Python Code Smell Detection Using Machine Learning

Natthida Vatanapakorn, Chitsutha Soomlek and Pusadee Seresangtakul
Department of Computer Science, College of Computing, Khon Kaen University
Khon Kaen, Thailand
Email: natthida.w@kkumail.com, chitsutha@kku.ac.th, pusadee@kku.ac.th

2022 26th International Computer Science and Engineering Conference (ICSEC) | 978-1-6654-9198-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSEC56337.2022.10049330

***Abstract*—Python is an increasingly popular programming language used in various software projects and domains. Code smells in Python significantly influences the maintainability, understandability, testability issues. This paper proposes a machine learning-based code smell detection for Python programs. We trained eight machine learning models with a dataset based on 115 open-source Python projects, 39 class-level software metrics, and 22 function-level software metrics. We intended to identify five code smell types in both class and function levels, i.e., *long method*, *long parameter list*, *large class long scope chaining*, and *long based class list*. Correlation-based feature selection (CFS) and logistic regression-forward stepwise (conditional) selection were employed to improve the performance of the model. This research concluded with an empirical evaluation of the performance of the machine learning approaches against the tuning machine method. The results show that the machine learning method achieved 99.72% accuracy when identifying *long method* and *long base class list*. The machine learning-based code smell detection also outperformed the tuning machine method. Moreover, we also found a set of high-impact features that contributed most when identifying each type of code smell.**

***Index Terms*—code smells, machine learning, Python**

I. Introduction

Software maintenance is critical in software development as software products are constantly changed to support user requirements and new technology [1]. Therefore, software maintenance will end after the software reaches its 'end of life'. A common problem in software maintenance that developers face is poorly-design code. As a result, the source code is difficult to understand and hard to maintain. Martin Fowler [2] defined code smells as a problem caused by design flaws.

Researchers have proposed various techniques to detect code smells in a program [3]–[8]. Machine learning is a widely used technique for detecting code smells with promising results [4], [9]–[14]. For examples, [4] and [9] demonstrated that the random forest technique outperformed other classifiers in detecting code smells. Furthermore, most of the studies have used open-source software in the Java programming language [10], whereas more recent and modern software projects tend to incline towards Python [15]. Software projects in Python also suffer from maintenance issues. As such, code smell detection for the Python programming language will play an important role in mitigating problems.

Compared to Java, there are a very limited number of studies on code smell detection in Python projects. In recent works relative to the research problem, we refer the readers to [16]–[19]. Code smell detection is a complex task, due to the lack of common definitions and subjectivity issues. Manual detection by human experts requires a lot of effort and is time-consuming and expensive.

In this research, we employ machine learning techniques to capture the human perspectives on code smells and automatically identify code smells in Python projects. We evaluated the performance of eight supervised machine learning techniques; decision tree [20], gradient boosted trees [21], random forest [22], support vector machine [23], k-nearest neighbors [24], logistic regression [25], multilayer perceptron [26], and naive bayes [27], when identifying *long method*, *long parameter list*, *large class*, *long base class list*, and *long scope chaining*. In addition, we compared the results produced by our best models to that of the tuning machine method [17]. We also investigated the high impact features that contributed most in this research context.

The main contributions of our research are as follows:

- We proposed eight machine learning models for detecting five types of code smells in Python and their performance evaluation results. We also compared the results with those from the tuning machine method [17].
- We created a dataset by collecting 61 software metrics in class and function levels from 115 Python open-source projects using Python static code analysis tools; Pysmell [17], Understand (SciTools) [28], Cohesion [29], and our handcraft program.
- We applied feature selection techniques, like correlation-based feature selection (CFS) and logistic regression-forward stepwise (LRFS) to find the most relevant features for each type of code smell.

We made the dataset and source code publicly available at https://github.com/NatthidaW/pythoncodesmell to support the research community with reproducible content.

II. Related Work

Many researchers have applied machine learning techniques to code smell detection with promising results [30]–[34]. Examples of machine learning techniques used to identify code smells are the decision tree [31]–[34], random forest [30], [33], [34], logistic regression [34], and ensemble methods [35].

In addition, hyperparameter tuning techniques were adopted to obtain better result. For example, Mhawish and Gupta [33] applied feature selection techniques; i.e., GA-CFS, GA-Naive Bayes, and grid search, to three different categories of machine learning models to compare the performance of code smell detection (i.e., *data class*, *god class*, *feature envy*, and *long method*) [33]. The results revealed that GA-CFS was the most effective in improving accuracy and tree-based models, particularly random forest that outperformed kernel-based and network-based models [33]. In later research, Yadav et al. [31] applied the grid search method to the decision tree to improve the performance of detecting blob and data class. The results showed that the decision tree could improve reliability and reduce estimation error. In their research context, the comparison results indicated that the decision tree outperformed the naive bayes when identifying blob and also outperformed J48 when identifying data class [31]. However, there were no significant differences when compared to the results of the decision tree with CFS.

