Improving Clone Detection Precision using Machine Learning Techniques

Vara Arammongkolvichai*, Rainer Koschke†, Chaiyong Ragkhitwetsagul*,
Morakot Choetkiertikul*, Thanwadee Sunetnanta*
*Faculty of Information and Communication Technology (ICT), Mahidol University, Thailand
†University of Bremen, Germany
Email: vara.ara@student.mahidol.ac.th, koschke@uni-bremen.de,
{chaiyong.rag, morakot.cho, thanwadee.sun}@mahidol.ac.th

***Abstract*—Code clones or similar segments of code in a software project can be detected by using a clone detection tool. Due to modifications applied after copying and pasting of the cloned code, the current code clone detection tools face challenges to accurately detect clones with heavy modifications (i.e., Type-3 clones or clones with added/deleted/modified statements). One challenge is because the clone results contain several false positives.**

In this paper, we propose an approach for increasing the precision of code clone detection using machine learning techniques. By training a decision tree on 19 clone class metrics, we use the trained decision tree as a clone filter by placing it in the last step in the clone detection pipeline. This aims to remove false positive clone classes reported by a clone detection tool.

We found that the decision tree clone filter is helpful for decreasing the number of false positive clone classes in iClones, a well-known code clone detector. After training the decision tree on 537 clone classes in JFreeChart and evaluating it on the test data set, it could improve iClone's precision from 0.94 to 0.98. The findings show that decision tree can be used effectively for filtering false positive clones. Nonetheless, we found that the filter is only effective for Java and does not offer satisfying performance when running on a Django Python project.

***Index Terms*—code clones, machine learning, decision tree**

I. Introduction

Copying and pasting pieces of source code is a common activity in software development. The result of this activity are duplicated pieces of code, i.e., code clones. The clones can be exactly identical, slightly different, or largely different depending on the amount of modifications made to them after copying [1]. Code clone detection is an established research area in software engineering aiming to locate such duplicated pieces of code in software. There are several techniques introduced in the literature to detect clones such as text-based, token-based, tree-based, and graph-based techniques [1]–[4].

Although exact-match clones (i.e., Type-1 clones) or clones with renaming (i.e., Type-2 clones) can be effectively detected by the current state-of-the-art clone detectors, the clones that contain several modifications such as added/deleted/modified statements (i.e., Type-3 clones) still remain a challenge [5]. Moreover, they are the largest number of clones found in software projects [6]. Lately, machine learning techniques are also used for code clone detection, both for the detection technique itself [7], [8] and for enhancing the clone detection performance [5], [9]. All the mentioned techniques are performed on the clone pair level and only work with clone detectors that report clone pairs.

This paper presents an alternative approach based on using a machine learning model to enhance the precision of code clone detection in filtering spurious clones at *the clone class level*. We selected 19 clone class metrics that capture different characteristics of cloned and non-cloned classes and trained a decision tree binary classifier to help to filter out spurious clone classes from the original clone result reported by a clone detection tool.

The paper makes the following contributions.

- A technique of using clone class metrics and a machine learning model to create a clone filter and its evaluation.
- A case study of applying a clone filter trained from one language to clones in another language.

The introduced technique is useful for code clone detectors that report the clones based on clone classes, not clone pairs. The examples of such clone detectors are iClones [10], NiCad [11], [12], and Simian [13]. Moreover, by including the machine learning technique as a clone class filter, the approach can be applied to several existing clone detectors without the need to alter their detection methods.

II. Background

A. Code Clones

Code clones are similar code fragments (i.e., segments of code). They are created by copying and pasting with or without minor adaptation, which are common activities in software development. Code clones need to be located when developers want to enhance or update an existing clone fragment. The benefits and drawbacks of clone clones are still controversial [14]–[16]. While the clones may create maintenance issues in general, they are also beneficial in some specific cases [16]. The literature in code clone research has separated the clones into four types [1]

- **Type-1** clones are exact copies of code without any modification except formatting, such as white space and comments.
- **Type-2** clones are copies of code with only variations in formatting and parameters (i.e., variables, type, function identifiers, or literals).

