Code Clone Detection Using Syntactic and Semantic Properties

BLG 630E Recommendation Systems in Software Engineering

Halit Uyanık, *504202506, ITU*

***Abstract*—Detecting clones in a software enables developers to reduce the cost of maintenance in the long run. Duplicate codes and functionalities increases the cost of making changes, and can also introduce bugs in multiple places at the same time. In this project, a method clone detection approach from another study is implemented using Java language. A machine learning model is built and then trained with a huge code clone database. Then, a UI is implemented in order to use this trained model to detect clones in any software written in Java. Results show that the detection tool performs well on type1 and type2 clones, whereas it does not perform well in distinguishing type3 and type4 clones when the test set is similar to real world scenario.**

***Index Terms*—Machine Learning, Recommendation Systems, Sofware Quality, Clone Detection**

I. Introduction

CLONE detection in software is one of the problems which have taken the attention of researches for a long time. After taking a short look at the reseach papers, it is easy to see that detecting type1 and type2 clones with 100% accuracy is possible. There are also commercial and non-commercial tools which makes it easy for developers to detect clones in their applications. These tools enable users to also see not only exact clones, but also clones which are only similar with a degree such as type3 and type4 clones.

Recommendation systems using the clone detection approach takes different routes. It is possible to detect clones using token matching, graph matching or any other static analysis based approach. Others take the route of machine learning to train a model which can detect clones in a code using the information gained from software elements. Different tools can also focus on different types of objects, such as files, classes or functions. In this study functions will be compared with one another in order to detect clones.

In this paper, a function clone detection approach from another paper is implemented using Java language [1]. Basically, semantic features are fetched from functions in source codes and each function is paired with one another. Then these pairing features are represented as vectors and fed into a machine learning model. Then using Winforms [2], a simple user interface is implemented where users can input a directory where their Java source codes are in. Then after triggering the detection tool, users can see which methods have similarity with each other, what is the type of the prediction and the confidence of the prediction.

Study tries to answer the following questions:

- What are syntactic and semantic elements which can be taken from a software?
- How can a machine learning model be trained using the gathered elements?
- Is there a difference between detecting different types of clones?
- How can a basic recommendation system UI for clone detection can be implemented?

Rest of the report is organized as follows: 2nd chapter gives brief information about the related work and some of the tools used in the project. 3rd chapter describes the dataset used in the project. 4th chapter explaines the entire clone recommendation system design. 5th chapter gives information about the training and testing of the machine learning model. 6th chapter briefly shows the user interface and the tool implemented. Finally 7th chapter summarizes the study.

January 15, 2021

II. Related Work and Tools

A. Semantic Clone Detection Using Machine Learning

This study is based on the reseach done by [3] where authors used syntactic and semantic properties of software in order to build a model where code clones can be detected. Then they proceed on evaluating their model using Big IJaDataset [4] with cross validation. They also compare their results with the existing clone detection tools.

Difference between this study and the referenced paper is, first authors did not supply the paper with the syntactic and semantic properties they used. They only give very few examples in a table. So in this study I tried to find out which properties can be used to train a model.

Secondly authors only use cross validation to validate their models. They also mention that they ensured the ratio between matches and non-matches are same in each fold, which could mean that they did not try to validate the model with real dataset ratios. In this study, different scenarios, both equal, and real-world ratios are tested with both cross-validation, and test on unseen data.

Lastly, there are no tools implemented using the found model in the referenced paper. However, I have implemented a user interface where results can be seen.

B. Clone Types

As given in the referenced paper [3], four different clone types are defined:

1) Type1: These are clones where there is an exact matching, as in 100% everything is same in two functions.

2) Type2: Different from Type1 clones, Type2 clones can have different names for variables, methods etc, however after simplifying the names the process between two methods will still be exactly same.

3) Type3: These clones are functionally same, but the ratio is not 100%. Some statements might be added-removed.

4) Type4: These are semantic clones where the clones carry similar meanings with a ratio, however the implementation and syntactic definitions does not match.

Type3 and Type4 clones are treated in the same syntactic similarity group in the dataset used. In order to differentiate them, similarity percentages needs to be checked. Following ratios are the more detailed naming for these clones:

- VST3 includes the similarity ratios between $[0.9, 1.0]$
- ST3 includes the similarity ratios between $[0.7, 0.9)$
- MT3 includes the similarity ratios between $[0.5, 0.7)$
- WT3 includes the similarity ratios between $(0.0, 0.5)$

Also in the report FP means false positives, as in, wrongly classified instances of the data. These are used as a separate instance in previous studies [3]. Therefore it is included as a separate class in this study as well. After checking a few examples in the false positive dataset, it is seen that these are part of a different clone type than the type they are assigned in. Such as a Type1 clone is wrongly classified as Type2 and is included in the dataset as false positive. But these pairs can also be completely different from one another with no clone relationship as well.