Because code smell datasets are imbalanced by nature, researchers address the related issues, considerations, and potential solutions when using imbalanced data in machine learning based code smell detection [36], [37]. Pecorelli et al. [36] and Alkharabsheh et al. [37] agreed that handling an imbalanced code smell dataset in the data preprocessing step had no significant effect on the accuracy. Boutaib et al. [38] indicated that under-sampling and over-sampling may cause data errors. They, therefore, tried to overcome the imbalanced data issue at the algorithm level instead of at the data level. The results showed that their smell detector achieved an F1-score ranging from 91.23 % to 95.24 % and the value of AUC lies between 0.9273 and 0.9573.

As mentioned in Section I, while Python has been adopted in many software projects and problem domains, there is a limited number of studies in Python code smell identification, e.g., [17]–[19]. Chen et al. [17] proposed a metric-based code smell detection to identify code smells in open-source Python projects. Experience-based, statistics-based, and tuning machine strategies were employed to find the threshold values to be used in the detection rules. The results showed that the tuning machine strategy [39] achieved the best accuracy. Like other programming languages, Python also has test smells which can be found in the test code of a project. Wang et al. [18] introduced PyNose, which is an IDE plug-in for PyCharm. PyNose can detect 17 test smells in Python code. Wang et al. also proposed a new type test smell, i.e., suboptimal assert, in their study [18].

Differing from other research efforts, our work focuses on applying machine learning techniques to identify five types of Python code smells; *long method*, *long parameter list*, *large class long scope chaining*, and *long based class list*.

III. Code Smells in Python

There are various types of code smells and test smells in Python programs. For comprehensive catalogs, we refer the readers to [17]–[19]. In the research herein, we focused on the commonly found Python code smells in both class and function levels as follows:

- Long method (LM) is a method or a function with too many responsibilities, parameters, and temporary fields that lead to a lengthy and complex method or function [2].
- Long parameter list (LPL) is a method or a function with too many arguments or parameters [2].
- Large class (LC) is a class with too many responsibilities, methods, instance variables, and a duplicate code that leads to a large class [2].
- Long scope chaining (LSC) is a multiply-nested method or a function with multiple scopes [40].
- Long base class list (LBCL) is a class that contains too many base classes [17].

IV. Research Methodology

There are four major steps in our study: data collection and pre-processing, model construction, feature selection, and model evaluation. The details of each step are as follows.

A. Data Collection and Pre-processing

To conduct our empirical study, we expanded the dataset provided by Chen et al. [17] which is publicly available at https://github.com/chenzhifei731/Pysmell. [Note that the version of the dataset we used is from commit hash: 233afebac24c910be89d065ffa661ec72458a62c.] The dataset contained both positive and negative examples of ten Python code smells labeled by three Master's and two Ph.D. students with extensive experience in the Python programming language. The overview of the dataset [17] is presented in Table I; however, the dataset is not ready to be used in our research, as it provides a limited number of features. To capture the human perception of code smells, more features capable of representing the characteristics of a source code are needed. Therefore, we relied only on the labeling results of the dataset.

TABLE I
An overview of the original dataset [17]

Code smell	Training		Testing	
	positive	negative	positive	negative
Long method	2	298	25	35
Large class	11	289	28	32
Long parameter list	31	269	32	28
Long scope chaining	6	63	18	42
Long base class list	0	300	31	29
Long message chain	36	264	38	22
Long lambda function	16	184	28	32
Long ternary conditional expression	11	190	26	34
Complex container comprehension	22	278	28	32
Multiply-nested container	15	285	30	30
Total	150	2420	284	316

Table I reveals that the number of positive and negative instances in the training set are significantly different, indicating imbalanced data. In the case of the *long base class list*, there is no positive example in the training set. For these reasons, we decided not to use the same training and testing sets employed in Chen et al. [17]. Instead, we combined data from the initial training and testing sets for each code smell type in Table I and split them into training and testing sets using 10-fold

cross-validation. Moreover, according to previous studies (see Section II), it is clear that balancing the code smell dataset in the data pre-processing step had no significant effect on the accuracy. Therefore, we did not focus on handling imbalanced data in data pre-processing.

