SmellDetector: Code Smell Detection and Refactoring with Large Language Models

Anonymous ACL submission

Abstract

Large Language Models (LLMs) have demonstrated impressive capabilities in many tasks such as code generation and automated program repair. However, code LLMs have ignored another important task in programmers' daily development work, which is to improve the maintainability, readability, and scalability of the program. All of these characteristics are related to code smells and we study how to improve them by detecting and removing code smells. Most works on code smells still rely on using measures formulated by experts as features, but lack of use of the rich prior knowledge contained in code LLMs. In this paper, we propose SmellDetector, a comprehensive model for both code smell detection and refactoring opportunities detection in Java. We train the model with the designed prompt which contains both code smells of class-level and method-level in the same code snippet, including more than 20 types. We achieve state-of-the-art performance on the code smell detection task and change the basic paradigm of code smell detection from binary classification problem to multi-label classification. Finally, it has been verified through experiments that good code smell detection helps to detect refactoring opportunities.

1 Introduction

Recently, Large Language Models (LLMs) have achieved impressive performance in code generation (Roziere et al., 2023), especially in the scene of algorithm competitions, and there are many commercial code models available on the market. When it comes to daily development of software engineers, it is an important but often overlooked issue that how to keep system maintainability (Löwe and Panas, 2005), or reduce the code smell. Code smells usually appear in object-oriented programming scenarios that use a large number of class structures and long codes, bring technical dept to

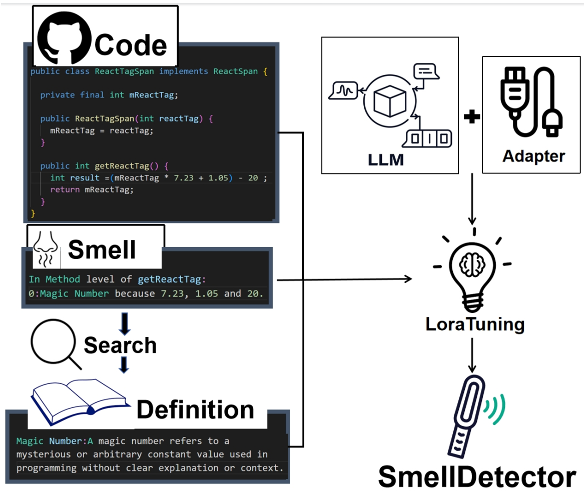

Figure 1: Flowchart of code smell detection work.

a software system (Foster et al., 2012) and harming its maintainability and evolution (Sjøberg et al., 2012). In other words, code smell does not currently affect the running of the program and output correct results, but it hinders its further development and iteration. Many researchers have paid attention to the problem of code smells as early as the millennium, and proposed that correct code smell identification can help provide reasonable refactoring locations and opportunities for code refactoring (Fowler and Beck, 1997). The traditional method calculates various metrics for the code, such as LCOM (Lack of COhesion in Methods) and NMD (number of methods declared), and comprehensively determines whether the code has a certain code smell based on whether it reaches a threshold. When machine learning and deep learning algorithms became popular, many researchers input metrics of code smell as features into the model for training to avoid the instability caused by directly selecting thresholds (Jha et al., 2019; Sharma et al., 2021). Besides, in researches of code refactoring, an important research direction is finding refactoring opportunities, which are usually treated as a binary classification and characterized by calculating various metrics of program fragments to predict whether a specific refactoring method should be used (Aniche et al., 2020).

However, some of the above methods have drawbacks: they rely on calculating measures designed by experts as features, which is not in line with the current trend of LLM development. Moreover, code refactoring opportunity detection and code smell detection lack a good connection to make them mutually reinforcing, although they are essentially information-complementing tasks.

In this paper, we present SmellDetector, a comprehensive code smell detection and elimination model, aiming to provide adapters based on LLM for detecting code smells' types and find refactoring opportunities.

