Code Smell Detection Research Based on Pre-training and Stacking Models

Dongwen Zhang,Shuai Song,Yang Zhang*,Haiyang Liu,Gaojie Shen

***Abstract*—Code smells detection primarily adopts heuristic-based, machine learning, and deep learning approaches. However, to enhance accuracy, most studies employ deep learning methods, but the value of traditional machine learning methods should not be underestimated. Additionally, existing code smells detection methods do not pay sufficient attention to the textual features in the code. To address this issue, this paper proposes a code smell detection method, *SCSmell*, which utilizes static analysis tools to extract structure features, then transforms the code into txt format using static analysis tools, and inputs it into the *BERT* pre-training model to extract textual features. The structure features are combined with the textual features to generate sample data and label code smells instances. The REFCV method is then used to filter important structure features. To deal with the issue of data imbalance, the Borderline-SMOTE method is used to generate positive sample data, and a three-layer Stacking model is ultimately employed to detect code smells. In our experiment, we select 44 large actual projects programs as the training and testing sets and conducted smell detection for four types of code smells: brain class, data class, God class, and brain method. The experimental results indicate that the *SCSmell* method improves the average accuracy by 10.38% compared to existing detection methods, while maintaining high precision, recall, and F1 scores. The *SCSmell* method is an effective solution for implementing code smells detection.**

***Index Terms*—Code Smell, Pre-training Model, Textual Features, Stacking Model.**

I. Introduction

Re factoring techniques can eliminate code smells without changing the code's external behavior. By detecting code smells, researchers can effectively reveal code design flaws and determine specific locations that need to be refactored. The projects of this technique can significantly improve code quality and maintainability [1].

Early automated code smell detection tools rely heavily on structure features and heuristic rules [2] to identify code smells. Typically, this method selects specific metrics and rules, and judges whether the code structure complies with these metrics and rules through manually set thresholds in order to detect whether there are code smells. Popular code smell detection tools, such as JDeodorant [3] and iPlasma [4], mainly used heuristic rule methods and judge code smells based on set thresholds. However, the different thresholds used by different tools result in a high false positive rate in detection results. Therefore, to overcome the limitations of heuristic rule methods and address these problems, many researchers have turned to machine learning methods. For example, Fontana et al. [5] demonstrated the effectiveness of machine-learning methods in detecting code smells by utilizing multiple machine-learning techniques for detection. Fabiano et al. [6] compared the performance of heuristic rule methods and machine learning methods in detecting code smells, and pointed out that the detection accuracy of both methods needs to be improved.

More and more researchers are applying deep learning technology to code smell detection due to its continuous development. Wang et al. [7] detected various code smells in the same dataset by combining method-level and class-level code smells using backpropagation neural networks. Liu et al. [8] proposed a method to automatically generate a code smell dataset and utilized deep learning techniques for code smell detection.

Current code smell detection methods have achieved some results, but there are still several areas that need to be further improved: First, existing machine learning-based code smell detection methods are relatively single and need to fully utilize the advantages of multiple machine learning models; Second, there are few publicly available code smell datasets in existing work, and most of them only contain structure features, ignoring textual features; Finally, the structure features in most work ignores the relationship between different code smells, and only some effective struture features are selected.

This paper proposes a code smell detection method called *SCSmell*, which is based on pre-training and stacking models. The method uses static analysis tools to extract structure features from the code. The code is then converted into txt format and input to a *BERT* [9] pre-training model to extract textual features. The extracted structure features and textual features are combined to generate sample data and label code smell instances. These instances are then input to a three-layer Stacking model [10]. The experiment constructed a training set and a test set using 44 large-scale real-world projects to detect four types of code smells: data class, brain class, god class, and brain method. By evaluating the effectiveness of code smell detection through answering four research questions, the experimental results demonstrate that *SCSmell* improves the average accuracy by 10.38% compared to existing detection methods. Furthermore, this method maintains high precision, recall, and F1 values, highlighting its effectiveness.

This paper has the following three contributions:

* corresponding author: zhangyang@hebust.edu.cn

The authors are with the School of Information Science and Engineering, Hebei University of Science and Technology, Shijiazhuang, Hebei, China, 050018; Dongwen Zhang and Yang Zhang are also with Hebei Technology Innovation Center of Intelligent IoT, Shijiazhuang, Hebei 050018, China.

The authors would like to thank the insightful comments and suggestions of those anonymous reviewers, which have improved the presentation. This work is partially supported by the Natural Science Foundation of Hebei under grant No.F2023208001, and the Oversea High-level Talent Foundation of Hebei under grant No.C20230358.

1) A dataset was constructed by selecting 44 large-scale real-world projects from different domains, and extracting the structure features and textual features for each projects.
2) A method for detecting code smells is proposed using pre-training and stacking models.
3) *SCSmell* was compared with six existing machine learning models and six deep learning-based code smell detection methods, and the effectiveness of *SCSmell* was validated.

II. Code Smell Detection Method

This paper proposes a research method for code smell detection that enhances precision by utilizing pre-training and stacking models. Fig. 1 displays the framework of the suggested method.