C. Tools

1) JavaParser: JavaParser [5] is a parsing tool which can be used for different purposes. In this study, it is used to read .java extension source codes from IJaDataset [4] in AST format. After getting the AST, it is possible to get a lot of information about a class, and its methods in a tree like node format. Using AST, the required features are generated which will be explained further in the report.

2) Weka (Waikato Environment for Knowledge Analysis): Weka [6] is a machine learning and data preprocessing tool which is made by Waikato University. It provides a Java API in order for developers to use a large set of tools available in the project. In this project, weka API is used for both preprocess features such as attribute reduction, and training different machine learning models such as J48, KStar etc similar to the work done in the referenced paper.

3) H2 Database: H2 Database is a database engine where users can apply the basic CRUD operations on tables and their data [7]. In this project SQL is heavily used to process and find matching between preprocessed functions and whether the pairs are clones or not.

III. BigCloneBench Dataset

Dataset used in the project is taken from the BigCloneBench [4]. Database is open to public and easy to use. Dataset includes two different resources. First one is a h2 database [7] where each clone and non-clone method pair information is given with their clone types, similarity ratios, file names, function IDs, begin-line and end-line for each function etc. Second resource is the source code files for the classes used in building the database. This recourse is called IJaDatabase.

Database is being updated from time to time, and includes different number of clones and data in it from the previous studies. Table I shows the current number of function pairs for each clone type in the database for the functions in the IJaDataset. Note that the numbers are generated on 16.01.2021 and can change the next time readers might check it.

In summary database includes over 8 million clone pairs where nearly 3% of it is false positives. Important to note that WT3 clones have the most data available with being 95% percentage. Reason for this is probably because it is more common to find non-exact clones between functions than finding exact ones. Because exact clones could be noticed more easier and can be refactored sooner.

TABLE I
CLONE TYPE DISTRIBUTIONS IN THE DATABASE.

Clone Type	Count	Overall Ratio
TYPE1	48116	0,005428748
TYPE2	4234	0,000477706
VST3	4577	0,000516406
ST3	16818	0,001897512
MT3	86341	0,009741532
WT3	8424067	0,950455959
FP	279032	0,031482137
Total	8863185	1

IJaDatabase has a lot of source files in it. After applying preprocessing for the feature extraction, the total number of functions found are around 760000. Of course, not all pairs give a method clone. Number of all possible pairs can be found with formula: $N * (N-1)/2$ and is around 288 million. So only 8 million of these are pairs which corresponds to around 2% ratio.

More detailed information about how the database is used in the experiments will be given further in the report.

IV. Recommendation System Design

A. Engine Design

Figure 1 shows the recommendation system designed in the project. It includes three parts, in the first part source codes from IJaDataset are parsed into their AST forms using javaparser library. Then semantic features are extracted and for each function, begin-end lines, class names and features are saved into the H2 database. In the H2 database, function information are matched with the existing data in order to find the function IDs. This is required because clone pairs are given as function ID pairs. Lastly function pairs and their corresponding clone results are saved in a new table with the number of required dataset for training and testing steps.

In the second step, using the feature vectors saved in the database, training and testing splits are created. More detail will be given in experiments section. After creating the splits normalization is applied on training set, because some features such as lines of code is too huge considering other features.

Then attribute selection algorithm is applied on the training set in order to find the best attributes which represent the dataset. Then a machine learning model is trained using different models such as J48, RandomForest, KStar etc. Finally test set is preprocessed using the same scaling as training set for the normalization. Then saved models are tested and results are reported. After a satisfying result is obtained, the entire project is compiled into a .jar file so it can be used by the user interface in the next step.

Third and final step describes how the user interface works. A user inputs a directory where the Java source codes are in. Feature vectors are fetched from the functions in these codes using javaparser. Then .jar file uses the saved model in order to find clone pairs from these functions. Results are shown to the user using the same interface.

B. Feature extraction

In order to extract features from source codes, a java software is written. This software takes a directory path input, then recursively iterates through every file under that directory and the possible files under the directory of that directory as well. Then for each class inside these .java extension source codes, program extracts the function bodies in AST format using the javaparser library.

After function bodies are extracted, features are generated by iterating through the node representation of the AST. The number of features extracted is 86. These features include static properties such as, the number of if conditions, number of parameters, number of assignments etc, and program dependency graph relations, such as how many assignments are done after a condition check, how many declarations are done without assignments and so on. Full list of features can be found at the end of the report in Table VIII.

After obtaining the features for each function, they are saved into the H2 Database with additional information such as the classname, filename, directory name in order to identify their function IDs which already exists in the database.