Because we employed the labeled results from [17], we needed to ensure that the collected Python projects were exactly matched to the project versions in [17]. However, Chen at al. did not provide the commit hash of each project nor did they record the version of many projects in their dataset. We, therefore, needed to search through the repository of each project manually. We also analyzed the anticipated projects using the same Pysmell tool as [17] and compared the results to the original dataset by considering the project name, path to file, start line of the class, start line of the method, and software metrics. Lastly, we obtained the matched version of each project.

After successfully collecting all open-source projects matched to their project versions in [17], we extracted 61 different class-level and function-level software metrics from the 115 open-source projects using static code analysis tools; i.e., Pysmell [17], Understand (SciTools) [28], and Cohesion [29]. In addition, we also developed our handcraft program to extract two more types of software metrics from the source code: NP (number of parameters for a function including default Arguments, Non-Keyword Arguments, and Keyword Arguments); and NPW (number of parameters for a function without Non-Keyword Arguments and Keyword Arguments). A list of software metrics derived from all static code analysis tools is presented in Table II. Due to the limitations of the static code analysis tools, we were unable to collect the statement-level metrics. As a result, we excluded the statement-related code smells from our experiments and, therefore, included five types of code smells in both class and function levels (i.e.; *long method*, *long parameter list*, *large class*, *long base class list*, and *long scope chaining*) in the new dataset.

We then merged the results obtained from the static code analysis tools, the results from our handcraft program, and the labeling results of the five code smells from [17]. The major challenge here was for each data source to describe a location or path in different formats. Therefore, some pre-processing steps were needed to ensure that the path to the file, class name, method name, start line of the class declaration, and start line of method declaration were matched; therefore, the data were correctly joined to form a new dataset.

B. Machine Learning Model for Code Smell Detection

We employed eight supervised machine learning algorithms; decision tree (DT), gradient boosted trees (GBT), random forest (RF), support vector machine (SVM), k-nearest neighbors (KNN), logistic regression (LR), multilayer perceptron (MLP), and naive bayes (NB), to identify code smells in Python projects. We used scikit-learn [41] to develop the binary classifiers.

TABLE II
EXTRACTED SOFTWARE METRICS

Tool	Class-level	Function-level
Pysmell [17]	Class line of code (CLOC) Number of base classes (NBC)	Method line of code (MLOC) Number of parameters (PAR) Depth of closure (DOC)
Understand [28]	AvgCyclomatic AvgCyclomaticModified AvgCyclomaticStrict AvgEssential AvgLine AvgLineBlank AvgLineCode (AMS) AvgLineComment CountClassBase (IFANIN) CountClassCoupled (CBO) CountClassCoupledModified CountClassDerived (NOC) CountDeclInstanceMethod (NIM) CountDeclInstanceVariable (NIV) CountDeclMethod (WMC) CountDeclMethodAll (NM) CountLine (NL) CountLineBlank (BLOC) CountLineCode (LOC, SLOC) CountLineCodeDecl CountLineCodeExe CountLineComment CountStmt CountStmtDecl CountStmtExe MaxCyclomatic MaxCyclomaticModified MaxCyclomaticStrict MaxEssential MaxInheritanceTree (DIT) MaxNesting RatioCommentToCode SumCyclomatic (WCM) SumCyclomaticModified SumCyclomaticStrict SumEssential	CountLine (NL) CountLineBlank (BLOC) CountLineCode (LOC, SLOC) CountLineCodeDecl CountLineCodeExe CountLineComment CountPath (NPATH) CountPathLog CountStmt CountStmtDecl CountStmtExe Cyclomatic (V(G)) CyclomaticModified CyclomaticStrict Essential (EV(G)) MaxNesting RatioCommentToCode
Cohesion [29]	Cohesion	-
Our handcraft program	-	Number of parameter including-default and arbitrary(*,**) arguments (NP) Number of parameter without arbitrary(*,**) arguments (NPW)

In our experiments, we used two groups of configurations. In the first group, we employed the default parameter values of scikit-learn [41]. Table III presents another group of configurations, in which we specified a set of hyperparameters for each machine learning model and employed the grid search method with nested cross-validation based on a set of hyperparameters to find the best hyperparameter values for our machine learning models, and therefore, to improve their performance. Table IV presents the performance evaluation results of the machine learning classifiers.