A. Overview of SCSmell

The code smell detection method *SCSmell* framework bases on pre-training and stacking models is shown in Fig. 1. First, static analysis tools extract structure features from 44 large-scale projects. Then, the code is converted to txt format and input to a *BERT* pre-training model to extract textual features.The textual features is reduced to 1 dimension using the LDA dimensionality reduction technique, and combined with the structure features to generate data samples and label code smell instances. Then, the RFECV method was used to select the features on a merit basis, and the top 10 features were selected according to their scores, and then the dataset was expanded by adding positive samples using the Borderline-SMOTE [11] method, which is used as input to a three-layer stacking model. The model is trained multiple times on the training set to obtain a well-trained classifier, which is then tested on the test set to evaluate the performance of the classifier and provide code smell detection results.[1]

B. Data Collection

We conduct research by collecting 44 real projects from GitHub. Twelve of the projects have more than 10KLOC. Table I shows the projects and their configurations, where NOC represents the number of classes, NOM represents the number of methods, and LOC represents the number of code lines.

C. Textual Features Extraction

To analyze the textual features extracted from the source code, we adopt a two-step process. First, the code is converted into a txt format using static analysis tools. Next, the converted code is processed using a pre-training model *BERT*. The SentenceTransformer class from the sentence_transformers library is utilized to load the pre-training *BERT* model and encode the input code text. This ensures that the encoding format used during model training is maintained. The model's word vector extraction function is then employed to obtain the word vector for each input vocabulary. By performing a forward pass, the corresponding hidden state vector from the final hidden layer is extracted. To consolidate all word vectors into a single fixed-length vector, average pooling is applied to the entire sequence of word vectors. Finally, the 768-dimensional vector is reduced to one dimension using the LDA dimensionality reduction technique. This enables us to obtain the vector representation required for this paper.

TABLE I
PROJECTS AND CONFIGURATIONS

Project Name	Purpose or Usage	NOC	NOM	LOC
Cayenne	Open Source Persistence Framework	2928	16997	135020
Cobertura	Java Code Coverage Reporting Tool	165	1156	14723
SPECjbb2005	Java Application Server Testing	76	747	12713
Javacc	Parser Generator	180	1487	20861
JSmooth	Executable Wrapper	101	886	10411
RxJava	Reactive Extension of Java Virtual Machine Implementation	736	4181	41273
Fitjav	Open Source Testing Program	61	456	2916
Xomoj	JMX Specification Implementation	29	266	3392
JGroups	Group Communication Tool	273	2235	15587
JBoss	Application Server	612	5269	75513
Job	Distributed Task Scheduling Framework	46	227	1638
Batik	Java-based Application Toolkit	1682	16247	166673
Xalan	XSLT Processor	968	10413	171427
Jadventure	Java Text-based Game	36	145	1215
JUnit	Unit Testing	462	3960	20645
HSQLDB	Database	548	11043	190614
Rhino	Interpreter	270	6781	79406
Blueblock	Java Text-based Game	13	88	1178
Cassandra	Row Storage with Partitioning	1066	10640	83001
JavaStud	Java Example Series Project	218	459	4229
Mmseg4j	Chinese Analyzer for Java	16	97	716
Mybaits3	Test Code	224	1003	5183
Redomor	Java Text-based Game	55	463	3359
Anthelion	Nutch Plugin for Centralized Semantic Data Crawling	357	2480	32756
Argouml	UML-based Visualization Tool	1953	17456	160295
Three	3D Engine	232	799	10542
Pixi	HTML5 2D Drawing	88	610	9513
Freecs	Open Source Testing Program	139	1404	20720
Freedomotic	Secure Internet of Things Framework	501r	3867	33857
Mylyn	Tool	2762	16470	276401
Hadoop	Distributed System Infrastructure	1340	13464	158560
Itext7	Powerful PDF Toolkit	1518	12771	124562
Jedit	Text Editor	584	7376	103503
Maven	Project Management Tool	712	4008	23435
Nutch	Java-based Search Engine	366	1983	23036
Parallelcolt	Java-based High-Performance Scientific Computing Tool	1130	14527	216685
Pmd	Source Code Analyzer	2166	9902	50510
Xerces	XSLT Processor for Converting XML Documents	964	10359	188637
Displaytag	Open Source Suite for Custom Markup	262	945	9019
Freecol	Turn-based Strategy Game	344	3066	30581
Brackets	Integrated Development Environment	160	4740	35154
Freemind	Top Free Mind Mapping Software Written in Java	532	7303	65866
Gantt	Desktop Project Scheduling and Management Tool	685	5848	40009
Drjava	Lightweight Java Programming Environment	1145	16920	147284
Total		28798	252340	2827762

D. Structural Features Extraction

We use iPlasma to select multiple structure features to detect code smells. Structure features are extracted from methods and classes in specific projects, and 24 metric standards are selected to evaluate the negative characteristics of coding for BrainClass, DataClass, and GodClass, while 21 metric standards are selected to evaluate the negative characteristics of coding for BrainMethod. The selected metrics are shown in Table II, these metrics, including the base class usage ratio (BUR), CC, CM, etc., are commonly utilized in code smell detection.