We summarize our contributions below:

- We propose the first model based on code LLM fine-tuning for code smell detection and refactoring opportunities detection. Our training dataset and method is general and can be easily applied to other LLMs with greater capabilities. The model has achieved the state-of-arts in code smell detection task.
- We collect and organize the first hierarchical code smell dataset from previous datasets, which contains multiple code smells in the same code snippet, including 212,612 code smells and 22 types.
- We have experimentally proven that effective code smell detection is helpful in detecting code refactoring opportunities, and provides researchers with research ideas that the two tasks should be reasonably combined.

2 Related Work

2.1 Code Smell Detection

Code Smell is considered as inadequate implementation and design in code (Fowler and Beck, 1997), bringing various hazards, such as damaging code readability and maintainability. Beck et al. provide a detailed definition of 22 code smells through natural language. In order to automate the detection of code smell in batches, Moha et al. proposed a method of calculating program metrics and determining whether they have reached a preset threshold. Additionally, Palomba et al. use history information to detect the code smells and inspire the ideas of many researchers.

Traditional metric based methods may not be as accurate in distinguishing some fuzzy and complex code smells because of threshold. Therefore, some researchers (Fernandes et al., 2016) use machine learning methods to solve this problem, such as Bayesian (Khomh et al., 2011), SVM (Maiga et al., 2012) and Random Forest (Hall et al., 2011). Although ML algorithms perform well in smell detection, experts are still needed to perform feature extraction. On the contrary, deep learning algorithms can autonomously learn advanced features. Some researcher (Guo et al., 2019) use LSTM and CNN to extract text and metric information separately while White use RtNN and RvNN to capture features in source code and abstract syntax tree (White et al., 2016). DeepSmell (Ho et al., 2023) has reached the state-of-art performance before, but it still has a serious flaw: it needs to train a separate model for each type of code smell, which increases training and deployment costs.

2.2 Code Refactoring

When Beck et al. purposed the definitions of 22 code smells, corresponding refactoring methods have been proposed at the same time, such as Extract (method, variable...), Rename(method, variable...), Move method and so on, which are still the most commonly used (Al Dallal and Abdin, 2017). Refactoring methods and code smells have some semantic connections and they are not one-to-one correspondences and one refactoring can mitigate multiple code smells, such as Extracting Class is useful for duplicate code/clones, god classes and data blocks (Lacerda et al., 2020).

Since the usable code refactoring methods remain largely unchanged, finding an opportunity for refactoring means completing one refactoring. Many researchers use metric to search code snippets suitable for a certain type of refactoring, like the cohesion metric (Al Dallal and Briand, 2012), in-class semantic similarities metric (Bavota et al., 2014) and between-class cohesion metric (Bavota et al., 2010). Other researchers proved the effectiveness of using machine learning algorithms (Aniche et al., 2020) and neural networks (Alenezi et al., 2020) to find refactoring opportunities. When more and more reliable refactoring datasets are being proposed by integration of manual annotation and detection tools (Tsantalis et al., 2020; Moghadam et al., 2021), LLM may be a new method to help break through code refactoring.

Class smell	Number	Method Smell	Number
Feature Envy	8,337	Magic Number	32,157
Insufficient Modularization	575	Long Parameter List	8,296
Deficient Encapsulation	19,943	Complex Method	16,900
Unnecessary Abstraction	8,662	Empty catch clause	7,953
Rebellious Hierarchy	1,173	Long Method	6,366
Multifaceted Abstraction	2,325	Long Statement	34,240
Broken Modularization	8,524	Long Identifier	8,313
Cyclic Hierarchy	2,060	Complex Conditionall	9,919
Missing Hierarchy	2,320	Missing default	8,348
Blob	293	/	/
Data Class	356	/	/
Class clean	14,631	Method Clean	12,261

Table 1: The specific number of various code smells' categories in our detection dataset

Refactor Method	Number
Extract Variable	8,337
Rename Parameter	8,180
Rename Variable	8,038
Extract Method	8,511
Rename Method	8,253
Extract Class	3,338
none	8,307

Table 2: The specific number of various refactor methods' categories in our refactoring dataset

2.3 Code Large Language Models

Recently, Large Language Models (LLMs) have achieved excellent performance in code generation tasks, such as codellama (Roziere et al., 2023), Alphacode (Li et al., 2022), InCoder (Fried et al., 2022) and GPT-3 (Brown et al., 2020). Thanks to the large model's massive training data and generation capabilities brought by huge parameters, the large code model has the ability to generate code that meets the needs of the problem under the input of natural language prompts. In addition, it can also correct erroneous codes (Silva et al., 2023) According to review of code smell (Malhotra et al., 2023), the use of LLM for code smell detection is still a blank area currently.