TABLE III
THE SETS OF HYPERPARAMETERS USED

Machine learning classifier	Set of hyperparameters
Decision Tree	'max_depth' : [None, 2, 4, 6, 8, 10, 12] 'criterion' : ['gini', 'entropy']
Gradient Boosted Trees	'learning_rate': [0.1, 0.01] 'n_estimators' : [100, 500, 1000] 'max_depth' : [3, 4, 6, 8, 10, 12]
Random Forest	'n_estimators': [5, 20, 50, 100] 'max_features': ['auto', 'sqrt'] 'max_depth': [None, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
Support Vector Machine	'C':list(range(−1, 11))
K-Nearest Neighbors	'n_neighbors' : list(range(1, 10)) 'leaf_size' : list(range(1, 50)) 'p':[1, 2]
Logistic Regression	'C':list(range(−1 ,11)) 'penalty' : ['l1', 'l2'] 'max_iter': [20, 50, 100, 200, 500, 1000] 'solver': ['lbfgs', 'liblinear'] 'class_weight': [None,'balanced']
Multilayer Perceptron	'alpha': [0.0001, 0.05] 'learning_rate': ['constant','adaptive']
Naive Bayes	'var_smoothing': $[1e-11, 1e-10, 1e-9]$

C. Feature Selection

As evidenced in Table IV, the machine learning classifiers did not perform well when inferring from all of the features

TABLE IV
PERFORMANCE EVALUATION RESULTS

Classifier	Metric	Long Method			Long Parameter List			Long Scope Chaining			Large Class			Long Base Class List		
		Precision (%)	Recall (%)	F1-score (%)	Precision (%)	Recall (%)	F1-score (%)	Precision (%)	Recall (%)	F1-score (%)	Precision (%)	Recall (%)	F1-score (%)	Precision (%)	Recall (%)	F1-score (%)
K-NN	All	92.00	85.19	88.46*	56.76	33.33	42.00**	41.67	20.83	27.78**	78.57	56.41	65.67*	25.00	3.23	5.71*
	Chen et al.	92.86	96.30	94.55**	96.23	80.95	87.93	100.00	70.83	82.93	92.50	94.87	93.67**	96.88	100.00	98.41
	CFS	96.30	96.30	96.30*	89.66	82.54	85.95*	68.75	45.83	55.00**	77.78	71.79	74.67**	87.50	67.74	76.36**
	LRFS	92.86	96.30	94.55**	90.00	85.71	87.80*	89.47	70.83	79.07**	92.50	94.87	93.67**	96.88	100.00	98.41
LR	All	85.19	85.19	85.19**	88.89	88.89	88.89**	90.48	79.17	84.44**	72.22	66.67	69.33**	90.63	93.55	92.06**
	Chen et al.	89.66	96.30	92.86**	96.23	80.95	87.93	100.00	70.83	82.93	94.12	82.05	87.67*	96.88	100.00	98.41
	CFS	89.66	96.30	92.86**	96.49	87.30	91.67*	95.45	87.50	91.30*	85.71	76.92	81.08*	96.88	100.00	98.41
	LRFS	89.66	96.30	92.86**	94.83	87.30	90.91*	88.46	95.83	92.00**	94.12	82.05	87.67*	96.88	100.00	98.41
DT	All	89.29	92.59	90.91	91.53	85.71	88.52**	91.67	91.67	91.67**	89.47	87.18	88.31**	93.75	96.77	95.24
	Chen et al.	96.43	100.00	98.18*	96.23	80.95	87.93	100.00	70.83	82.93	90.24	94.87	92.50**	96.88	100.00	98.41
	CFS	92.59	92.59	92.59*	91.67	87.30	89.43*	91.67	91.67	91.67**	89.74	89.74	89.74**	96.77	96.77	96.77
	LRFS	96.43	100.00	98.18*	91.67	87.30	89.43*	100.00	95.83	97.87	90.24	94.87	92.50**	96.88	100.00	98.41
GBT	All	92.59	92.59	92.59*	87.50	88.89	88.19*	95.65	91.67	93.62**	86.84	84.62	85.71*	96.77	96.77	96.77*
	Chen et al.	96.43	100.00	98.18	96.23	80.95	87.93	100.00	70.83	82.93	89.74	89.74	89.74	96.88	100.00	98.41
	CFS	92.59	92.59	92.59	91.67	87.30	89.43	86.96	83.33	85.11*	86.49	82.05	84.21	96.77	96.77	96.77
	LRFS	96.43	100.00	98.18	91.67	87.30	89.43*	100.00	95.83	97.87	89.74	89.74	89.74	96.88	100.00	98.41
RF	All	96.15	92.59	94.34	88.33	84.13	86.18*	90.91	83.33	86.96**	86.84	84.62	85.71*	96.88	100.00	98.41
	Chen et al.	96.43	100.00	98.18	96.23	80.95	87.93	100.00	70.83	82.93	89.74	89.74	89.74*	96.88	100.00	98.41
	CFS	92.59	92.59	92.59	91.67	87.30	89.43	91.30	87.50	89.36	91.89	87.18	89.47*	96.88	100.00	98.41**
	LRFS	96.43	100.00	98.18	92.06	92.06	92.06**	100.00	95.83	97.87	89.74	89.74	89.74*	96.88	100.00	98.41
NB	All	91.67	40.74	56.41**	17.08	87.30	28.57*	40.00	16.67	23.53**	56.86	74.36	64.44	57.45	87.10	69.23
	Chen et al.	90.00	100.00	94.74	91.07	80.95	85.71	100.00	70.83	82.93	94.29	84.62	89.19	96.88	100.00	98.41
	CFS	64.29	100.00	78.26	81.67	77.78	79.67	91.67	45.83	61.11	81.40	89.74	85.37	96.88	100.00	98.41
	LRFS	90.00	100.00	94.74	82.09	87.30	84.62	91.30	87.50	89.36	94.29	84.62	89.19	96.88	100.00	98.41
MLP	All	68.97	74.07	71.43*	47.83	17.46	25.58*	35.29	25.00	29.27**	81.25	66.67	73.24**	100.00	35.48	52.38**
	Chen et al.	96.15	92.59	94.34*	96.23	80.95	87.93	100.00	4.17	8.00*	94.29	84.62	89.19*	100.00	35.48	52.38**
	CFS	91.30	77.78	84.00**	95.92	74.60	83.93**	66.67	8.33	14.81	84.85	71.79	77.78*	100.00	29.03	45.00*
	LRFS	96.15	92.59	94.34*	96.36	84.13	89.83	50.00	4.17	7.69	94.29	84.62	89.19*	100.00	35.48	52.38**
SVM	All	100.00	7.41	13.79	0.00	0.00	0.00	0.00	0.00	0.00	86.21	64.10	73.53**	0.00	0.00	0.00
	Chen et al.	92.59	92.59	92.59	96.23	80.95	87.93	100.00	70.83	82.93	94.44	87.18	90.67	96.88	100.00	98.41
	CFS	96.30	96.30	96.30	91.53	85.71	88.52**	0.00	0.00	0.00	79.31	58.97	67.65*	100.00	32.26	48.78**
	LRFS	92.59	92.59	92.59	91.53	85.71	88.52	90.00	75.00	81.82**	94.44	87.18	90.67	96.88	100.00	98.41