E. Feature Selection and Sample Integration

We addresse the issue of overfitting on training data and poor generalization on new data by utilizing the RFECV method for feature selection. Initially, the original feature set is fed into the model, followed by partitioning the dataset and training the model. RFECV assesses the importance of

[1] https://github.com/Lansforever/SCSmell.git/

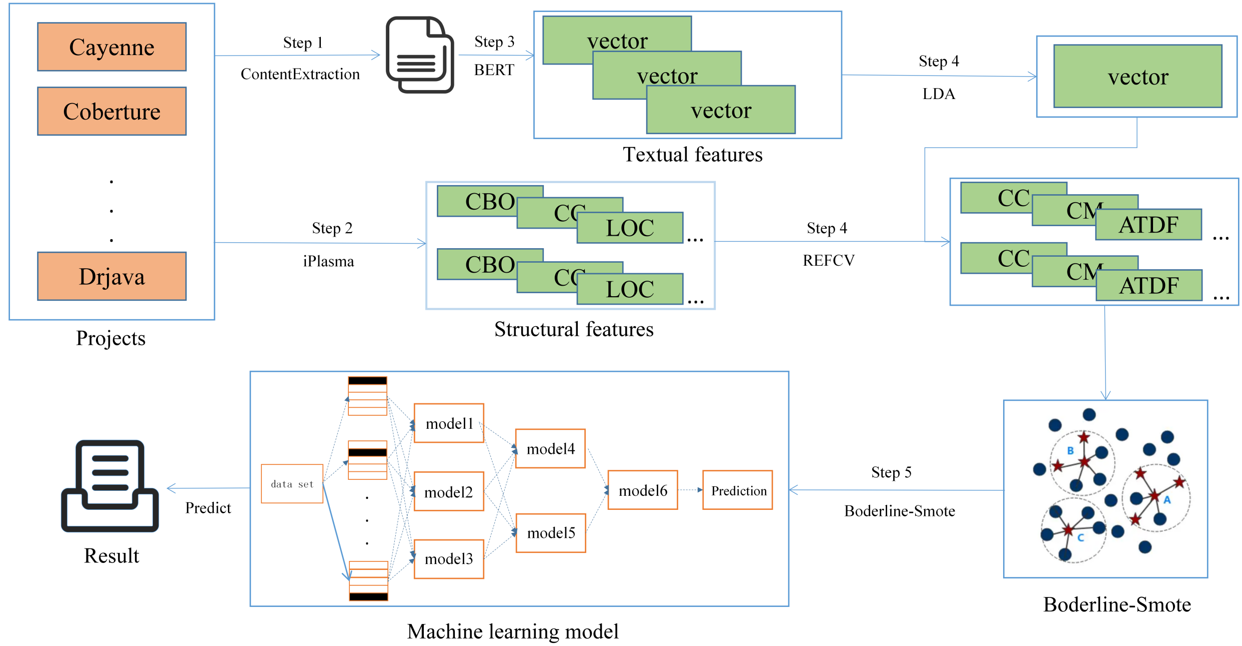

Fig. 1. Overview of SCSmell

TABLE II
METRIC

Class-level		Method-level	
Metric	Description	Metric	Description
AMW	Average Method Weight	ALD	Attribute to Local Data ratio
ATFD	Access to Foreign Data	ATFD	Access to Foreign Data
BOVM	Base Class Overriding Method	CALIN	Coupling Between Associated Classes
BUR	Base Class Usage Ratio	CC	Cyclomatic Complexity
CBO	Coupling Between Object	CCL	Cohesion among Class Attributes
CC	Cyclomatic Complexity	CDISP	Coupling Dispersion
CM	Class Methods	CEXT	Coupling Between External classes
CRIX	Change Risk Index	CINT	Coupling Between Internal classes
DAC	Data Abstraction Level	CM	Class Methods
DIT	Depth of Inheritance Tree	CYCLO	Cyclomatic Complexity
FANOUT	Fan-Out	FANIN	Fan-In
FDP	Foreign Data Providers	FANOUT	Fan-Out
GREEDY	Global access ratio EDDY	FANOUTCLASS	Fan-Out for a Class
LOC	Lines of Code	FDP	Foreign Data Providers
NAS	Number of Accessed Services	LAA	Locality of Attribute Accesses
NDU	Number of Declared Used	LOC	Lines of Code
NOA	Number of Attributes	MAXNESTING	Maximum Nesting Level
NOAM	Number of Overridden Accessor Methods	NOAV	Number Of Accessed Variables
NOD	Number of Overridden Methods	NOEU	Number of Executed Units
NOM	Number of Methods	NOLV	Number Of Local Variables
NOPA	Number of Public Attributes	NOP	Number Of Parameters
TCC	Tight Class Cohesion		
WMC	Weighted Methods per Class		
WOC	Weight of a Class		

each feature to the model by utilizing the recursive feature elimination algorithm, which involves training the model multiple times and calculating the feature's importance score. This process allows for a more accurate evaluation of feature subsets and reduces potential risks by selecting the best feature subset through cross-validation in each fold and repeating the process. Ultimately, the top 10 structure features are selected based on their scores. The selected structure features are presented in Table III.

After feature selection, the method-level dataset has over 300,000 samples, and the class-level dataset has over 20,000

TABLE III
FILTERED METRICS

Code Smell	Metrics									
BM	ALD	ATFD	CALIN	CC	CCL	CDISP	CEXT	CINT	CM	CYCLO
DC	AMW	ATDF	BOVM	BUR	CBO	CC	CM	CRIX	DAC	DIT
GC	AMW	ATDF	BOVM	BUR	CBO	CRIX	DAC	DIT	FANOUT	FDP
BC	AMW	ATDF	BOVM	BUR	CBO	CC	CM	CRIX	DAC	DIT

samples. The final dataset contains approximately 42% positive samples.

