms, including Random Forest, Rotation Forest, and Extreme Gradient Boosting (XGBoost).

Saini, et al. [12] proposed an approach that not only identifies T1 to T3 but also code clones in the Twilight Zone - clones between T3 and T4. Machine learning, information retrieval, and software metrics are all combined to establish their code clone approach. For the dataset, they collected 50,000 arbitrary Java projects from GitHub. Their approach introduced the Siamese architecture of Deep Neural Networks and established high performance in the manual evaluation of precision and evaluation of recall using BigCloneBench [13-15].

For software, functional clone detection, Wu, et al. [16] blended token-based and graph-based methods. As an initial step, they collected Control Flow Graph from the source code. Using social-network-centrality analysis, the centrality of each token in a basic block is then assigned, and the centralities of the same token in other basic blocks are added. This converted a graph into specific tokens with graph information. The siamese architecture of the Neural Network model is trained by utilising the above-mentioned tokens. Google Code Jam and BigCloneBench repository are used for evaluations.

Sheneamer, et al. [17] extracted features from pair of method blocks using ASTs and PDGs. These two feature vectors were fused into a feature vector using three distinct ways – linear combination, multiplicative combination, and distance combination. This research used seven datasets, including Eric, sample j2sdk 1.4.0-java-swing, sample eclipse-jdtcore, eclipse-ant, netbean-javadoc, and Suple and IJaDataset 2.0. They evaluated sixteen classification models, including Random Forest, Convolutional Neural Network, and XGBoost.

Jo, et al. [9] created an approach using a two-pass strategy and a Tree-based Convolution Neural Network to identify different code clones. Initially, source code is converted into ASTs then vector representation of AST nodes is collected. In the first pass, clones are detected. The detected clones are then passed into the second pass, where the clone types are classified. BigCloneBench, a notable and generally utilised repository of cloned code, was used for evaluations. Their approach detected clones (in the first pass) with an average of 96% recall and precision and classified clones (in the second pass) with an average of 78% recall and precision.

White, et al. [18] proposed a deep learning-based approach for code clone detection. Recurrent and Recursive Neural Networks were employed for deep learning code at the lexical and syntactic levels. Tokenising the source code was done using ANTLR, and training was done using the RNNLM toolkit. Additionally, they created AST using the Eclipse Java programming environment. As a result, they were successful in identifying all four varieties of code clones at the file and method levels.

Several approaches have been used to detect code clones based on the related works listed above. However, none of the researchers utilised boosting algorithms such as AdaBoost, CatBoost, Gradient Boosting, and LightGBM to detect code clones. It's reasonable to extract features using ASTs as they efficiently represent the exact syntactic structure of source code and are scalable with massive code. Therefore, it can be a strong foundation to detect code clones using prevailing boosting algorithms to impact the software development industry and the research community positively.

III. Methodology

Definition 1 (Method). A method M refers to a Java method. Within a pair of curly brackets, an ordered sequence of statements, $S_i, i = 1, \dots, N$ that represents how the method should behave, for example, declarations, assignments, method calls, loops, and branching.

$$M = < S_1, \dots S_N >$$

Definition 2 (Code Clones). Two methods M_i, and M_j are considered as code clone pairs if they are similar based on extracted metrics.

$$clone(M_i, M_j) = \begin{cases} 1, if\ sim(M_i, M_j) > \theta \\ 0, otherwise. \end{cases}$$

The entire methodological framework and all the steps in this study are depicted in Fig. 2 below, and each stage has been thoroughly explained.

A. BigCloneBench Java Repository

This study uses BigCloneBench [13-15], a big data inter-project Java repository comprises of known true and false positive clones. BigCloneBench is a prominent repository in code clone detection studies [9, 16]. It's hard to separate T3 and T4 clone pairs with the same functionality since there is no general agreement on the T3 clone's minimum syntactical similarity. Therefore, researchers, based on syntactical similarity, separated them into four categories. They are Very-Strongly Type-3 (VST3), Strongly Type-3 (ST3), Moderately Type-3 (MT3), and Weakly Type-3/Type-4 (WT3/4). VST3 ranges from 90% (inclusive) to 100% (exclusive), ST3 ranges from 70-90%, MT3 ranges from 50-70%, and WT3/4 ranges from 0-50% [13, 14]. BigCloneBench repository offers false clone pairs as well. The meaning of false clone pair is that they are syntactically dissimilar code fragments that don't implement the same functionality (functionally dissimilar code fragments). The syntactical similarity between false clone pairs is purely coincidental [13].