- **Type-3** clones are copies of code with modifications including added, deleted, or modified statements.
- **Type-4** clones are code fragments that perform a similar task (similar semantic) but with different implementations.

We mainly focus on code clones of Type-1 to Type-3 in this study since we are interested in code clones that contain some level of syntactic similarity.

B. Machine Learning (ML) Model: Decision Tree

Decision Tree is a supervised learning algorithm that generates decision nodes using the information gain obtained from the value of each feature [17]. It can be represented as a tree graph model that consists of multiple levels of nodes representing a decision rule. The classification is made by passing the data through the tree from the top to a leaf node. At each decision node, the branch is selected based on the value of the corresponding feature. A significant benefit of the decision tree model over other models is that it provides an interpretable result [18].

C. Code Metrics for Clone Filtering

Higo et al. [19] propose a metric called RNR as a clone filter for CCFinder clone detector [20]. RNR measures the ratio of non-repeated code sequence over the whole code sequence in a clone set. Saini et al. [5] employed 24 Java method-level code metrics to create a filter for challenging clones. The authors used a deep neural network trained on those metrics to classify cloned/non-cloned pairs. The technique combined with other two filters has shown to give high precision and recall. Moreover, the study by Koschke and Bazrafshan [21] uses nine code metrics extracted from the clones including number of tokens, number of parameters, clone type, the number of distinct token types in a sequence, fraction of non-repetitive tokens, parameter overlap, parameter consistency, degree of valid references, and fraction of repeated parameters. Then, a filter is created using a decision tree machine learning model to filter spurious clone candidates. The filter is used to complement the authors' token-based code clone detector *cpf* in their study of clone rates in open-source programs written in C or C++.

III. Research Questions

In this paper, we ask the following research questions:

- **RQ1: Effectiveness of ML clone filter to improve clone detection precision** *How well does the trained ML model based on clone class features help to improve the precision of code clone detection?* We investigate whether the features that are extracted from clone classes can be used as guidelines for filtering clone classes.
- **RQ2: Effectiveness of applying an ML clone filter trained from one language to another language** *How well can an ML model based on clone class features be applied on another different language?* We also investigate whether the clone filter using a machine learning model that is trained on clones from one programming language can be effectively applied to clones from another programming language. This would help us realize how possible it is to create a universal (cross-language) model for code clone filter.

IV. Methodology

To answer the research questions, we performed an experiment of training a clone filter using a decision tree model and evaluating its effectiveness in removing false clone classes from clone detection results.

A. Experimental framework

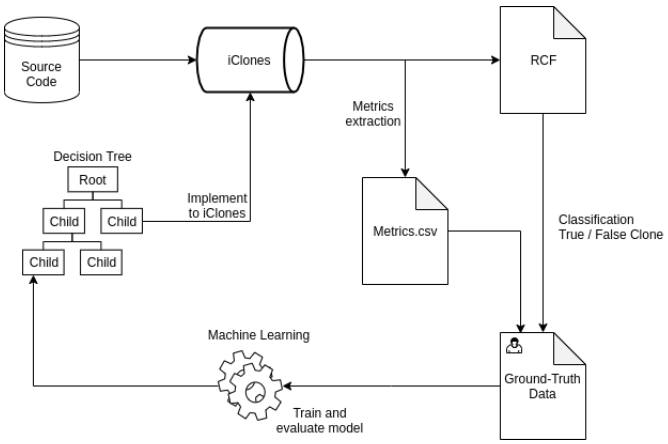

Fig. 1. Experimental frame-work diagram

We followed the experimental framework shown in fig. 1 in this study. First, the source code data of a project under analysis is fed into iClones [10], a code clone detector that is used in several clone studies [2], [3], [22]. The tool detects clones of Type-1 to Type-3 in a given project and reports clone classes in the RCF format [10]. At the same time, the metric extraction is performed on the reported clone classes. The clone validation is then performed by a human on the reported clone classes to classify them into true or false clone classes. These validated clone classes are used as ground truth data in the training and the evaluation of the model. The training and evaluation of the decision tree model is then performed based on the 19 metrics and the manually validated ground truth data. Finally, the derived decision tree is implemented as a module in iClones.