F. Positive Sample Generation

To solve the problem of low positive samples in the training data, we employ the Borderline-SMOTE method to balance the dataset. Borderline-SMOTE, an improved version of SMOTE, prioritizes the optimization of boundary samples by leveraging SMOTE and effectively handles minority class boundary samples. The application of the Borderline-SMOTE method results in an increase in the dataset's positive samples from 22% to 42%.

G. Machine Learning Model

Fig. 2 illustrates the structure of a machine learning model. This model improves upon the traditional two-layer Stacking model and implements a three-layer Stacking model with feature selection from the RFECV method. It includes primary, intermediate, and advanced models to validate the model. The primary model selects three basic models: model1 is an *SVM*, model2 is a *Logistic*, and model3 is a *Random forest*. The intermediate model chooses model2 and model4, where model2 is a *Random forest* from the first layer and model4

is a model aggregator. The advanced model selects the model aggregator from the second layer.

The Stacking model is designed in a way that each basic model is trained separately using different algorithms and hyperparameters. The output of each basic model is saved for later use. Next, the evaluation of each basic model's performance is conducted on the test set. Subsequently, the test result is incorporated as an additional feature into the intermediate model. The output of the intermediate model is then fed into the advanced model, which uses it to make predictions on new samples and produces the final prediction result. By combining multiple algorithms, the Stacking model is able to minimize errors caused by algorithm selection and effectively capture various prediction features offered by the basic models.

1) Selection of Base Model Classifiers: To validate the superiority of the selected primary model classifiers, we conducte experiments using six combinations of primary model classifiers. The experimental results are shown in Table IV. SLR represents *SVM* [12]+*Logistic* [13]+*Random forest* [14], KRA represents *KNN* [15]+*Random forest+Adaboost* [16], SLA represents *SVM+Logistic+Adaboost*, SKR represents *SVM+KNN+Random forest*, SRA represents *SVM+Random forest+Adaboost*, and KLR represents *KNN+Logistic+Random forest*.

According to the data in table IV, the SLR set performs slightly better on all metrics. This is because different features of the datasets and different datasets have different feature distributions and class distributions, which may affect the performance of the underlying model. On BM and BC datasets, so the recall of KRA combination is better than SLR combination. However, to better utilize the advantages of each base classifier, we choose the SLR combination.

2) Selection of Model Layer Number: We demonstrate through experiments that a three-layer stacking model is superior to a two-layer stacking model, with all base models being SLR. The experimental results are shown in Table V.

The data analysis from Table V indicates that the three-layer stacking model outperforms the two-layer stacking model in all evaluation metrics. Thus, we can confidently assert that the three-layer stacking model is better suited for effectively detecting code smells and identifying them.

III. Experimental Verification

This section introduces the experimental setup, research questions, evaluation metrics, and experimental results.

A. Setup

All experiments were conducted on a workstation with a 5.0 GHz Intel(R) Core(TM) i9-13900HX 4060 processor and 16GB of RAM running 64-bit Windows 11. Python version is 3.7, sklearn version is 1.0.2, numpy version is 1.21.5, pandas version is 1.4.2.

Table VI shows the parameters of all methods used in the experiments, including the degree of the polynomial kernel function (degree), the stopping criterion (tol), the amount of memory specified for training (cache_size), etc. The values of the parameters of the machine learning model were selected by their own manual adjustment, and the parameters of the deep learning model were fine-tuned based on the values derived from previous work.

B. Research Questions

We evaluate the effectiveness of the *SCSmell* method by addressing the following research questions (RQ):

RQ1: How effective is *SCSmell* at detecting code smells?

RQ2: Is the performance of the model trained by a dataset with structural and textual features better than that with barely structural or textual features? RQ3: How effective is *SCSmell* at detecting code smells compared to traditional ML models?

RQ4: How effective is *SCSmell* at detecting code smells compared to DL models?

RQ5: How does SCSmell perform in detecting bad code flavors for each part of the time performance?

C. Evaluation Metrics

We assess the efficiency of *SCSmell* by computing accuracy, precision, recall, and F1-score using formulas (1)-(4).

Accuracy is the ratio of the number of correct predictions to the total number of predictions and is calculated as follows:

$$\text{Accuracy } = \frac{TP+TN}{TP+FP+TN+FN} \tag{1}$$

True Positive (TP) represents the number of positive samples that are correctly classified as positive. False Positive (FP) represents the number of negative samples that are incorrectly classified as positive. True Negative (TN) represents the number of negative samples that are correctly classified as negative. False Negative (FN) represents the number of positive samples that are incorrectly classified as negative.

Precision represents the probability of correctly predicting positive samples among the samples predicted as positive and is calculated as follows:

$$\text{Precision } = \frac{TP}{TP+FP} \tag{2}$$

Recall represents the probability of correctly predicting positive samples among the actual positive samples in the original dataset and is calculated as follows:

$$\text{Recall } = \frac{TP}{TP+FN} \tag{3}$$

The F1-score is the weighted average of precision and recall, with values ranging from 0 to 1. A higher F1-score indicates a balance between precision and recall, with both being maximized. It is calculated as follows:

$$\text{F1} = 2 * \frac{precision * recall}{precision + recall} \tag{4}$$

In the experimental results table, Accuracy is abbreviated as Acc, Precision is abbreviated as Pre, and Recall is abbreviated as Rec.

D. Experimental Results

This section presents the experimental results.