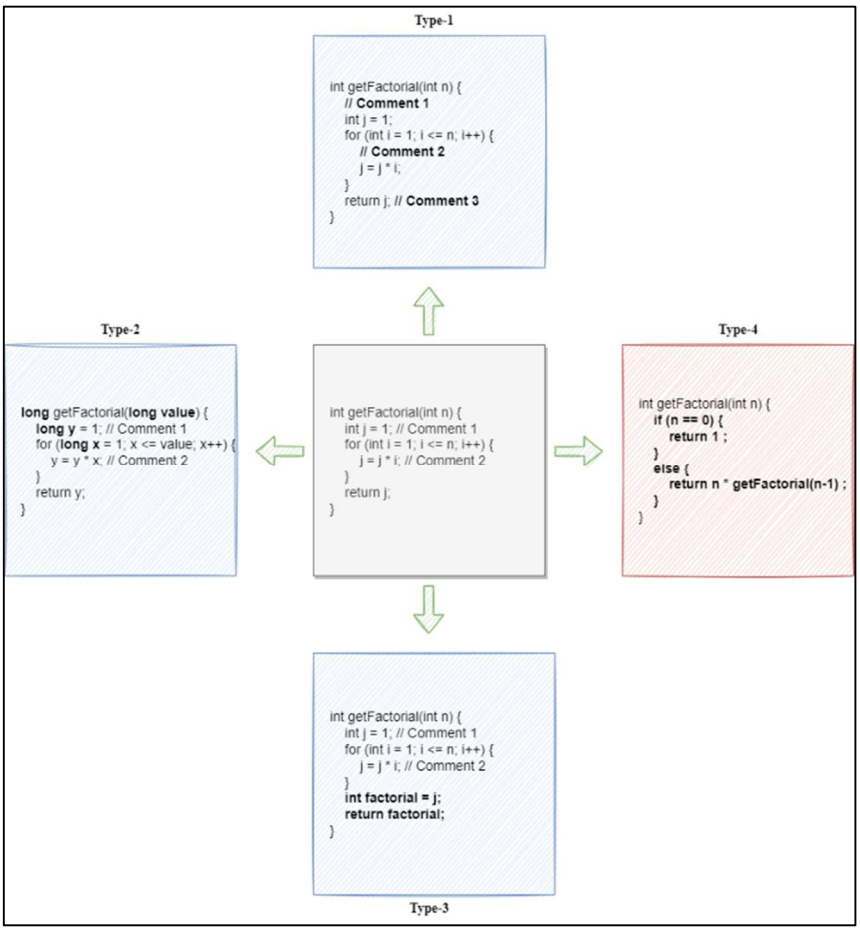

Fig. 1. Examples of different types of code clones

Features of this study are the frequency of programming constructs, and there's no difference between T1 and T2 clones apart from literal values, identifier names, and types. Programming constructs specified in [17] supplementary material cannot focus on differentiating literals, identifiers, and types. Therefore, it cannot be distinguished when fusing two feature vectors of a pair of T1 or T2 methods using the distance combination strategy (as used in this study). Therefore, the author has considered only T1 clone pairs. However, this study can detect the rest of the clone types. Not detecting T2 clones along with the rest is a limitation of this study.