Next, we explain the three key steps in the experimental framework (clone class metric extraction, clone validation, and model training and evaluation) in detail below.

B. Clone Class Metric Extraction

We invented 19 clone class metrics for this study as shown in table I. Some of the metrics, such as clone type (cloneType) or number of distinct token types (uniqueTokenNo), are adopted and modified from the previous study by Koschke and Bazrafshan [21]. The clone class metrics aim to capture different properties of the clones. For example, the file name similarity (similarFileNameRate) metric shows whether the

clone classes occur in the same file, in different files but with similar file names, or in files with totally different file names. The ratio of clone gap tokens over all the tokens (gapRate) metric indicates how much (percentage) of the source code in the same clone class are not exactly the same.

C. Data Set and Clone Validation

To answer RQ1, we selected JFreeChart, an open-source Java project that generates graphical charts. The project, containing 987 Java files, is used to create ground-truth data. The statistics of JFreeChart is presented in table II. We extracted the clone classes from this project using iClones. The result contained 671 clone classes. The first author, who is a computer science student and is familiar with code clones, took the role of an investigator and performed a clone validation by manually checking all the 671 clone classes and decided whether they were true or false clone classes. After the clone validation, we found 46 false clone classes and 625 true clone classes.

D. Error Measures

We trained and evaluated the ML filter towards precision. Its definitions are as follows:

$$\text{precision} = \frac{TP}{TP + FP}$$

where *TP* represents the actual true clone classes that are classified as true clone classes by the decision tree, and *FP* represents the actual spurious clone classes that are classified as true clone classes.

E. Model Training, Tuning, and Evaluation

We divided the validated clone data into three sets: training, validation, and test. Seventy percent of the data (470 clone classes) is selected to train the model, ten percent (67 clone classes) is used to validate the model and tune the parameters, and twenty percent (134 clone classes) is for testing the model.

After training the decision tree using the training set, the tuning of the decision tree's parameters using the validation set is performed. In general, a machine learning algorithm may offer different performance if tuned differently. Thus, it is important to derive optimal (or close to optimal) parameter values for each machine learning technique used. The decision tree has multiple parameters to configure (e.g., `max_depth`, `min_samples_split`, `min_samples_leaf`, `min_weight_fraction_leaf`, and `max_features`). Nonetheless, in this study, we focus only on tuning the `max_depth` parameter (i.e., how many level the decision tree should have). The models were trained on the training set with different `max_depth` values and were executed to classify the data in the validation set. We varied the value of `max_depth` from 3, 4, 5, 6, 7, and 8 and the precision score associated with each `max_depth` value was recorded. At the end, we selected the `max_depth` value that offered the highest precision score.

After training and tuning the decision tree model, we evaluated the trained decision tree on the test set of 134 clone classes, which consist of 126 true clone classes and eight false clone classes. Since the data set is created from the result of iClones, we could only measure the model precision. We could not compute recall because we did not have a complete ground truth of all the clones in the JFreeChart project.

F. Evaluation of the Clone Filter on Clones in Another Language

To answer RQ2, we integrated the decision tree clone filter into iClones by translating the structure of the decision tree into sequences of `if-else` statements. The filter is put at the last step in the clone detection pipeline to analyze the clone class candidates and remove spurious clones before the tool generates the RCF clone report.

Then, we ran iClones with the trained decision tree built in as a clone class filter on a Python project and evaluated its effectiveness. We picked the Django project as our subject of study due to its popularity and its relatively similar size to the JFreeChart project. To test the effectiveness of the filter, we ran iClones twice: with and without the filter. We then compared the results to find the differences, i.e., the clone classes that were removed by the filter. Then, the first author manually validated the removed clone classes.