(software metrics). By going through the definition of each code smell and software metrics, we suspected that some features might be not related to a certain code smell and potentially affect the outcomes. To confirm our assumption, we used the mean decrease in impurity (MDI) measure to explain the learned model.

Figure 1 shows the effect of each feature on the trained random forest models. The results confirmed that the models were capable of capturing human perceptions on the five code smells, as well as matching the high impact features to the definitions of the code smells. For example, the features producing a larger effect when classifying *long method* are related to size, e.g., method lines of code, lines of code, number of lines containing executable source code. The high impact features used to identify *long parameter list* involve the number of parameters. The results from the measurement of feature importance also indicated some less influential features.

To improve the performance of the model, we applied the correlation-based feature selection (CFS) [42] and forward stepwise (conditional) selection methods linking the logistic regression model (LRFS) to the dataset. These two feature selection techniques helped us identify the most relevant software metrics for each type of code smell. Table V shows the results obtained from the feature selections. LRFS suggested quite similar results to the features used by Chen et al. [17] while CFS indicated that more features should be included. After removing all irrelevant features, the results of the machine learning classifiers were considerably improved (see Table IV).

D. Model Evaluation

We used k-fold cross-validation (where k = 10, shuffle = True, and random_state = 42) to separate the dataset into 10-folds. The 9-folds (90% of the data) represent the training sets used for training the models and 1-fold (10% of the data) was the testing set used for testing the models. Accuracy, precision, recall, and F1-score were the performance metrics used to evaluate the performance of the binary classifiers.

TABLE V
FEATURE SELECTION RESULTS

Code Smell	Level	Chen et al.	CFS	LRFS
LM	Function	MLOC	CountStmtExe EV(G) MaxNesting NP MLOC	MLOC
LC	Class	CLOC	NL CLOC	CLOC
LPL	Function	PAR	NPW PAR	NP PAR
LSC	Function	DOC	CountStmt CountStmtDecl DOC	CountLineCodeDecl DOC
LBCL	Class	NBC	CountLineCodeDecl NBC	NBC

V. RESULTS AND DISCUSSION

This section discusses the performance evaluation results of the machine learning-based code smell detection in term of precision, recall, and F1-score. We also compared our machine learning-based approach with the tuning machine method proposed by Chen et al. [17]. [Note that all experiments were run on a machine with AMD Ryzen 7 3750H Processor, 2.30 GHz, 7.44 GB of RAM, and Windows 10 Home 64-bit operating system.]