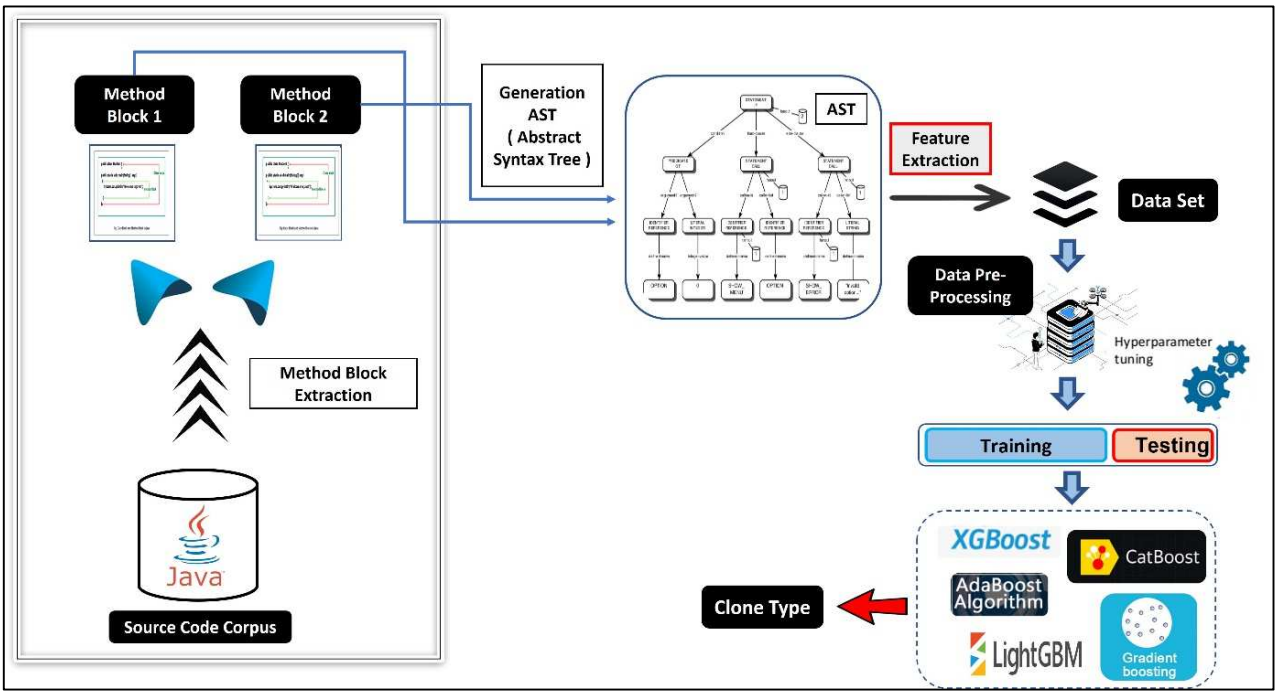

Fig. 2. Research Method

Hence, for classifying code clones properly, this study focuses on six target classes – T1, VST3, ST3, MT3, WT3/4, and False. The author retrieved 4190 pairs for every kind from the BigCloneBench repository. In total, the author retrieved 25140 pair instances representing six target classes.

B. Feature Extraction

The author extracted all, a total of twenty-eight features listed under traditional (fourteen features) and AST (fourteen features) categories specified in the supplementary material of [17], including lines count, assignments count, selection statements count, iteration statements count, synchronised statements count, and return statements count. The author used Eclipse Java Development Tools (JDT) to generate ASTs and extract features. Fig 3 shows the simplified view of AST.

Extracted features of paired methods M_i and M_j can be represented as feature vectors: $M_i = < f_{i1}, f_{i2}, \dots, f_{ik} >$, and $M_j = < f_{j1}, f_{j2}, \dots, f_{jk} >$.

C. Fusion of Method Features

A pair of feature vectors are combined into a single vector followed with corresponding class label. Here, class label refers to the clone class type. Given a pair of methods M_i and M_j , and their class label $C1$, the fused feature vector can be represented as $features(< M_i, M_j >)$. The author fuses two vectors using the distance combination strategy [17]. Distance combination strategy is done by calculating the absolute difference between the two associated values of a feature. In the end, a fused feature vector for a pair of a method with its class label can be represented as $features(< M_i, M_j >) = < |f_{i1} - f_{j1}|, \dots, |f_{ik} - f_{jk}|, C1 >$, where C1 represents the class label.

Summary of Steps Used to Finalise the Dataset

Step 1. Retrieve paired methods. This step retrieves paired methods from the BigCloneBench repository.

Step 2. Extract features from paired methods. This step extracts features from methods using Eclipse JDT.

Step 3. Fuse a pair of feature vectors using a distance combination strategy with their class labels.

Step 4. Feed the data (25140 fused vectors) into CSV.

D. Data Pre-processing

This is one of the prior tasks with the dataset to get prominent results. This process was carried out to remove noisy, duplicate, and unreliable data. It's a time-consuming and tedious task to be performed manually. This process was carried out within a limited time by accommodating a third-party library called "Pandas" and the different pre-processing functions available with mentioned library. Python was a core programming language to do almost all the tasks from pre-processing to model development.