In order to use the functions as a training model data, they need to be represented in pairs. Representation of a function pair is as follows: There exists two function X and Y with their features from 1 to 86. The ground truth value for whether X and Y is a clone or not is obtained from the database by matching the IDs of X and Y with the existing clone results. Then a clone feature vector can be represented as:

- $[X_1, X_2, X_3..., X_{86}, Y_1, Y_2, Y_3..., Y_{86}, clonetype]$

As mentioned in the referenced paper, it could also be possible to use a different representation such as $X_1 * Y_1, X_2 * Y_2, ...$, however, while this representation is less costly in terms of storage and training time, it also reduces the information gained from these functions. Therefore it is not used.

Finally these vectors are generated in the H2 Database using SQL language and saved in a separate table, which then can be used to generate train and test data for the training in the next part.

C. Model Training

In order to train the model first, the feature vectors saved in the database are fetched using weka API. Number of features fetched from database is nearly 1.8 million. Numbers taken from each clone type can be seen in Table II. These numbers are gathered so that the minimum train-test split, which is for Type2 clones, has a ratio of 80% - 20%.

TABLE II
REAL CLONE TYPE DISTRIBUTIONS FOR TRAIN AND TEST DATA.

Clone Type	Train Count	Test Count	Total
TYPE1	3360	38184	41544
TYPE2	3360	840	4200
VST3	3360	908	4268
ST3	3360	3337	6697
MT3	3360	17130	20490
WT3	3360	1671284	1674644
FP	3360	55358	58718
SUM	23520	1787041	1810561

Then first without touching the test set, train set is normalized. This is done because the difference between some feature distribution is too high, and it has been noticed that the normalization improved the performance of the model. Secondly, since the number of attributes were too high, and some of them might not be related to whether a function is clone or not, an attribute filter algorithm called CfsSubsetEval is used. There are a lot of attribute filtering algorithms, however, CfsSubsetEval seemed to be used in other studies as well, therefore it is preferred here. It is also possible that some attributes do not contain enough information to represent the clone relationship, such as not correlating enough. Some attributes might also correlate with each other too much that one of them needs to be cut off. These preprocessing steps reduces the number of attributes and improves the overall performance for the model.

After obtaining the reduced feature vectors, machine learning model is trained using different models. The list will be given as result in the experiments section. After all the training is done, best model is chosen and saved in harddisk. Now it is possible to use this model whenever the users requests.

In order to use the test set, first test sets scale is normalized using the scaling applied for the training set. This is done so that the further test sets will have the same scaling and will not affect the results in an unexpected way. Both test results and cross-validation results are reported in the experiments section.

D. Using Saved Model

The best model acquired in the previous section can be used by the written java software anytime a user requires. This is achieved by compiling the software into a .jar format. Then this .jar file can easily be called with a directory input by any application using a command line arguments. Software saves the function pair results in a directory which then can be read by any .txt reading code piece to show the results to user. Format of the results after code detection is as follows:

[SourcePath1, SourceFileName1, BeginLine, EndLine, SoucePath2, SourceFileName2, BeginLine, EndLine, Probability, CloneType]

Information about the detection tool will be given briefly in a further section.