3 Methodology

3.1 Overview

We provide an overview of the SmellDetector pipeline in Figure 1. The SmellDetector consists of two parts: two fine-tuned adapters can be plug-and-played with LLM to complete code smell detection and refactoring. In the following subsections, we will show how the it works through dataset construction, adapter training and conversations round.

3.2 Dataset Construction

We built a detection dataset consists of 97,316 files and 212,612 code smells including 22 types in class level and method level. For refactoring, we built a dataset consists of 16,000 refactoring program fragments pairs.

Detetion. Through literature research, we have selected two publicly available code smell datasets as the first step: QScore (Sharma and Kessentini, 2021) and MLCQ (Madeyski and Lewowski, 2020). These datasets have been verified for their reliability by automatic detection and expert certification, but they usually retain the relationship between a code snippet and a code smell. We clone the source code from Github and check for different code smells that share the same code snippets. When a class has multiple member methods, we will mark its class smell and method smell respectively based on the collected data. Since different code repositories may reference the same third-party code, when splitting the training and testing data, we need pairwise matching to filter out repeated identical fragments. Considering the lack of negative examples in the data set, we traversed other methods and classes in the directory where the positive examples were located and used them as negative examples(clean). In addition, there is an obvious category imbalance in the dataset. We select a certain class or method if and only if it has a code smell that appears less than k times. Based on the mean of the less frequent categories, we set k to 10000. The specific number of code smells can be found in Table 1.

Refactoring. Through literature research, we

Model Name	Feature Envy	Data Class	Long Method	
	F_1	F_1	F_1	AvgF$_1$
CodeLLama-7B	12.50	**34.99**	**15.94**	**21.14**
Baichuan2-7B	15.45	33.77	12.20	20.47
Qwen-7B	15.16	33.61	9.16	19.31

Table 3: Results of few-shot tests of three base models on the MLCQ partitioned data set

choose (Aniche et al., 2020) as source data, which consists of Apache,F-Droid, and GitHub's repositories. Since the original database only contains the corresponding files and refactoring methods before and after refactoring, we need to find pairs of actually refactoring code snippets. If it is a refactoring of the class, then we extract the class code pairs as training data, or use string matching to detect member methods whose content has changed while it is a refactoring of the method. Considering the lack of negative labeling in Aniche's open dataset, we firstly collect manually labeled negative examples from Refactoring Miner2 (Tsantalis et al., 2020). Since the number of collected negative examples is still relatively small, we use some successfully refactored program fragments as negative examples, which means this type of refactoring is no longer needed. The specific number of refactoring methods can be found in Table 2.

3.3 Adapter Training

3.3.1 Base Model Selection

We hope that in addition to having the ability to finish traditional code smell work (classification of a single smell or classification of a single level such as method or variable), the SmellDetector also has strongr generalization, which means perform better in few-shot scenarios. Therefore, we prefer to choose a base model that has both context understanding and code generation capabilities. We first selected LLMs that have performed well recently as the first step, like CodeLLama-7B (Roziere et al., 2023), Baichuan-7B (Baichuan, 2023) and Qwen-7B (Bai et al., 2023). Then, we use Chain-of-Thought(COT) to test the performance of the model on code smell detection when it is not trained. Specifically, we provide the definition of a certain code smell, positive examples, and negative examples as prompts, let the model imitate and generate predictions and reasons for code smells, and evaluate its generation quality by calculating f1 metric. The experimental results can be referred to Table 3 and we choose CodeLLama as our base model.