The normalisation was performed with the aforementioned dataset under pre-processing to refine the data further. To do so, Min-Max Scaling has accommodated to re-scale the features in the range of [0,1] for the specified data.

E. Identifying the Optimum Values for the Hyper-parameters

A machine model can behave differently from different datasets. So, it's one of the crucial parts of controlling the model's behaviour or identifying the optimum values for its respective hyper-parameters for a specific dataset. This task can be frequently accomplished through a search algorithm. During this study, the GridSearchCV method offered by Scikit-learn was used. The entire dataset was split into two partitions, namely training and testing, with a percentage of 70 and 30, respectively. Then 'train_test_split' method facilitated by the Sklearn was configured initially to perform the GridSearchCV.

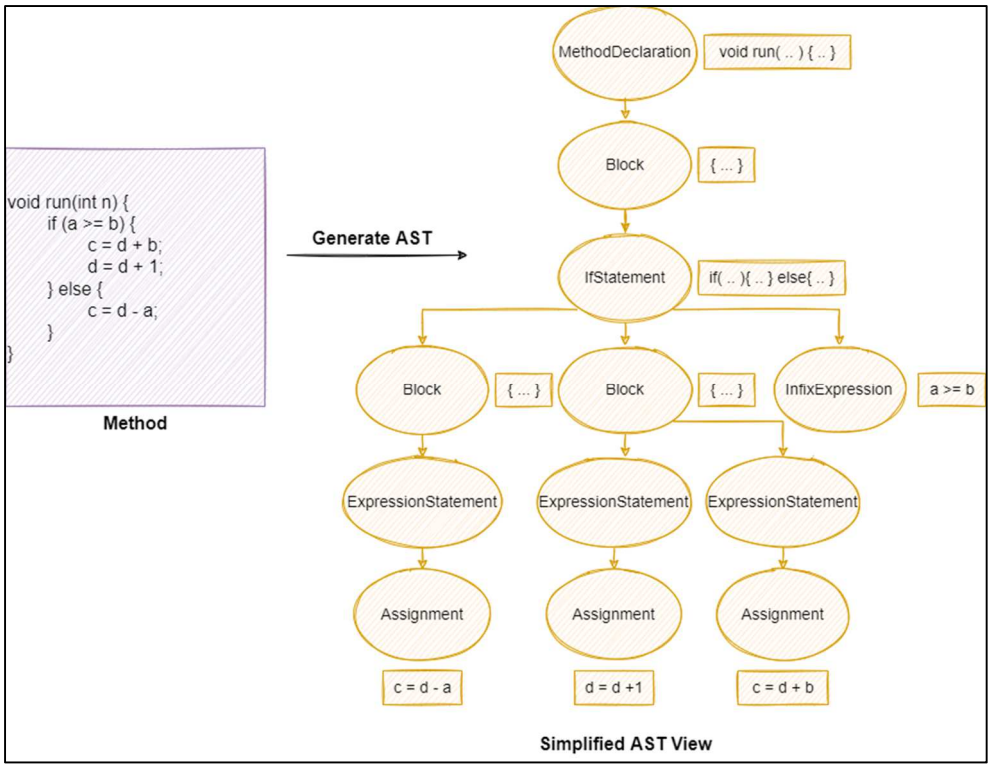

Fig. 3. Simplified AST View

F. Implementing the Model

Each algorithm/model needs to be trained along with the values for its respective hyper-parameters. So, once the individual hyper-parameter values have been set, the data can be passed into the aforementioned boosting algorithms for the training, and each boosting algorithm will be tested for the highest classification accuracy. Moreover, the evaluation matrices like Precision, Recall, F1-Score, and Support values also get evaluated to check the performance of the classifiers considered.

IV. Results and Discussion

This section explored the results obtained from the significant processes mentioned above. The optimum values discovered from GridSearchCV for the hyper-parameters of the selected boosting algorithms are mentioned in Table I below.