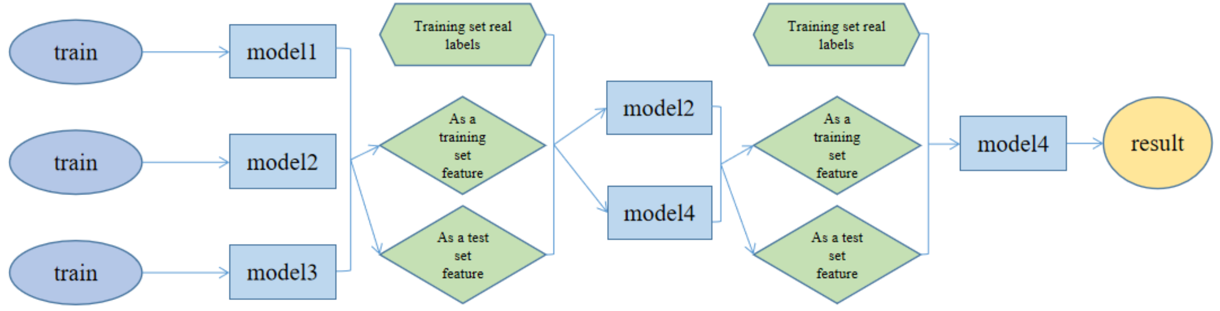

Fig. 2. Machine Learning Model

TABLE IV
RESULTS OF SELECTION OF BASE MODEL CLASSIFIERS(%)

Combination	BM				DC				GC				BC			
	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1
SKR	97.86	97.58	96.96	97.27	97.15	97.08	96.24	96.66	97.25	97.17	97.04	97.10	97.11	97.36	97.00	97.18
SLA	98.14	98.24	98.47	98.35	**98.68**	98.61	97.88	98.24	96.69	96.89	96.71	96.80	97.17	97.25	96.94	97.09
SRA	97.59	98.16	98.15	98.16	96.24	96.56	95.47	96.01	96.45	96.56	95.44	96.00	95.69	95.49	95.48	95.48
KRA	99.07	98.86	**99.13**	98.99	98.41	98.28	**98.04**	98.16	97.89	98.04	97.94	97.99	97.85	98.11	**98.40**	98.25
KLR	97.58	97.47	98.01	97.74	96.58	96.42	95.84	96.13	96.11	96.20	95.89	96.04	96.60	97.07	96.45	96.76
SLR	**99.17**	**99.31**	98.97	**99.14**	98.34	**98.83**	96.66	**98.74**	**98.23**	**98.76**	**98.42**	**98.59**	**98.47**	**99.07**	98.25	**98.66**

TABLE V
RESULTS OF SELECTION OF MODEL LAYER NUMBER(%)

Layers	BM				DC				GC				BC			
	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1
Two Layers	98.88	99.11	98.45	98.78	97.58	97.88	96.84	97.36	97.49	98.07	97.82	97.94	97.77	97.82	96.87	97.34
Three Layers	**99.17**	**99.31**	**98.97**	**99.14**	**98.36**	**98.83**	**98.66**	**98.74**	**98.23**	**98.76**	**98.42**	**98.59**	**98.47**	**99.07**	**98.25**	**98.66**

TABLE VI
MODELS AND CONFIGURATIONS

Model	Hyperparameters	Value
SVM	degree	3
	tol	0.001
	cache_size	200
Random forest	n_estimators	100
Logistic	penaltye	12
	tol	0.0001
	max_iter	100
Adaboost	learning_rate	0.05
	n_estimators	40
BP	Number of units(Dense)	3
GRU	*GRU*_units	(1,2)
ResNet	Filters in convolution layer	(64,128,256,512)
	Kernel size in convolution layer	(3)
DeepFM	Dimensionality of embedding layer	(256)
	Number of units(Dense)	(32,64,128,256)
	LSTM units	(12)
DeepSmell	*GRU*	(768,256)
	LSTM	(256,256)
	CNN	(12,32,128)
	Attention	(256,256)
	Linear_1	(256,2)
DeleSmell	Filters in convolution layer	(32,64,128,256,512)
	Kernel size in convolution layer	(6,10)
	GRU units	(1,2)
	Number of units(Dense)	(128,256)

1) Results for RQ1: To answer RQ1, seven representative large programs, Batik, Xalan, HSQLDB, Argouml, Mylyn, Parallelcolt, and Xerces, selected by *SCSmell* were used as the sample programs for this experiment. The experimental results are shown in Table VII.

Table VII shows the individual and average detection results for the seven programs. The last row indicates the overall average detection results for all seven programs.

The results in Table VII demonstrate that our method, *SCSmell*, achieve high accuracy, precision, recall, and F1 scores of 99.51%, 99.38%, 98.12%, and 98.75% respectively on the Xerces program for BM code smell. Nevertheless, there are noticeable variations across programs, possibly attributed to the significant sample size differences. In the case of BM code smell in the Xalan program, all metrics are approximately 2.15% (=98.86%-96.71%), 1.51 %(=99.29%-97.78%), 1.66% (=98.51%-96.85%), and 1.59% (=98.90%-97.31%) lower than the averages, respectively. This discrepancy can be explained by the scarcity of BM code smells in Xalan. The lower metrics observe for Xalan in DC code smell and HSQLDB in GC code smell can be attributed to the same cause. On the other hand, our method exhibites strong performance for other code smells and programs.

The experimental results indicate that *SCSmell* has high accuracy in detecting code smells and can accurately detect code smells.

2) Results for RQ2: To answer RQ2, we construct three datasets for four types of code smells in the experiment: datasets containing textual features, datasets containing structure features, and datasets containing two features.