Table IV presents the performance evaluation results of the machine learning classifiers trained on different sets of software metrics as presented in Table II and V. For a class-level code smell, all software metrics in the class level of Table II were adopted. In the case of a function-level code smell, all software metrics in the function level were employed. [Note that "*" indicates that the default parameter values of scikit-learn [41] were used; and that "**" indicates that the hyperparameters in Table III were applied to the corresponding model.]

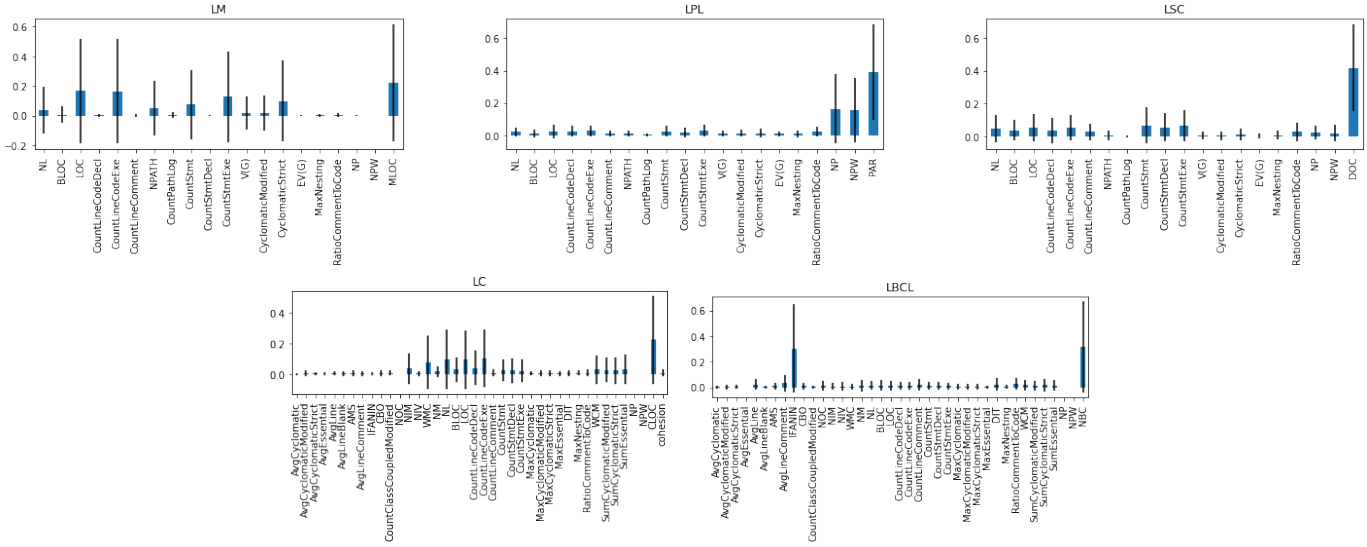

Fig. 1. Mean Decrease in Impurity (MDI) feature importance measures obtained from training the random forest classifiers

TABLE VI
COMPARISON OF OUR RESULTS WITH CHEN ET AL. [17]'S WORK

	Chen et al.						Proposed method					
code smell	Algorithm	Metric	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)	Best Algorithm	Metric	Accuracy (%)	Precision (%)	Recall (%)	F1-score (%)
LM	TM	MLOC	98.89	89.66	96.3	92.86	DT, GBT, RF	MLOC	99.72	96.43	100.00	98.18
LPL	TM	PAR	96.11	96.23	80.95	87.93	RF	PAR, NP	97.22	92.06	92.06	92.06
LSC	TM	DOC	94.57	100	70.83	82.93	DT, GBT, RF	DOC, CountLineCodeDecl	99.22	100.00	95.83	97.87
LC	TM	CLOC	98.33	90.24	94.87	92.5	K-NN	CLOC	98.61	92.50	94.87	93.67
LBCL	TM	NBC	99.72	96.88	100	98.41	K-NN, LR, DT, GBT, RF, NB, SVM	NBC	99.72	96.88	100.00	98.41

As expressed in Table IV-V, LRFS and Chen et al. [17] suggested that the same set of metrics be used to detect the *long method*, *large class*, and *long base class list*. In these cases, the machine learning classifiers produced identical performance evaluation results.

Within the *long method* and *long scope chaining*, the tree-based classifiers (DT, GBT, and RF) trained on features selected by LRFS achieved the best results. The classifiers produced precision, recall, and F1-score of 96.43%, 100.00%, and 98.18% for *long method* detection, and 100.00%, 95.83%, 97.87% for *long scope chaining* identification, respectively. In the case of the *long parameter list*, the random forest trained on features selected by LRFS outperformed the comparative models. This may be the result of the use of high-impact features (see Figure-1). In the *large class*, K-NN produced the best results. Surprisingly, In the *long base class list* code smell, all machine learning classifiers except MLP produced the same precision, recall, and F1-score at 96.88%, 100.00%, and 98.41% respectively.