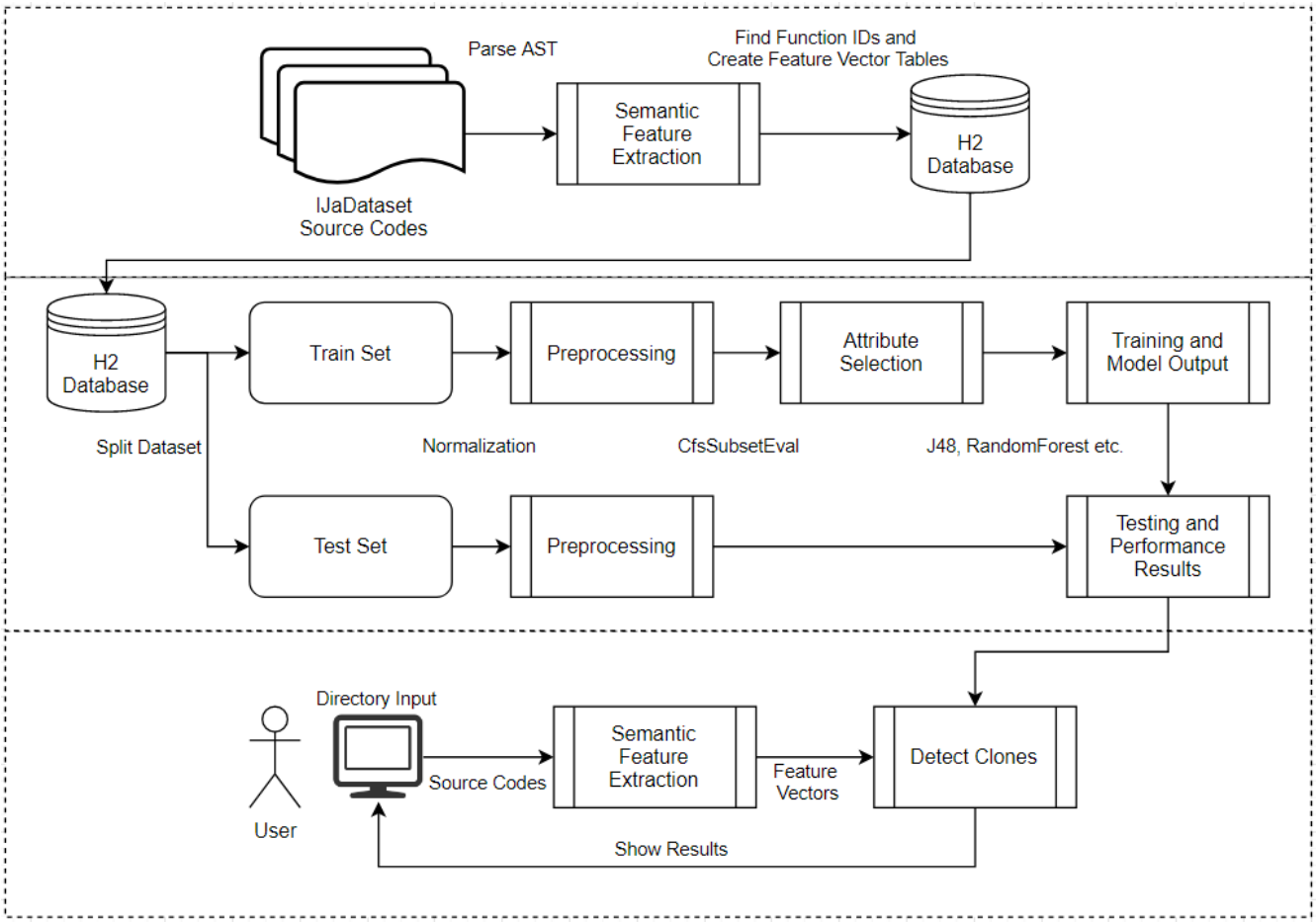

Fig. 1. Recommendation engine design.

V. Experiments

Model is evaluated using two different feature sets, one is the default 10 features given in the referenced paper as the most widely used basic approach, secondly the feature set generated in this study. Both 10-fold cross validation, and train-test splitting is used. Also train-test splits include two different distributions, first being the same number of elements for the test cases, second being the real ratio between test cases according to the real clone benchmark dataset. Each combination is trained using three different models, J48, RandomForest, and RandomCommittee. These models are selected based on their performance in previous studies. For each evaluation, recall, precision, and F-measures are given.

Table III shows the number of clones used in evaluating for the equal dataset distribution case. Notice that unlike Table II, this table presents each clone type and false positives as equals to one another, therefore it does not represent the real world scenario.

Table IV shows the results for 10-Fold Cross Validation experiments. These results are given for the sake of showing what is expected from the train-test split results. Also the original referenced paper conducted their experiments with cross validation method, therefore it is seen necessary to include them here as well. From these results, it can be seen that for all models, using all features perform better than using only 10 features. Also for all feature models, RandomForest performs better than others. Model can predict Type1, Type2, and False positives clearly, however its performance falls when Type3 clones are considered.

TABLE III
Equal clone type distributions for train and test data.

Clone Type	Train Count	Test Count	Total
TYPE1	3360	840	4200
TYPE2	3360	840	4200
VST3	3360	840	4200
ST3	3360	840	4200
MT3	3360	840	4200
WT3	3360	840	4200
FP	3360	840	4200
SUM	23520	5880	29400

Table V shows the results for the equal train-test dataset split given in the Table III. From the results, it can be seen that using all features give better performance than using default ten features. However unlike cross validation results, here it can be seen that RandomCommittee gives better performance than RandomForest for all features MT3 and WT3. But the difference is not too much, and since the test case does not mirror the real ratios it is possible that the results are affected from the random number generator used during splitting.

Table VI shows the results for the train-test dataset which is

TABLE IV
RESULTS FOR 10-FOLD CROSS VALIDATION.