TABLE VII
EXPERIMENTAL RESULTS FOR RQ1(%)

Program	BM				DC				GC				BC			
	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1
Batik	98.55	99.27	98.57	98.92	98.64	98.71	97.94	98.32	98.54	99.17	98.68	98.92	98.40	99.05	98.08	98.56
Xalan	96.71	97.78	96.85	97.31	97.72	98.66	98.84	98.75	98.52	98.67	98.61	98.64	99.20	99.51	99.18	99.34
HSQLDB	99.75	99.92	99.52	99.72	98.62	99.59	98.12	968.85	97.48	97.53	97.34	97.43	98.92	99.74	98.52	99.13
Argouml	99.83	99.88	99.60	99.74	98.92	99.48	99.08	99.28	98.71	98.62	97.18	97.89	98.60	99.46	98.61	99.03
Mylyn	98.82	98.92	97.95	98.43	98.04	98.51	98.11	98.31	99.17	99.38	98.81	99.09	99.12	99.16	99.09	99.12
Parallelcolt	98.88	99.87	98.97	99.42	99.07	99.22	98.53	98.87	98.73	99.36	98.49	98.92	98.39	98.18	98.11	98.14
Xerces	99.51	99.38	98.12	98.75	98.93	99.04	98.87	98.95	98.61	98.51	98.31	95.41	98.12	98.10	98.29	98.19
Mean	98.86	99.29	98.51	98.90	98.56	99.03	98.50	98.76	98.54	98.75	98.20	968.47	98.68	99.03	98.55	98.79

The metrics of the dataset containing two types of features are significantly higher than those containing only one type of features, as shown in Table XI. For the detection of the four code smells, the recall of the dataset containing two features are 18.28% (=98.97%-80.69%), 21.05% (=98.66%-77.61%), 20.01% (=98.42%-78.41%), and 17.11% (=98.25%-81.14%) higher than that containing textual features. For BC code smell, the metrics of the dataset containing only structural features are 96.53%, 95.64%, 95.73%, and 95.68%, respectively. These values are 1.94% (=98.47%-96.53%), 3.43% (=99.07%-95.64%), 2.52% (=98.25%-95.73%), and 2.98% (=98.66%-95.68%) lower than those of the dataset containing two features. This indicates that textual features plays a crucial role in *SCSmell*'s prediction results, surpassing the contribution of textual features. Furthermore, it is worth noting that *SCSmell*'s metrics at the method level slightly outperform those at the class level when detecting the four code smells. This can be attributed to the larger dataset available at the method level, which allows for better model training in this paper.

According to the experimental results, we draw the conclusion that models trained with only single feature cannot detect code smells more effectively. Datasets with two kinds of features are more suitable for detecting code smells.

3) Results for RQ3: To answer RQ3, we compare the performance of *SCSmell* with six machine learning models: *Random forest* [14], Decision Tree [17], *KNN* [15], *SVM* [12], *Logistic* [13], and *Adaboost* [16]. To ensure experiment accuracy, we employe five-fold cross-validation.

The experimental results are shown in Table IX, and the overall performance of *SCSmell* on the test program is the best. For BM code smell, the accuracy rate of *KNN* is 89.87%, which is 9.44% (=99.31%-89.87%) lower than that of *SCSmell*. The *KNN* obtains a higher recall rate at the cost of reducing the accuracy rate. For DC code smell, the indicators of *SCSmell* are 48.8% (=98.34%-49.54%), 40.16% (=98.83%-58.67%), 33.09% (=98.66%-65.57%) and 36.81% (=98.74%-61.93%) higher than *SVM*. For GC code smell, the accuracy rate, recall rate and F1 value of *SCSmell* are 98.23%, 98.42% and 98.59% respectively, which are 3.16% (=98.23%-95.07%) and 7.73% (=98.42%-90.69%), 3.84% (=98.59%-94.75%), however, the accuracy of *Decision tree* is 0.42% (=99.18%-98.76%) higher than *SCSmell*. *Decision trees* achieve higher precision at the cost of lower recall. For BC code smell, it is not difficult to find that the indicators of *Logistic* are all low, and the recall rate is 86.43%, which is 11.82% (=98.25%-86.43%) lower than *SCSmell*.

The experimental results show that *SCSmell* performs significantly better than existing methods in detecting BM, DC, GC, and BC code smells. Although *Adaboost* performs well in identifying code smells, its computational complexity is too high, and it takes a long time to process the dataset.

4) Results for RQ4: To answer RQ4, we compare the *SCSmell* model with existing deep learning models, including *BP* [18], *GUR* [19], *ResNet* [20], *DeepFM* [21], *DeepSmell* [22]and *DeleSmell* [23]. All models were trained on the dataset collected in this paper to ensure a fair comparison.