The tree-based classifiers generally proved superior to the other models in detecting the five types of Python code smells with RF as the best among them. Moreover, LRFS outperformed CFS in choosing features that were the most useful in code smell detection.

Moreover, we replicated the experiment of Chen et al. [17] to compare the machine learning classifiers with the tuning machine method. [Note that we repeated every step in [17] on the new dataset and used 10-fold cross-validation technique in the experiments.] Table VI presents our comparison results. The machine learning based code smell detection outperformed the tuning machine method in the prediction of *long method*, *long parameter list*, *long scope chaining*, and *large class*. In the case of the *long base class list*, our best machine learning classifiers provided results similar to that of the tuning machine method.

VI. CONCLUSIONS

This paper proposes a machine learning-based code smell detection to support a developer in identifying five types of code smells in the Python program. We evaluated the performance of eight machine learning classifiers on the tasks of Python code smell detection. The machine learning models were trained on a dataset based on 115 open-source Python projects, 39 class-level software metrics, and 22 function-level software metrics to identify five types of Python code smells in both class and function levels (*long method*, *long parameter list*, *large class long scope chaining*, and *long based class list*). Correlation-based feature selection and logistic regression-forward stepwise (conditional) selection were employed to improve the performance of the model. The 10-fold cross-validation technique was also applied in our experiments to reduce bias. The experiment results confirmed that the machine learning-based code smell detection outperformed the tuning machine method. The random forest classifier produced the best results among the eight machine learning techniques. Moreover, we found a set of high-impact

features that contributed the most in identifying each type of code smell. To support the advancement in the research community and reproducible research, we will make the dataset and source code publicly available at https://github.com/NatthidaW/pythoncodesmell.

ACKNOWLEDGMENT

This research was partially supported by the Department of Computer Science, College of Computing, Khon Kaen University, Khon Kaen, Thailand.