TABLE I. Hyper-parameters of boosting algorithms

Boosting Technique	Hyper-parameters and Their Optimum Values
CatBoost	depth=6, iterations=90, learning_rate=0.02
LightGBM	n_estimators = 460, colsample_bytree = 0.8, max_depth= 8, num_leaves=10, reg_alpha=1.2, learning_rate=0.12, reg_lambda=1.2, subsample=0.8, subsample_freq=10
XGBoost	n_estimators=400, gamma=1, colsample_bytree=0.7, max_depth=10, reg_alpha=1.2, reg_lambda=1.2, subsample=0.8
AdaBoost	learning_rate = 0.01, n_estimators = 600
Gradient Boosting	n_estimators=60, learning_rate= 0.1, max_features='sqrt', max_depth=5, random_state=10

Once the appropriate hyper-parameter values were set with respective algorithms, training and evaluating the algorithms were carried out. Table II below specifies the accuracies and error rate obtained for each boosting algorithm in classifying the clone types.

TABLE II. Evaluation results obtained for boosting algorithms

Technique	Recall	Precision	F1 Score
CatBoost	0.73	0.74	0.71
LightGBM	**0.82**	**0.83**	**0.81**
XGBoost	0.81	0.82	0.80
AdaBoost	0.39	0.27	0.28
Gradient Boosting	0.73	0.74	0.72

As mentioned in Table II, the LightGBM model performs well with a Recall of 0.82, Precision of 0.83, and F1 Score of 0.81. Moreover, evaluation metrics are also grabbed and represented in Table III to ensure the performance of boosting algorithms in classifying the clone types.

The highest values across different algorithms and clones for each precision, recall, and F1 score are highlighted. Interestingly, a 1.00 recall score is obtained by all models for T1 clones. Although all the models exhibit good performances overall, the AdaBoost model fails to predict WT3/4 and false clone pairs. According to Table III, it's concluded LightGBM model has produced more reliable and precise results.

Results on BigCloneBench are reported in Table IV. Results of the prevailing techniques are obtained from [19]. It can be seen that the LightGBM model outperforms SourcererCC, RtvNN, and Deckard in Recall. However, it shows a lack of performance in precision. It surpasses SourcererCC, RtvNN, and Deckard in F1 Score. Overall, TreeCen performs well regarding Recall, Precision, and F1 Score. It indicates that more novel and valuable features must be extracted from the source code. Further, for effective code clone detection, Deep Learning approaches can be utilised.

TABLE III. Performance of boosting algorithms

Algorithms	Type of Clone	Precision	Recall	F1 Score
CatBoost	T1	0.70	**1.00**	0.82
	VST3	0.81	0.40	0.54
	ST3	0.68	0.68	0.68
	MT3	0.70	0.74	0.72
	WT3/4	0.72	0.86	0.79
	False	0.82	0.66	0.73
LightGBM	T1	**0.71**	**1.00**	**0.83**
	VST3	**0.86**	**0.52**	**0.65**
	ST3	**0.81**	0.80	**0.80**
	MT3	**0.83**	**0.84**	**0.83**
	WT3/4	**0.88**	**0.90**	**0.89**
	False	**0.89**	**0.86**	**0.87**
XGBoost	T1	**0.71**	**1.00**	**0.83**
	VST3	0.84	**0.52**	0.64
	ST3	**0.81**	0.77	0.79
	MT3	0.80	0.83	0.82
	WT3/4	0.86	**0.90**	0.88
	False	**0.89**	0.84	**0.87**
AdaBoost	T1	0.69	**1.00**	0.82
	VST3	0.35	0.03	0.06
	ST3	0.41	**0.83**	0.55
	MT3	0.20	0.48	0.28
	WT3/4	0.00	0.00	0.00
	False	0.00	0.00	0.00
Gradient Boosting	T1	0.70	**1.00**	0.82
	VST3	0.80	0.44	0.57
	ST3	0.68	0.68	0.68
	MT3	0.69	0.72	0.70
	WT3/4	0.79	0.81	0.80
	False	0.78	0.75	0.76

TABLE IV. Results on BigCloneBench

Technique	Recall	Precision	F1 Score
LightGBM	0.82	0.83	0.81
Sheneamer, et al. [17]	0.95	0.95	0.95
SourcererCC	0.07	0.98	0.14
RtvNN	0.01	0.95	0.01
DeepSim	0.98	0.97	0.98
SCDetector	0.92	0.97	0.95
Deckard	0.06	0.93	0.12
ASTNN	0.94	0.92	0.93
TreeCen	**0.99**	**0.99**	**0.99**