V. Results and Discussion

In this section, we discuss the results from the experiment including the correlations between the metrics, the trained decision tree, and the answers to RQ1 (the effectiveness of using a decision tree to improve clone detection precision) and RQ2 (the effectiveness of using a decision tree trained from one language for another language).

A. Correlations Between Clone Class Metrics

From fig. 2, the heatmap shows the pair-wise mutual Pearson correlations between the 19 clone class metrics. The value of the correlation is between -1 and +1, where +1 means the two metrics perfectly correlate positively and -1 perfectly correlate negatively. The light color (yellow) indicates positive correlations, while the dark color (blue) indicates negative correlations among the metrics. We observe that there are some metrics that are highly correlated. For example, the set of metrics on the top left of the chart, which includes {`minFragmentLength`, `maxFragmentLength`, `cloneLength`, `uniqueTokenNo`, `identifierNo`, `uniqueIdentifierNo`, `operatorNo`}, shows medium to high correlation among themselves (0.32–0.99). Another correlated metric pair is {`similarFileNameRate`, `similarFilePathRate`} (0.92) which is quite expected due to the semantic similarity of the two metrics. Lastly, we found that the metrics about clone gaps are also highly correlated, i.e., {`totalGap`, `maxGap`, `longestGap`, `gapRate`, `and gapCount`} (0.41–0.96).

B. Trained Decision Tree

The decision tree from the training on 625 clone classes in the training set and tuning of the `max_depth` parameter

TABLE I
CLONE CLASS METRICS

No.	Metric	Description
1	cloneType	The type of clone class: Type-1, Type-2, or Type-3
2	volume	Total number of tokens in the entire clone class
3	maxFragmentLength	Longest fragment length
4	minFragmentLength	Shortest fragment length
5	cloneLength	Number of tokens that are clone
6	uniqueTokenNo	Number of unique tokens
7	identifierNo	Number of identifiers
8	uniqueIdentifierNo	Number of unique identifiers
9	operatorNo	Number of operators
10	overlapIdentifierRate	Identifier overlap ratio
11	fileNO	Number of files
12	tokenTypeDiversity	Number of unique token types
13	similarFileNameRate	File name similarity score
14	similarFilePathRate	File path similarity score
15	totalGap	Number of tokens that are modification section or gap
16	maxGap	Size of the largest clone gap (tokens)
17	longestGap	Size of the longest clone gap (tokens)
18	gapRate	Ratio between number of gap tokens and number of non-gap tokens
19	gapCount	Number of gaps between the cloned segments

TABLE II
STATISTICS OF THE DATA SETS

Project	Files	Clone classes (T/F)	Training – 70% (T/F)	Validation – 10% (T/F)	Test – 20% (T/F)
JFreeChart	987	671 (625/46)	469 (439/30)	68 (60/8)	134 (126/8)

on 67 clone classes in the validation set is shown in fig. 3. The trained decision tree has the `max_depth` value of 5. We can see that, from the structure of the tree, the root node is the `gapRate` metric with the split of 30 spurious clone classes and 439 true clone classes. This is the best metric according to our data set that represents the first split between the two result categories. A low `gapRate` tends to show that the clone classes are true clones, while a high `gapRate` hints that the clone classes are spurious ones. Then, the clone class metrics that are evaluated next are the `longestGap` and `similarFilePathRate`. We observed that from the total of 19 clone class metrics that we extracted, only 9 metrics, `gapRate`, `longestGap`, `similarFilePathRate`, `totalGap`, `overlapIdentifierRate`, `uniqueIdentifierNo`, `maxFragmentLength`, `volume`, and `gapCount`, contribute to the final decision tree. Thus, they are the most effective metrics for building the clone class filter in this study.

C. RQ1: Effectiveness of ML clone filter to improve clone detection precision

After evaluating the trained decision tree on the test set, we found that the decision tree filter removed 5 out of 8 false clone classes in the total of 134 classes in the test data set. Compared to the original result that is reported by iClones (126 true clone classes and 8 false clone classes), this improves the precision of iClones on the test data set from 0.94 to 0.98[1].