3.3.2 Prompt Design

Since our application scenario is different from the pre-training scenario of general code models: the input is code and the output is natural language, including the classification and explanation of code smells, we conducted some preliminary experiments using different prompts to verify the training effect, and finally selected the following prompts, and specific examples can be seen in Figure 2 and Figure 3:

In code smell detection, the input and output prompt format is that:

Detection Input:[Analyze Instruction] + [Java code]

Detection Output: [$Class$ Name] + [$Class$ Smell 1,2,...n] + [$Method_1$ Name] + [$Method_1$ Smell 1,2...m_1] + [$Method_2$ Name] + [$Method_2$ Smell 1,2...m_2] + ... + [Definition of $Smell_1$, $Smell_2$..$Smell_k$].

The design of output prompt helps the model to have the ability to output multiple smells of multiple fragments of code under a class-method structure after fine-tuning. Since there are too many types of code smells, it is not feasible to put the description of code smells as knowledge in the input prompt like the traditional COT idea, which will exceed the pre-training length limit of the most model. Therefore, we extract the code smells involved in each sample's description and put them into the output term as supervision.

In code smell refactoring, the input and output prompt format is that:

Refactoring Input:[Refactor Instruction] + [Java code]

Refactoring Output:[Refactoring Name] + [Refactoring code]

The output prompt is designed to make the model have the ability to identify application refactoring opportunities and specific refactoring at the

Detection Prompt

Input:

Please read the following Java class code and analyze the code smell of class level and method level separately, pointing out the reasons:

```
public class AlphaClass {
  //...class variables
  public methodA(Entity a) {
    // statement 1
    // statement 2
  }
  ...
}
```

Output:

We analyze the code smells at the class level and method level separately as follows:

In Class level of AlphaClass:
0:There is no class level's code smell and it's clean.

In Method level of methodA:
1:Long Statement because ...//reason 1

The following is my reference standard for determining whether the smell of the above code exists:

clean: ... // Clean smell definition
Long Statement: ... // Long statement definition

Figure 2: Example of Code Smell Detection Dataset.We added the definition and description of predicted code smells as output supervision, with the purpose of enhancing the model's understanding of code smells.

Refactoring Prompt

Input:

Read the following Java code, there are some code smells in it: **Magic Number**, **Long Method**. Please identify the appropriate refactor method and use it to **refactor** the code and remove the code smell:

```
public class NeedRefactorClass {
  //...class variables
  public methodBIG(Entity a, Entity b) {...}
}
```

Output:

The appropriate refactor method is **Extract Method** and there is the code after refactoring:

```
public class NeedRefactorClass {
  //...class variables
  public methodA(Entity a, Entity b) {methodB(a,b)...}
  public methodB(Entity a, Entity b) {...}
}
```

Figure 3: Example of Code Smell Refactoring Dataset

same time. Based on the conjecture that correct code smell information helps the refactoring model predict refactoring opportunities, we use the smell detection model trained previously to perform smell detection on the samples in the training set of the refactoring model, and add the smell information to the refactoring model.The input prompt is adjusted to:

Refactoring Adjusted Input:[Refactor Instruction] + [Name of $Smell_1$, $Smell_2..Smell_k$] + [Java code]

3.4 Advice for Refactoring

In addition to directly adding smell names to fine-tune the refactoring opportunity detection model, we also considered another way to utilize code smell detection. Specifically, we use a code smell detection model to detect the code of each refactored example and classify it into three levels:

0: No code smell.
1: Only method-level code smells.
2: Exists class-level code smells.

For each predicted refactoring method, we obtain the code smell level it can solve through its definition. We tested the performance of the refactoring opportunity detection model on 6 methods in total. The classification suggestions for refactoring methods based on the three code smell levels are:

0: Rename Parameter, Rename Variable, Rename Method and none.
1: Extract Method and Extract Variable.
2: Extract Class and Extract Method.

We only consider examples for which the actual prediction matches the opinion on the refactor, i.e., we ignore examples for which the actual predicted reconstruction method is not in the refactoring set corresponding to the code smell levels.

3.4.1 Model Training

We use QLora-tuning (Dettmers et al., 2023) to train SmellDetector, which means inserting several new parameters, called adapters, to the base of the original model. During training, the parameters of the original model are frozen and only the parameters of the adapter are updated. Instead of fine-tuning the LLM, lora-based method can achieve good performance on relatively small datasets.