Algorithm	Features	Clone Type	Precision (%)	Recall (%)	F Measure (%)
J48	Default Ten Features	TYPE1	90,7	93,2	91,9
		TYPE2	82,7	88,8	85,6
		VST3	76,4	81	78,6
		ST3	74,3	68,6	71,3
		MT3	73,5	72,4	72,9
		WT3	84,9	78,4	81,5
		FP	94,5	95,4	95
		Overall	82,4	82,5	82,4
	All Features	TYPE1	98,8	97,9	98,3
		TYPE2	96,9	98,1	97,5
		VST3	93,6	93,6	93,6
		ST3	84,1	86,8	85,4
		MT3	79,6	80,1	79,9
		WT3	89,3	85,3	87,2
		FP	97,1	97,3	97,2
		Overall	91,3	91,3	91,3
Random Forest	Default Ten Features	TYPE1	92,9	93,3	93,1
		TYPE2	85	88,8	86,9
		VST3	81,2	82,1	81,6
		ST3	80,8	71	75,6
		MT3	78,7	82,6	80,6
		WT3	89,5	89,5	89,5
		FP	96,4	97,7	97
		Overall	86,4	86,4	86,3
	All Features	TYPE1	99,9	97,4	98,6
		TYPE2	96,7	95,7	96,2
		VST3	94,2	91,7	92,9
		ST3	91,3	87,7	89,5
		MT3	82	90,4	86
		WT3	90,6	90,9	90,7
		FP	98,9	98,6	98,8
		Overall	93,4	93,2	93,2
Random Committee	Default Ten Features	TYPE1	92,1	93,4	92,7
		TYPE2	83,9	89	86,4
		VST3	79,6	82	80,8
		ST3	79,1	71,3	75
		MT3	77,7	79,9	78,8
		WT3	89,9	87	88,4
		FP	97,1	96,9	97
		Overall	85,6	85,6	85,6
	All Features	TYPE1	99,6	97,5	98,5
		TYPE2	95,9	96,2	96
		VST3	93,4	92,1	92,7
		ST3	89	88,5	88,7
		MT3	81,8	87,9	84,7
		WT3	91,1	88,6	89,8
		FP	98,8	98	98,4
		Overall	92,8	92,7	92,7

TABLE V
RESULTS USING TRAIN-TEST SPLIT FOR EQUAL NUMBER OF CLONE TYPES.

Algorithm	Features	Clone Type	Precision (%)	Recall (%)	F Measure (%)
J48	Default Ten Features	TYPE1	90,6	92,7	91,6
		TYPE2	81,3	89,3	85,1
		VST3	75,8	80,1	77,9
		ST3	76,2	68,6	72,2
		MT3	73,7	74,4	74,1
		WT3	87,5	78,9	83
		FP	95,3	96,2	95,7
		Overall	82,9	82,9	82,8
	All Features	TYPE1	98,9	96,8	97,8
		TYPE2	96,6	97,6	97,1
		VST3	94,7	94	94,4
		ST3	85,8	86,2	86
		MT3	78,3	81,9	80
		WT3	89,2	86	87,6
		FP	97,6	98	97,8
		Overall	91,6	91,5	91,5
Random Forest	Default Ten Features	TYPE1	91,9	93	92,4
		TYPE2	84,8	89,9	87,3
		VST3	81,7	81,3	81,5
		ST3	83,1	69,6	75,8
		MT3	80,1	86,5	83,2
		WT3	91,8	90,1	90,9
		FP	95,2	98,2	96,7
		Overall	86,9	87	86,8
	All Features	TYPE1	99,9	96,5	98,2
		TYPE2	96,7	95,1	95,9
		VST3	94,5	91,5	93
		ST3	91,2	86,7	88,9
		MT3	80,7	91	85,5
		WT3	90,7	91,5	91,1
		FP	99,2	98,6	98,9
		Overall	93,3	93	93,1
Random Committee	Default Ten Features	TYPE1	92	93	92,5
		TYPE2	83,5	89,9	86,6
		VST3	79,4	81,7	80,5
		ST3	79,6	70,6	74,8
		MT3	79,2	82,9	81
		WT3	91,9	86,8	89,3
		FP	96,8	97,5	97,2
		Overall	86,1	86	86
	All Features	TYPE1	99,8	96,5	98,1
		TYPE2	96,1	96,4	96,3
		VST3	94	91,9	93
		ST3	89	87,7	88,4
		MT3	81,7	89,3	85,3
		WT3	91,1	89,4	90,3
		FP	98,8	98	98,4
		Overall	92,9	92,8	92,8

distributed according to the real ratio in the original dataset, as mentioned before this distribution can be found in Table II. From these results, it can be seen that again, using all features is better than using just 10 features. After reaching the same conclusion in three different sets, it can be said that using more features is affecting the model output positively. It could be said that using more features is not always the best choice for training a model. However, during clone detection, it is possible that any discarded features might be required to identify a clone in a completely unknown dataset. During experiments the effect of the CfsSubsetEval is also evaluated. Result is that using attribute filtering did not always changed the performance of the model positively. Therefore, for gathering the end results, CfsSubsetEval is not included in the model training process.

Second conclusion from the same table is that, between the all feature results for different models, RandomForest clearly performs better. It also has a 100% precision which makes it reach closer to the modern clone detection tools for Type1 clones. However unlike the equal test distribution case, other than Type1, WT3, and FP types, model performs poorly in terms of precision and F-measure.