> **Answer to RQ1:** The ML clone filter is effective on improving clone detection precision. The case study of decision tree filter integrated in iClones shows that it can increase iClones's precision from 0.94 to 0.98.

D. RQ2: Effectiveness of applying an ML clone filter trained from one language to another language

After comparing the clone result of iClones between with and without the filter, we found 33 clone classes that were removed after adding the filter to iClones. The manual validation of the removed clone classes shows that eight of them were actually false clones. Nonetheless, the other 25 removed clone classes were true clones. The finding shows that the Java filter is too strict in removing Python clones and removed several true clone classes. There are a few possible explanations for this. First, the data set may be too small, which makes the model overfitting to the Java clone data. A larger training data set is needed in order to create a more generalized clone filter. Second, the invented metrics may not effectively capture the characteristics of clones in different languages. Other clone class metrics better capturing the clones in different languages may be needed.

[1]Original precision = $\frac{126}{(126+8)} = 0.94$, Filter's precision = $\frac{126}{(126+3)} = 0.98$

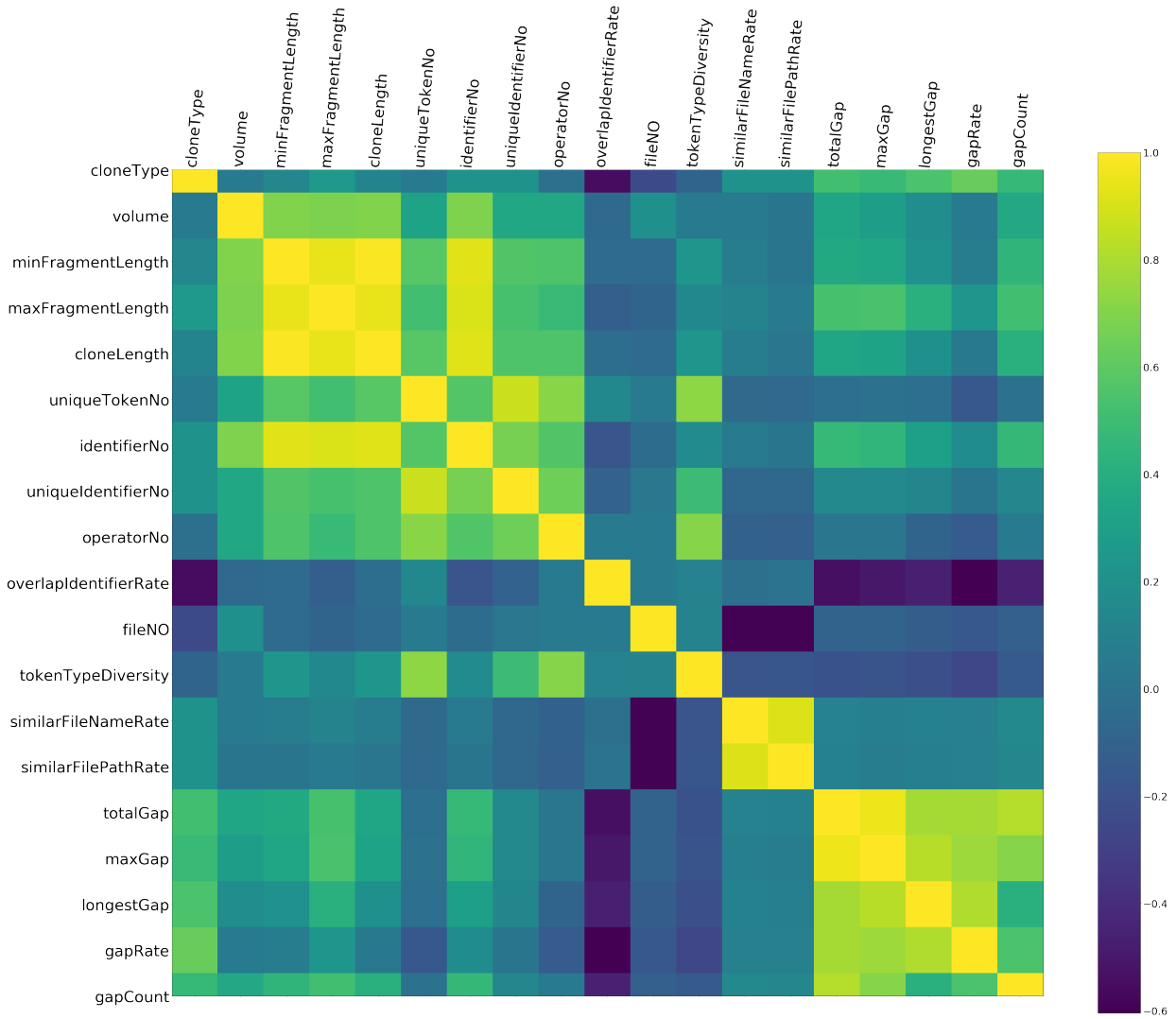

Fig. 2. Correlation between clone class metrics

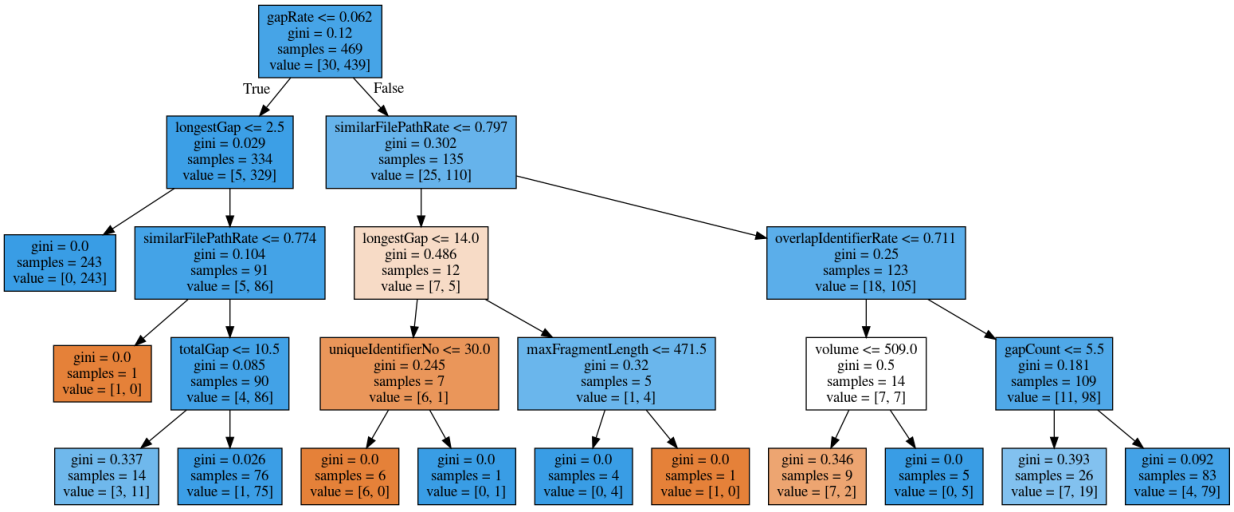

Fig. 3. Trained decision tree

Answer to RQ2: The ML clone filter trained on Java is not effective on filtering clones in Python.

VI. Threats to Validity and Future Work

Internal validity: The training and parameter tuning of the decision tree model is based on precision. With the restriction of the creation of ground truth data, it is the only error measure we could use. If the data allows, training the decision tree model using other error measure (e.g., recall, F1) may produce different result. The decision tree performance may be varied based on the tuning of its parameters. We mitigated this threat by tuning the `max_depth` parameter using the validation set. Third, the data set that we used suffer from imbalance between the true and false clone classes, i.e., the number of true clone classes are much higher than the number of false clone classes. Moreover, the clone validation is performed by only one investigator, which may introduce bias into the result. We plan to mitigate these two threats in the future work.

External validity: The findings in this study is based only on the JFreeChart open source Java project and Django open source Python project. It may not be generalized to software written in other languages or commercial software. Moreover, the effectiveness of the filter is only from a decision tree model. Selecting other machine learning techniques for this task may give different results.