4 Experiment

We conducted the following experiments, including two different topics and corresponding data sets: code smell detection and refactoring opportunity detection. The purpose of the experiment is to explore the following questions: (1)How does the adapter obtained by fine-tuning based on the large code model perform in the two tasks of code smell detection and code refactoring opportunity detection after combined with the designed

CodeSmellType	Model	Detect Metric		
		Precision	Recall	F_1
Complex Method	DeepSmell	0.731	0.779	0.754
	AE-Dense	0.483	0.630	0.547
	SmellDetector(not tuned)	0.934	0.366	0.526
	SmellDetector(tuned)	0.995	0.925	**0.956**
Complex Conditional	DeepSmell	0.575	0.604	0.589
	AE-Dense	0.170	0.387	0.237
	SmellDetector(not tuned)	0.999	0.519	0.683
	SmellDetector(tuned)	0.998	0.989	**0.994**
Multifaceted Abstraction	DeepSmell	0.287	0.272	0.279
	AE-Dense	0.031	0.747	0.060
	SmellDetector(not tuned)	-	-	-
	SmellDetector(tuned)	0.995	0.710	**0.831**
Feature Envy	DeepSmell	0.341	0.258	0.294
	AE-Dense	0.170	0.387	0.237
	SmellDetector(not tuned)	0.959	0.079	0.146
	SmellDetector(tuned)	0.988	0.899	**0.936**

Table 4: Comparison of our approach with other 2 baseline code smell detection methods (DeepSmell, AE-Dense) under 4 kinds of code smells. The precision, recall and f1 score of the baseline are from their paper because they trained binary-classification model for each type when we conduct experiment with a multi-classification model.

prompt.(2)How does this compare to traditional work on code smells. (3)How does it compare with using the COT method with code LLMs.

4.1 Experiment Setup

Dataset. In addition to testing on the dataset we created, we conduct code smell detection experiment on the benchmark created by (Sharma et al., 2021). Considering that the original benchmark had four types of code smells, with significant distribution imbalance and excessive negative examples, we set the maximum number of positive and negative examples for each type of code smell to 10000 and after shuffling the order, divide it into 25% as the test set. In terms of refactoring opportunity detection, we conduct experiment on the benchmark created by (Aniche et al., 2020). Considering that our current method mainly processes a single file, we selected 9 types of refactoring operations that basically complete the refactoring operation within a single file for detection.

Baseline.For the code smell detection task, we have chosen DeepSmells (Ho et al., 2023) and AE-Dense (Sharma et al., 2021) as the baseline. At the same time, we evaluate the performance of SmellDetector without secondary fine-tuning (only fine-tuned on the dataset we created) and after secondary fine-tuning (also fine-tuned on the benchmark training set). For the refactoring opportunity detection task, we have chosen (Aniche et al., 2020) as the baseline.

Metric. The two tasks of code smell detection and refactoring opportunities are actually classification problems. The SOTA work mentioned in the baseline all handles it as a binary classification problem, which means training a separate classifier for each code smell. On the contrary, we handle it as a multi-classification problem, because this is more in line with people's usage habits and can significantly reduce training and deployment costs. For each type, we report and compare the mean precision, recall and F1 score. We use classification report tools to calculate the metric in multi-label classification scene (Pedregosa et al., 2011).

Training/Inference Settings.For SmellDetector training, we set max sequence length to 6000 to handle situations where the code is particularly long, and use 2 epoch and a learning rate with 1e-4 to train the adapter. For the training parameters of qlora, we set lora rank to 64, lora alpha to 16 and lora dropout to 0.05. For SmellDetector inference, we set top p to 0.9, temperature to 0.35 and repetition penalty to 1.0. For the computing infrastructure, we use 2 NVIDIA A100-SXM4-80GB for training and 8 NVIDIA GeForce RTX 3090 for inference. The number of trainable parameters is 159,907,840 and the number of all parameters is 3,660,451,840. All training and inference processes are based on data parallelism and do not require model parallelism to expand GPU memory.