V. Conclusion and Future Works

Detection of code clones is necessary for active software maintenance, minimising software maintenance cost and reducing bad smells in the code. Twenty-eight features are extracted from source code using ASTs because of their high scalability, large codebase, well-defined syntax, and minimal effort required to generate them. The author presents a machine-learning approach for detecting code clones. For the greatest classification accuracy, boosting classifiers were also considered, including XGBoost, CatBoost, LightGBM, Gradient Boosting, and AdaBoost. According to the results, LightGBM performed better than all the other classifiers, with a maximum F1 score of 0.81. This work will encourage subsequent researchers to concentrate more on locating T2 clones and obtaining fresh features. As a future work, the

author plans to extract novel features from the source code to detect T2 clones with the rest of the clones and to improve the code clone detection performance.

ACKNOWLEDGEMENT

This research is partially funded by Foreign Expert Project of Ministry of Science and Technology of China under Grant DL2022030011L.

REFERENCES

[1] F. Rahman, C. Bird, and P. Devanbu, "Clones: What Is That Smell?," *Empirical Software Engineering,* vol. 17, pp. 503-530, 2012.

[2] S. M. H. Dehaghani and N. Hajrahimi, "Which Factors Affect Software Projects Maintenance Cost More?," *Acta Informatica Medica,* vol. 21, p. 63, 2013.

[3] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An Ethnographic Study of Copy and Paste Programming Practices in Oopl," in *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE'04.*, 2004, pp. 83-92.

[4] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, "A Systematic Review on Code Clone Detection," *IEEE access,* vol. 7, pp. 86121-86144, 2019.

[5] A. Monden, D. Nakae, T. Kamiya, S.-i. Sato, and K.-i. Matsumoto, "Software Quality Analysis by Code Clones in Industrial Legacy Software," in *Proceedings Eighth IEEE Symposium on Software Metrics*, 2002, pp. 87-94.

[6] A. Yamashita and S. Counsell, "Code Smells as System-level Indicators of Maintainability: an Empirical Study," *Journal of Systems and Software,* vol. 86, pp. 2639-2653, 2013.

[7] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Transactions on software engineering,* vol. 33, pp. 577-591, 2007.

[8] C. K. Roy and J. R. Cordy, "A Survey on Software Clone Detection Research," *Queen's School of Computing TR,* vol. 541, pp. 64-68, 2007.

[9] Y.-B. Jo, J. Lee, and C.-J. Yoo, "Two-pass Technique for Clone Detection and Type Classification Using Tree-based Convolution Neural Network," *Applied Sciences,* vol. 11, p. 6613, 2021.

[10] A. Sheneamer and J. Kalita, "Semantic Clone Detection Using Machine Learning," in *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 2016, pp. 1024-1028.

[11] (2013). *SeClone - secold*. Available: https://sites.google.com/site/asegsecold/projects/seclone

[12] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: Detection of Clones in the Twilight Zone," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 354-365.

[13] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a Big Data Curated Benchmark of Inter-project Code Clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 476-480.

[14] J. Svajlenko and C. K. Roy, "Evaluating Clone Detection Tools With Bigclonebench," in *2015 IEEE international conference on software maintenance and evolution (ICSME)*, 2015, pp. 131-140.

[15] J. Svajlenko and C. K. Roy, "Bigcloneeval: a Clone Detection Tool Evaluation Framework With Bigclonebench," in *2016 IEEE international conference on software maintenance and evolution (ICSME)*, 2016, pp. 596-600.

[16] Y. Wu, D. Zou, S. Dou, S. Yang, W. Yang, F. Cheng, *et al.*, "Scdetector: Software Functional Clone Detection Based on Semantic Tokens Analysis," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 821-833.

[17] A. Sheneamer, S. Roy, and J. Kalita, "An Effective Semantic Code Clone Detection Framework Using Pairwise Feature Fusion," *IEEE Access,* 2021.

[18] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep Learning Code Fragments for Code Clone Detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 87-98.

[19] Y. Hu, D. Zou, J. Peng, Y. Wu, J. Shan, and H. Jin, "Treecen: Building Tree Graph for Scalable Semantic Code Clone Detection," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1-12.