REFERENCES

[1] I. Standard, "Software engineering–software life cycle processes–maintenance," *ISO Standard*, vol. 14764, p. 2006, 2006.
[2] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
[3] G. Rasool and Z. Arshad, "A review of code smell mining techniques," *Journal of Software: Evolution and Process*, vol. 27, no. 11, pp. 867–895, 2015.
[4] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, 2019.
[5] B. Bafandeh Mayvan, A. Rasoolzadegan, and A. Javan Jafari, "Bad smell detection using quality metrics and refactoring opportunities," *Journal of Software: Evolution and Process*, vol. 32, no. 8, p. e2255, 2020.
[6] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018.
[7] N. Kamaraj and A. Ramani, "Search-based software engineering approach for detecting code-smells with development of unified model for test prioritization strategies," *International Journal of Applied Engineering Research*, vol. 14, no. 7, pp. 1599–1603, 2019.
[8] A. Kaur, S. Jain, S. Goel *et al.*, "A review on machine-learning based code smell detection techniques in object-oriented software system(s)," *Recent Advances on Electrical and Electronic Engineering*, vol. 2021, no. 14, pp. 290–303, 2021.
[9] A. Al-Shaaby, H. Aljamaan, and M. Alshayeb, "Bad smell detection using machine learning techniques: a systematic literature review," *Arabian Journal for Science and Engineering*, vol. 45, no. 4, pp. 2341–2369, 2020.
[10] J. Pereira dos Reis, F. Brito e Abreu, G. de Figueiredo Carneiro, and C. Anslow, "Code smells detection and visualization: a systematic literature review," *Archives of Computational Methods in Engineering*, pp. 1–48, 2021.
[11] T. Sharma, V. Efstathiou, P. Louridas *et al.*, "Code smell detection by deep direct-learning and transfer-learning," *Journal of Systems and Software*, vol. 176, p. 110936, 2021.
[12] A. Kovačević, J. Slivka, D. Vidaković, K.-G. Grujić, N. Luburić, S. Prokić, and G. Sladić, "Automatic detection of long method and god class code smells through neural source code embeddings," *Expert Systems with Applications*, vol. 204, p. 117607, 2022.
[13] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "On the feasibility of transfer-learning code smells using deep learning," *arXiv preprint arXiv:1904.03031*, 2019.
[14] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.
[15] S. Cass. (2022) Top programming languages 2022. [Online]. Available: https://spectrum.ieee.org/top-programming-languages-2022
[16] Z. Chen, L. Chen, W. Ma, and B. Xu, "Detecting code smells in python programs," in *2016 international conference on Software Analysis, Testing and Evolution (SATE)*. IEEE, 2016, pp. 18–23.
[17] Z. Chen, L. Chen, W. Ma, X. Zhou, Y. Zhou, and B. Xu, "Understanding metric-based detectable smells in python software: A comparative study," *Information and Software Technology*, vol. 94, pp. 14–29, 2018.
[18] T. Wang, Y. Golubev, O. Smirnov, J. Li, T. Bryksin, and I. Ahmed, "Pynose: A test smell detector for python," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 593–605.
[19] J. Gesi, S. Liu, J. Li, I. Ahmed, N. Nagappan, D. Lo, E. S. de Almeida, P. S. Kochhar, and L. Bao, "Code smells in machine learning systems," *arXiv preprint arXiv:2203.00803*, 2022.
[20] P. H. Swain and H. Hauska, "The decision tree classifier: Design and potential," *IEEE Transactions on Geoscience Electronics*, vol. 15, no. 3, pp. 142–147, 1977.
[21] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of statistics*, pp. 1189–1232, 2001.
[22] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.
[23] M. A. Hearst, S. T. Dumais, E. Osuna, J. Platt, and B. Scholkopf, "Support vector machines," *IEEE Intelligent Systems and their applications*, vol. 13, no. 4, pp. 18–28, 1998.
[24] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE transactions on information theory*, vol. 13, no. 1, pp. 21–27, 1967.
[25] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant, *Applied logistic regression*. John Wiley & Sons, 2013, vol. 398.
[26] L. Noriega, "Multilayer perceptron tutorial," *School of Computing. Staffordshire University*, 2005.
[27] G. I. Webb, E. Keogh, and R. Miikkulainen, "Naïve bayes." *Encyclopedia of machine learning*, vol. 15, pp. 713–714, 2010.
[28] scitools. Understand by scitools. [Online]. Available: https://www.scitools.com/
[29] mschwager. A tool for measuring python class cohesion. [Online]. Available: https://github.com/mschwager/cohesion
[30] L. Shen, W. Liu, X. Chen, Q. Gu, and X. Liu, "Improving machine learning-based code smell detection via hyper-parameter optimization," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2020, pp. 276–285.
[31] P. S. Yadav, S. Dewangan, and R. S. Rao, "Extraction of prediction rules of code smell using decision tree algorithm," in *2021 10th International Conference on Internet of Everything, Microwave Engineering, Communication and Networks (IEMECON)*. IEEE, 2021, pp. 1–5.
[32] M. Y. Mhawish and M. Gupta, "Generating code-smell prediction rules using decision tree algorithm and software metrics," *International Journal of Computer Sciences and Engineering*, vol. 7, no. 5, pp. 41–48, 2019.
[33] M. Y. Mhawish and M. Gupta, "Predicting code smells and analysis of predictions: using machine learning techniques and software metrics," *Journal of Computer Science and Technology*, vol. 35, no. 6, pp. 1428–1445, 2020.
[34] S. Dewangan, R. S. Rao, A. Mishra, and M. Gupta, "A novel approach for code smell detection: an empirical study," *IEEE Access*, vol. 9, pp. 162 869–162 883, 2021.
[35] H. Aljamaan, "Voting heterogeneous ensemble for code smell detection," in *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2021, pp. 897–902.
[36] F. Pecorelli, D. Di Nucci, C. De Roover, and A. De Lucia, "A large empirical assessment of the role of data balancing in machine-learning-based code smell detection," *Journal of Systems and Software*, vol. 169, p. 110693, 2020.
[37] K. Alkharabsheh, S. Alawadi, V. R. Kebande, Y. Crespo, M. Fernández-Delgado, and J. A. Taboada, "A comparison of machine learning algorithms on design smell detection using balanced and imbalanced dataset: A study of god class," *Information and Software Technology*, vol. 143, p. 106736, 2022.
[38] S. Boutaib, S. Bechikh, F. Palomba, M. Elarbi, M. Makhlouf, and L. B. Said, "Code smell detection and identification in imbalanced environments," *Expert Systems with Applications*, vol. 166, p. 114076, 2021.
[39] P. F. Mihancea and R. Marinescu, "Towards the optimization of automatic detection of design flaws in object-oriented software systems," in *Ninth European conference on software maintenance and reengineering*. IEEE, 2005, pp. 92–101.
[40] A. M. Fard and A. Mesbah, "Jsnose: Detecting javascript code smells," in *2013 IEEE 13th international working conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2013, pp. 116–125.
[41] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
[42] M. A. Hall, "Correlation-based feature selection for machine learning," Ph.D. dissertation, The University of Waikato, 1999.