Class Smell	Detect Metric		
	Precision	Recall	F_1
Feature Envy	0.864	0.845	0.855
Insufficient Modularization	0.659	0.346	0.454
Deficient Encapsulation	0.990	0.995	0.992
Unnecessary Abstraction	0.995	0.985	0.990
Rebellious Hierarchy	0.905	0.830	0.866
Multifaceted Abstraction	0.910	0.619	0.737
Broken Modularization	0.996	0.990	0.993
Cyclic Hierarchy	0.868	0.800	0.833
Missing Hierarchy	0.922	0.858	0.889
Blob	0.833	0.897	0.864
Data Class	0.913	0.824	0.866
clean	0.916	0.953	0.934
Method Smell	**Detect Metric**		
	Precision	Recall	F_1
Magic Number	0.944	0.868	0.904
Long Parameter List	0.951	0.918	0.934
Complex Method	0.946	0.909	0.927
Empty catch clause	0.947	0.945	0.946
Long Method	0.922	0.855	0.888
Long Statement	0.892	0.826	0.858
Long Identifier	0.970	0.762	0.853
Complex Conditional	0.945	0.895	0.919
Missing default	0.957	0.926	0.941
clean	0.634	0.789	0.703

Table 5: The results of testing the code smell detection task on the dataset we organized. We treat it as a multi-label classification problem.

4.2 Result of Code Smell Detection

Comparison with baselines. The result of comparison with other baselines is in Table 4. AE-dense (Sharma et al., 2021) propose the benchmark what we use for testing and DeepSmell (Ho et al., 2023), which consists of fusion of deep convolutional and LSTM recurrent neural networks, is a state-of-the-art method on the benchmark. Due to performance limitations, the code smell detection task is treated as multiple binary classification problems, and multiple binary classification models are trained to exchange space and cost for higher classification accuracy.

From experimental data, we can see that our SmellDetector achieved better results, and we essentially tested a multi-class model on a binary dataset. The three code smell types except Multifaceted Abstraction are actually relatively common in our data set, so SmellDetector without secondary fine-tuning also achieved good precision in this benchmark, The reason why recall performs poorly is that SmellDetector(not tuned) is a classifier with more than 20 categories and treats other categories that do not belong to this benchmark as negative examples, so the recall rate is significantly lower than the accuracy rate. Since Feature Envy is a method-level code smell in this benchmark but is a class-level code smell in our dataset, SmellDetector without secondary fine-tuning does not perform well on this type. This experiment can show that SmellDetector which is based on LLM has made great progress on the task of code smell detection.

Testing in our dataset including 20 types. The result of testing in our dataset is in Table 5. Considering the paradigm of the data set we organized, a code snippet may have multiple code smells, which is a multi-label classification problem. For the sake of convenience, we do not consider the correctness judgment of the predicted cause of code smell for the time being. We only consider whether the code smell itself occurs or not, and extract the predicted classification items through string matching. For example, "[1,0,0,1,0,0,0,0,0,0]" means that in this method, the model predicts that there are two code smells, Magic Number and Empty catch clause. In addition, a sample may contain predictions for a class and multiple member methods at the same time, so we match the predictions with the class names and method names in the real labels, and use the matching code snippets as a more fine-grained calculation metric. the basic unit. Finally, we use the scikit-learn tool library to calculate the precision, recall and f1-score of multi-label classification. From experimental data, we find that classification performance is basically positively related to the amount of data and some categories with relatively clear and concise definitions are exceptions, such as blob and data class.

4.3 Result of Refactoring Opportunities Detection

Comparison with baselines. The result of comparison with other baseline is in Table 6. Aniche has proposed the benchmark consisting of Class-level, Method-level and Variable-level refactoring and Random Forest achieved the best performance in this benchmark (Aniche et al., 2020). When we fine-tune a binary-classification model for each class of refactoring method like the baseline, we achieve the state-of-the-art performance.