Table X presents the experimental results. For BM code smell, *SCSmell* exhibites precision, recall, and F1 scores of 99.31%, 98.97%, and 99.14% respectively, which are 18.8%, 18.85%, and 18.83% higher than *BP*'s corresponding scores of 80.51%, 80.12%, and 80.31%. In addition, when compares to *GRU*, *SCSmell* improvements in accuracy, precision, recall, and F1, elevating the scores from 83.66%, 85.01%, 84.52%, and 84.76% to 99.17%, 99.31%, 98.97%, and 99.14% respectively. Furthermore, compares to *DeepFM*, *SCSmell* achieves a substantial increase in precision, soaring from 82.53% to 99.31%. On the other hand, for DC code smell, *DeepSmell* attaines a recall rate of 87.20%, but fell short in accuracy and precision with scores of 81.47% and 80.31% respectively, which are 16.87% (=98.34%-81.47%) and 18.52% (=98.83%-80.31%) lower than *SCSmell*. Turning to GC code smell detection, *DeleSmell* yieldes higher overall metrics, yet its accuracy and precision scores are 1.16% (=98.23%-97.07%) and 1.08% (=98.76%-97.68%) lower than *SCSmell*. Lastly, in comparison to *ResNet*, *SCSmell* displayes a 7.21% (=98.25%-91.04%) enhancement in recall for BC code smell.

From the data in Table X, it can be seen that *SCSmell* has better overall performance than other deep learning methods in detecting these four code smells, which proves that *SCSmell* is more effective than the other six deep learning methods.The reason why the three-layer stacking model works better than deep learning in code bad taste detection may be because the stacking model effectively reduces overfitting and improves the generalization ability of the model. In addition, stacking models can also improve detection by integrating the prediction results of multiple base learners. In contrast, deep learning models may suffer from problems such as overfitting and underfitting, leading to poor detection results.

5) Results for RQ5: To answer RQ5, we evaluated the time performance of SCSmell on detecting the bad taste of

TABLE VIII
EXPERIMENTAL RESULTS FOR RQ2(%)

Features	BM				DC				GC				BC			
	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1
With structural features	97.11	97.97	96.76	97.36	96.22	95.65	95.86	95.75	96.88	95.98	95.11	95.54	96.53	95.64	95.73	95.68
With textual features	92.53	91.64	80.69	85.82	90.59	91.04	77.61	83.79	91.05	90.69	78.41	84.10	91.13	89.92	81.14	85.30
With two features	**99.17**	**99.31**	**98.97**	**99.14**	**98.34**	**98.83**	**98.66**	**98.74**	**98.23**	**98.76**	**98.42**	**98.59**	**98.47**	**99.07**	**98.25**	**98.66**

TABLE IX
EXPERIMENTAL RESULTS FOR RQ3(%)

Model	BM				DC				GC				BC			
	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1
Random forest	88.28	87.59	87.45	87.52	87.22	88.14	85.34	86.72	88.18	90.15	86.54	88.31	86.48	88.49	88.29	88.39
Decision tree	95.52	98.77	86.81	92.40	94.54	98.65	89.07	93.62	95.07	**99.18**	90.69	94.75	94.15	97.59	88.46	92.80
KNN	92.48	89.87	96.75	91.80	93.07	94.85	93.71	94.28	95.15	93.26	95.09	94.17	94.63	92.59	95.14	95.85
SVM	57.51	68.29	61.84	64.91	49.54	58.67	65.57	61.93	50.15	59.15	65.48	62.15	51.09	58.26	62.95	60.51
Logistic	92.41	91.18	91.85	91.51	91.81	90.76	89.65	90.20	90.17	93.48	91.59	92.53	88.81	89.91	86.43	88.14
Adaboost	94.24	93.81	95.08	94.44	96.47	96.07	95.11	95.60	96.13	95.91	96.08	95.99	95.55	95.78	95.41	95.59
SCSmell	**99.17**	**99.31**	**98.97**	**99.14**	**98.34**	**98.83**	**98.66**	**98.74**	**98.23**	98.76	**98.42**	**98.59**	**98.47**	**99.07**	**98.25**	**98.66**

TABLE X
EXPERIMENTAL RESULTS FOR RQ4(%)

Model	BM				DC				GC				BC			
	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1	Acc	Pre	Rec	F1
BP	86.34	80.51	80.12	80.31	81.65	82.15	82.59	82.37	82.05	83.50	83.15	83.32	83.69	83.47	83.28	83.37
GRU	83.66	85.01	84.52	84.76	85.66	86.42	83.24	84.80	96.11	87.14	83.09	85.07	86.66	88.01	82.49	85.16
*ResNet*A	94.29	94.59	94.29	94.44	92.54	96.58	92.47	94.48	92.16	95.68	93.41	94.53	92.08	95.49	91.04	93.21
DeepFM	85.69	82.53	89.13	85.70	91.68	96.54	83.45	89.52	90.45	93.47	84.48	88.75	91.51	97.58	82.41	89.36
DeepSmell	84.28	84.68	84.81	84.74	81.47	80.31	87.20	83.61	88.69	89.04	90.11	89.57	86.51	84.72	85.15	84.93
DeleSmell	98.51	99.01	98.62	98.81	97.16	98.51	98.08	98.29	97.07	97.68	97.49	97.58	97.61	98.96	97.56	98.26
SCSmell	**99.17**	**99.31**	**98.97**	**99.14**	**98.34**	**98.83**	**98.66**	**98.74**	**98.23**	**98.76**	**98.42**	**98.59**	**98.47**	**99.07**	**98.25**	**98.66**

the brain method code, and recorded the time consumed by SCSmell on the whole detection of the bad taste of the brain method, as shown in Table 8. As can be seen from Table 8, the total time spent by SCSmell on detecting the bad taste of brain method code is 121 minutes, with an average of 2 minutes and 45 seconds per item.The SCSmell test is the most time-consuming, accounting for more than 40% of the overall detection time. This is mainly due to the large amount of data and the more complex and time-consuming operation process. The total time for text information generation is 8min (about 11s per project on average, of which Blueblock takes the shortest time of only 2s, and Mylyn takes the longest time of 58s), and obtaining the feature information through the static analysis tool takes 39min, with an average of 53s per project.