Future work: We plan to improve the work further on the following aspects. First, we plan to increase the number of data points to train and evaluate the model in Java. By using larger and more balance ground-truth data, more accurate models are generally expected. Moreover, this might also mitigate the issue of poor performance of the filter on Python clones. Second, we will try to train another decision tree on the Python clone data and compare it with the decision tree derived from the Java clones to gain more insights into their differences. Third, we will try other more sophisticated machine learning techniques, e.g., random forest. They may offer better performance than the current decision tree model. Last, we will implement a function that allows the iClones users to plugin their own filter without updating the source code.

VII. Acknowledgement

This research project was partially supported by Faculty of Information and Communication Technology, Mahidol University.

VIII. Conclusion

The paper introduces an approach to enhance existing code clone detection tools with a machine learning technique. We invent 19 clone class metrics to capture different 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 spurious clone classes from the clone result.

By training the decision tree on 671 Java clone classes detected by iClones clone detector, the result shows that the decision tree could remove five spurious clone classes that were originally reported by iClones, hence increasing the precision of the detection.

However, the application of the decision tree clone filter trained on Java clones to Python clones reveals that the filter was not effective on the other language and more future work is needed on this issue.

References

[1] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[2] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "Similarity of source code in the presence of pervasive modifications," in *SCAM '16*, 2016, pp. 117–126.

[3] ——, "A comparison of code similarity analysers," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2464–2519, 2018.

[4] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[5] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: detection of clones in the twilight zone," in *ESEC/FSE '18*, 2018, pp. 354–365.

[6] 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 *ICSME '14*, 2014, pp. 476–480.

[7] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *ASE '16*, 2016, pp. 87–98.

[8] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "CCLearner: A deep learning-based clone detection approach," in *ICSME '17*, 2017, pp. 249–260.

[9] J. Yang, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Classification model for code clones based on machine learning," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1095–1125, Aug 2015.

[10] N. Gode and R. Koschke, "Incremental clone detection," in *CSMR '09*. IEEE, 2009, pp. 219–228.

[11] C. K. Roy and J. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," in *ICPC '08*, 2008, pp. 172–181.

[12] J. R. Cordy and C. K. Roy, "The NiCad Clone Detector," in *ICPC '11*, 2011, pp. 219–220.

[13] S. Harris, "Simian – similarity analyser, version 2.4," http://www.harukizaemon.com/simian/, 2015, accessed: 2016-02-14.

[14] J. Krinke, "Is cloned code more stable than non-cloned code?" in *SCAM '08*, 2008, pp. 57–66.

[15] M. Mondal, M. S. Rahman, C. K. Roy, and K. A. Schneider, "Is cloned code really stable?" *Empirical Software Engineering*, vol. 23, no. 2, pp. 693–770, 2018.

[16] C. J. Kapser and M. W. Godfrey, "Cloning considered harmful considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.

[17] S. L. Salzberg, "C4.5: Programs for Machine Learning by J. Ross Quinlan. Morgan Kaufmann Publishers, Inc., 1993," *Machine Learning*, vol. 16, no. 3, pp. 235–240, Sep 1994.

[18] R. Conforti, M. D. Leoni, M. L. Rosa, W. M. P. V. D. Aalst, M. de Leoni, M. La Rosa, and W. M. van der Aalst, "Supporting risk-informed decisions during business process execution," *Advanced Information Systems Engineering - Lecture Notes in Computer Science*, Jan 2013.

[19] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Method and implementation for investigating code clones in a software system," *Information and Software Technology*, vol. 49, no. 9-10, pp. 985–998, Sep 2007.

[20] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions in Software Engineering*, vol. 28, no. 7, pp. 654–670, Jul 2002.

[21] R. Koschke and S. Bazrafshan, "Software-clone rates in open-source programs written in C or C++," in *SANER '16*. IEEE, 2016, pp. 1–7.

[22] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *ICSE '16*, 2016, pp. 1157–1168.