Table VII shows the confusion matrix for clone types for the best model in real ratio distribution test case. This confusion matrix is important because it shows why the model performs worse for some clone types. Columns represent the classified as relationship, which means for a clone type in a row, a number of N elements are identified as clone type in a column. For example 335 clones of type1 is falsely identified as Type2.

TABLE VI
RESULTS USING TRAIN-TEST SPLIT FOR REAL RATIO BETWEEN CLONE NUMBERS.

Algorithm	Features	Clone Type	Precision (%)	Recall (%)	F Measure (%)
J48	Default Ten Features	TYPE1	77,7	93	84,7
		TYPE2	6,1	89,3	11,3
		VST3	2	83,9	3,9
		ST3	3,9	69,2	7,4
		MT3	6,5	73,9	11,9
		WT3	99,8	79,8	88,7
		FP	52,1	95,5	67,5
		Overall	96,7	80,5	87
	All Features	TYPE1	100	97,4	98,7
		TYPE2	52,8	98,2	68,7
		VST3	22,6	93,5	36,4
		ST3	9,9	86,8	17,7
		MT3	7	79,8	12,9
		WT3	99,9	86,3	92,6
		FP	70	97,4	81,4
		Overall	97,8	86,8	91,4
Random Forest	Default Ten Features	TYPE1	93	93	93
		TYPE2	35,6	89,8	51
		VST3	11,6	84,3	20,3
		ST3	22,5	70,5	34,1
		MT3	10,2	85,9	18,3
		WT3	99,9	89,1	94,2
		FP	56,3	97,6	71,4
		Overall	97,3	89,3	92,5
	All Features	TYPE1	100	97	98,5
		TYPE2	68,2	95,5	79,5
		VST3	40,4	90,5	55,9
		ST3	37,9	86,2	52,7
		MT3	10,6	90,4	19
		WT3	99,9	91,3	95,4
		FP	82,6	98,6	89,9
		Overall	98,3	91,6	94,5
Random Committee	Default Ten Features	TYPE1	90,2	93,1	91,6
		TYPE2	21,8	89,9	35,1
		VST3	6,4	83,3	11,8
		ST3	12,2	71,6	20,9
		MT3	8,7	82,5	15,8
		WT3	99,9	87,6	93,3
		FP	63,1	97,2	76,6
		Overall	97,4	87,9	91,8
	All Features	TYPE1	97,9	97,1	97,5
		TYPE2	48	96,6	64,1
		VST3	13,1	90,2	22,9
		ST3	20,9	87,1	33,8
		MT3	8,4	87,4	15,4
		WT3	99,9	88,7	94
		FP	82,9	98,1	89,9
		Overall	98,2	89,2	93

First of all as expected, most of the Typ1 and False Positive clones are identified correctly. Also none of the type1 clones are identified as a non clone, which means the model can at least identify some relationship between methods but lacks enough information to fully identify them correctly.

Second point is that as mentioned before, model is having trouble for differentiating between Type3 clones. Number of WT3 clones in the dataset is too high that, when only a fraction of it is identified as VST3, ST3 or MT3 clones, that fraction is enough to lower the precision of the model for them. It should be reminded here that the ratio difference between these clone types are too small. For example, it is highly possible that a WT3 clone with 1% of clone similarity could be identified as false positive. Same thing can be said for a WT3 clone with 49% similarity, it could be identified as MT3 clone. This close relationship makes it harder for a model which depends on features but does not include all the features required.

Figure 2 shows the previous study results taken from the referenced paper [3]. In this study, these tools are also experimented on to get results from the up-to-date dataset. However, after checking the results, it is concluded that the newly obtained results are not correlated to the old ones. Therefore, the previous results are taken directly in order to prevent giving false information.

Results show that the model designed in this study performs better than the previous studies when Type3 clones are considered. Unfortunately previous studies mostly depend on cross validation instead of train-test split, therefore making a proper comparison is difficult.

Tool	Type of Clone	Recall	Precision	F-Measure
SourcererCC	VST3	93%		92%
	ST3	61%		73%
	MT3	5%	91% (as reported)	9%
	WT3/4	0%		0%
CCFinder	VST3	62%		$\approx 61\% - 67\%$
	ST3	15 %		$\approx 24\% - 25\%$
	MT3	1%	$\approx 60\% - 72\%$ (as reported)	$\approx 2\%$
	WT3/4	0%		0%
Deckard	VST3	62%		74%
	ST3	31%		47%
	MT3	12%	**93% (as reported)**	21%
	WT3/4	1%		2%
iClones	VST3	82%		
	ST3	24%		
	MT3	0%	(Unreported)	(Unreported)
	WT3/4	0%		
NiCad	VST3	**100%**		$\approx 89\% - \mathbf{98\%}$
	ST3	**95%**		$\approx 87\% - \mathbf{95\%}$
	MT3	1%	$\approx 80\% - \mathbf{96\%}$ (as reported)	$\approx 2\%$
	WT3/4	0%		0%

Fig. 2. Previous study results directly taken from [3].