TABLE XI
EXPERIMENTAL RESULTS FOR RQ5(MIN)

Step name	Time
Text Information Generation	8
Metrics Generation	39
Data Set Creation	23
Data Preprocessing	2
SCSmell Testing	49
Total	121

E. Threats to Validity

This section discusses the validity threats that may affect the test results in the experiment. The first internal validity threat is that high code complexity may lead to high time costs for the code smell detection tool to analyze and detect code smells. The second external validity threat is that the dataset used to detect code smells in this paper is quite large, which may increase the computational and storage resource requirements of code smell detection methods, thereby affecting the performance and efficiency of detection. For internal validity threats, the subsequent work will use modularized design to decompose the code into multiple modules, which can reduce the coupling and complexity of the code. For external validity threats, the subsequent work can use distributed computing, which distributes the dataset on multiple nodes for processing, which can improve the efficiency of computation and storage and shorten the detection time.

IV. CONCLUSIONS

In this paper, we propose a new code smell detection method called *SCSmell*. The method combines static analysis and textual features to detect code smell instances. Firstly, code structure features are obtained from multiple applications

using a static analysis tool, and these instances are labeled as code smell. Then, the textual features are transformed into word vectors using a *BERT* pre-training model. The structure features and textual features are combined to create the data samples for this study. The data samples are then evaluated using a three-layer Stacking model. Experiments are conducted on 44 large-scale open-source applications, comparing the performance of *SCSmell* with existing code smell detection methods. The experimental results demonstrate a 10.38% improvement in average precision with *SCSmell*, while maintaining high precision, recall, and F1 values. These findings indicate that *SCSmell* effectively implements code smell detection. Future work will involve the verification of this method's utility in detecting other code smells, as well as further improvements to enhance its performance. In future work, we will create visual charts or examples to better understand the test results.

REFERENCES

[1] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 482–482, 2018.

[2] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2009.

[3] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Jdeodorant: identification and application of extract class refactorings," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 1037–1039, 2011.

[4] R. Marinescu, "Measurement and quality in object-oriented design," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pp. 701–704, IEEE, 2005.

[5] 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, pp. 1143–1191, 2016.

[6] F. Pecorelli, F. Palomba, D. Di Nucci, and A. De Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pp. 93–104, IEEE, 2019.

[7] S. Wang, Y. Zhang, and J. Sun, "Detection of bad smell in code based on bp neural network," *Computer Engineering*, vol. 46, no. 10, pp. 216–222, 2020.

[8] H. Liu, Z. Xu, and Y. Zou, "Deep learning based feature envy detection," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pp. 385–396, 2018.

[9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[10] D. H. Wolpert, "Stacked generalization," *Neural networks*, vol. 5, no. 2, pp. 241–259, 1992.

[11] H. Han, W.-Y. Wang, and B.-H. Mao, "Borderline-smote: a new over-sampling method in imbalanced data sets learning," in *International conference on intelligent computing*, pp. 878–887, Springer, 2005.

[12] S.-i. Amari and S. Wu, "Improving support vector machine classifiers by modifying kernel functions," *Neural Networks*, vol. 12, no. 6, pp. 783–789, 1999.

[13] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*, vol. 4. Springer, 2006.

[14] T. Łuczak and B. Pittel, "Components of random forests," *Combinatorics, Probability and Computing*, vol. 1, no. 1, pp. 35–52, 1992.

[15] Y. Liao and V. R. Vemuri, "Use of k-nearest neighbor classifier for intrusion detection," *Computers & security*, vol. 21, no. 5, pp. 439–448, 2002.

[16] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[17] S. R. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE transactions on systems, man, and cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.

[18] M. Lanza and R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

[19] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[21] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He, "Deepfm: a factorization-machine based neural network for ctr prediction," *arXiv preprint arXiv:1703.04247*, 2017.

[22] Z. Y, D. CH, L. H, and G. CY, "Code smell detection approach based on pre-training model and multi-level information," *Journal of Software*, vol. 33, p. 1551, 05 2022.

[23] Y. Zhang, C. Ge, S. Hong, R. Tian, C. Dong, and J. Liu, "Delesmell: code smell detection based on deep learning and latent semantic analysis," *Knowledge-Based Systems*, vol. 255, pp. 109–737, 2022.

Dongwen Zhang received the Ph.D degree from the Department of Automation Control in Beijing Institute of Technology. She is currently a professor with the School of Information Science and Engineering, Hebei University of Science and Technology. Her research interests include software refactoring for parallelism and parallel programming.

Shuai Song is currently pursuing his master's degree in the School of Information Science and Engineering, Hebei University of Science and Technology. His research interests include parallel programming and software refactoring.

Yang Zhang received the Ph.D degree from the School of Computer, Beijing Institute of Technology. He is currently a professor with the School of Information Science and Engineering, Hebei University of Science and Technology. His research interests include software refactoring for parallelism and parallel programming.

Haiyang Liu is currently pursuing his master's degree in the School of Information Science and Engineering, Hebei University of Science and Technology. His research interests include software testing and software refactoring.

Gaojie Shen Obtained a bachelor's degree in electronic information from Anhui University of Technology, and now studies computer science at Hebei University of Science and Technology. The research direction is software refactoring and parallel programming design.