VI. CLONE DETECTION TOOL & USER INTERFACE

Figure 3 shows the designed interface for using the trained model in order to detect different clones in any .java source code directory. Right part of the image is cut off on purpose in order to increase pixel quality.

Tool is composed of 5 parts, which are numbered on the figure as well. First part is where user inputs the directory manually for the location which the .java source codes are under. Second part includes two buttons, "Select Directory", which user can use to choose a directory using the directory management system offered by the underlying OS. "Detect Clones" runs the .jar file and detects clones. Third section shows the detection results. After a user clicks on any of the detection result rows, software fetches the source codes and corresponding lines then prints them on section 4 and 5. Section 4 is the source code, and section 5 is the target code which is compared to the source.

Third part which shows the results of the detection also shows the type of prediction, confidence level of the predictor, source and target file names, paths of these files, and start-end lines for both source and target.

VII. CONCLUSION

In this study, a code clone detection approach is implemented using machine learning with semantic properties of a

TABLE VII
CONFUSION MATRIX FOR THE BEST RESULT OF REAL RATIO TEST, RANDOMFOREST ALL FEATURES. COLUMNS REPRESENT THE CLASSIFIED AS RELATION, AS IN HOW MANY OF THE CLONE TYPE OF THAT ROW IS IDENTIFIED AS THE COLUMN TYPE CLONE.

Class	TYPE1	TYPE2	VST3	ST3	MT3	WT3	nonmatch
TYPE1	37046	335	466	103	102	133	0
TYPE2	1	803	17	9	9	2	0
VST3	0	15	774	30	22	14	0
ST3	0	13	96	2878	295	56	0
MT3	0	0	53	745	15479	793	61
WT3	13	12	508	3820	129526	1525943	11463
nonmatch	0	0	0	2	95	693	54569

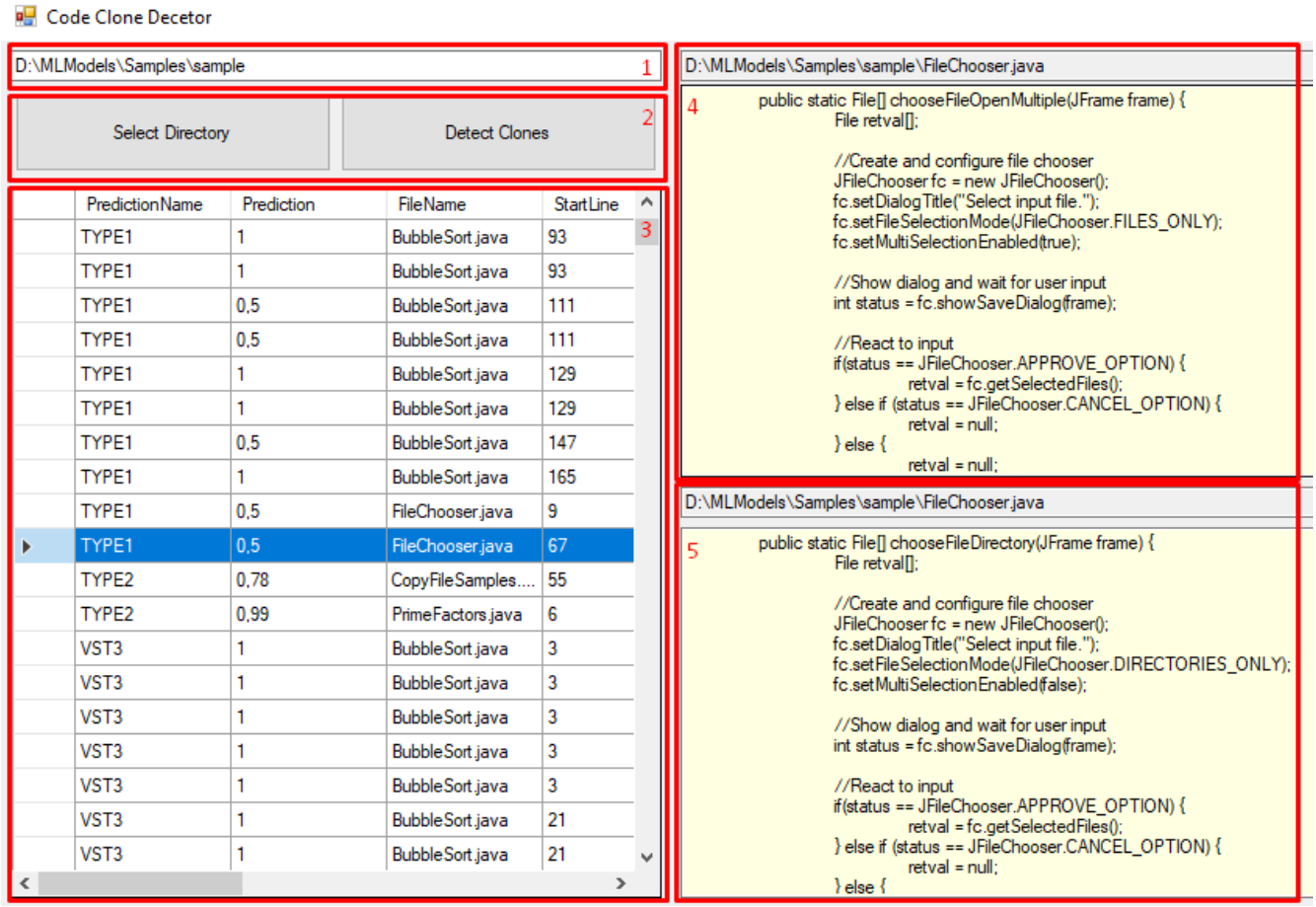

Fig. 3. Clone detection tool interface.

software. Then a simple user interface is designed to show how easy it is to use the trained model on any given project directory with .java extension files in it.

Performance of the model is not good when the number of type3 and type4 clones are too high. Model is having hard time distinguishing the differences between weak clone ratios.

In order to improve the model in the long run, more semantic and maybe syntactic features can be fetched from source codes. Similar to past works, it might also be useful to take a more token like approach in order for machine learning model to learn patterns between different clone types.

However, it should be noted that finding more features will definitely increase the cost of the model. While the static features can be fetched easily and one can use as much of them as possible, relationship dependent features do not work in the same way. It is not easy to think about every single possible path a program dependency graph can take and turn it into a feature.

Instead of a standalone software, model can also be implemented in famous Java IDE tools, such as eclipse, in order to provide more user friendly interfaces, and also increase the overall usability.

Finally, it is also possible that the distribution of a feature may not be the same across all clone types. So while clone types are splitted for their real ratio in database, it is possible that because of the differences between methods, features themselves are not distributed in a gaussian way for a clone type.

REFERENCES

[1] "Java documentation - get started," Dec 2020. [Online]. Available: https://docs.oracle.com/en/java/
[2] Adegeo, "Windows forms for .net 5 documentation." [Online]. Available: https://docs.microsoft.com/en-us/dotnet/desktop/winforms/?view=netdesktop-5.0
[3] A. Sheneamer and J. Kalita, "Semantic clone detection using machine learning," in *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2016, pp. 1024–1028.
[4] "Bigcloneeval." [Online]. Available: https://jeffsvajlenko.weebly.com/bigcloneeval.html
[5] Javaparser, "javaparser/javaparser." [Online]. Available: https://github.com/javaparser/javaparser
[6] "Weka." [Online]. Available: https://www.cs.waikato.ac.nz/ml/weka/
[7] [Online]. Available: https://www.h2database.com/html/main.html

TABLE VIII
ENTIRE FEATURE LIST INCLUDED FOR REFERENCE.

Features		
Method Length	Assignments in For	While Statement Count
Parameter Count	Assignments in Foreach	Distinct Primitive Types
Declared Variable Count	Assignments in If	Distinct Class Interface Types
Constructor Call Count	Assignments in Switch	Do Statement Count
Catch Clause Count	Assignments in Try	Assert Statement Count
Switch Count	Assignments in While	Double Literal Expressions
Case Count	Declarations in For	Boolean Literal Expressions
Iterator Count	Declarations in Foreach	Char Literal Expressions
Assignment Count	Declarations in If	String Literal Expressions
Method Call Count	Declarations in Switch	Array Acess Count
Field Access Count	Declarations in Try	Conditional Expressions
Expression Count	Declarations in While	Enclosed Expressions
Distinct Type Count	Variable Declarations With Assignments	Instance of Expressions
Object Creation Count	Variable Declarations Without Assignments	Lambda Expressions
If Count	Integer Literal Expression Count	Literal Expressions
Super Call Count	Binary Expression Count	Long Literal Expressions
Return Count	Primitive Type Count	Method Reference Expressions
Try Statement Count	Line Comment Count	Normal Annotation Expressions
Constructor Parameter Count	Array Type Count	Pattern Expressions
Array Creation Count	Overriden Method Count	Single Member Annotation Expression
Token Count	Name Expression Count	This Expression Count
Cast Expression Count	Simple Name Count	Type Expression Count
Is Static	Modifier Count	Unary Expression Count
Is Abstract	Sub Parameter Count	Break Statement Count
Is Public	Marker Annotation Count	Continue Statement Count
Is Private	Block Statement Count	Yield Statement Count
Is Protected	Null Literal Count	
Is Final	Throw Statement Count	
Is Synchronized	For Statement Count	
Method Type Length	Foreach Statement Count	