Testing about our refactoring methods. The result of testing about our refactoring methods is in Table 7. When we treat refactoring opportunities detection as a multi-class classification problem

Refactor Method	Random Forest			Binary Classification		
	P	R	F_1	P	R	F_1
Rename Parameter	0.99	0.99	**0.99**	0.98	0.99	0.98
Rename Variable	1.00	0.99	**0.99**	0.98	0.98	0.97
Rename Method	0.79	0.85	0.81	0.98	0.99	**0.98**
Extract Variable	0.90	0.83	0.87	0.98	0.87	**0.92**
Extract Method	0.80	0.92	0.84	0.99	0.95	**0.97**
Extract Class	0.85	0.93	0.89	0.89	0.93	**0.91**
Avg	0.89	0.92	0.90	0.97	0.95	**0.96**

Table 6: The results of testing the refactoring opportunities detection task on the benchmark created by the previous SOTA method , when Random Forest is the previous SOTA method, and Binary Classification is the method that finetuning a binary-classification model for each class, just like Random Forest.

Refactor Method	Base			+trained with smell			+advice		
	P	R	F_1	P	R	F_1	P	R	F_1
Rename Parameter	0.56	0.48	0.51	0.50	0.59	**0.54**	0.53	0.53	0.53
Rename Variable	0.36	0.39	**0.37**	0.37	0.31	0.34	0.34	0.37	0.36
Rename Method	0.74	0.38	0.50	0.54	0.44	0.49	0.73	0.47	**0.57**
Extract Variable	0.63	0.52	**0.57**	0.70	0.43	0.53	0.63	0.47	0.53
Extract Method	0.45	0.26	0.33	0.44	0.18	0.26	0.43	0.42	**0.43**
Extract Class	0.73	0.41	**0.53**	0.76	0.30	0.43	0.76	0.41	**0.53**
none	0.37	0.93	0.52	0.35	0.91	0.51	0.46	0.93	**0.62**
Avg	0.54	0.48	0.48	0.52	0.46	0.45	0.56	0.52	**0.51**

Table 7: The results of testing the refactoring opportunities detection task on the dataset we organized, when Base is the lora-tuning method in Chapter 3.3.2, +trained with smell is the lora-tuning method with code smell name as additional input information, and +advice is selecting examples that comply with advice in the Base method.

and output the refactor code at the same time, the classification performance is much lower than the data in Table 6. We try to treat code smell name as additional input information and the test result shows that it fails. In our inference, the reason why it can not make sense is that the given information of code smell is too little and LLM lacks enough prior knowledge of individual code smells to judge at present. Therefore, we added analysis based on expert prior knowledge and changed the method of directly using code smell names as additional input to recommended refactoring methods based on detected code smells. For details, please refer to Chapter 3.4 .The experimental data shows its effectiveness. However, the specific refactoring performance still does not meet our expectations. How to reasonably combine the two tasks of code smell detection and refactoring opportunity detection still requires further research.

5 Conclusion

In this paper, we proposed SmellDetector, a comprehensive code smell detection and elimination model. We collect and organize the first hierarchical code smell dataset from previous datasets, which contains multiple code smells in the same code snippet, including 212,612 code smells and 22 types of class level and method level. By testing on the benchmark built by previous SOTA method, our model has achieved the state-of-art in code smell detection and we can detect four times the number of smell types than before, changing the basic paradigm of code smell detection from binary classification problem to multi-label classification. We have experimentally demonstrated that effective code smell detection helps detect opportunities for code refactoring and provide researchers with ideas for a reasonable combination of two tasks.

Limitations and Future Work

In this paper, we followed the previous research paradigm on code smell, which focused on the two tasks of code smell detection and refactoring opportunity detection. However, we lack further attempts at specific reconstruction to eliminate odors. Although we have fine-tuned the refactoring model to output refactored code, we lack powerful tools to judge whether the refactored code is effective. Simply applying natural language generated metrics, such as calculating the BLEU or ROUGE of labeling refactoring code and predicting reconstructed code, is of little significance. In the future, we should solve this problem by establishing benchmarks or proposing new metrics, so as to establish a more direct research paradigm for code smell refactoring.

References

Jehad Al Dallal and Anas Abdin. 2017. Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. *IEEE Transactions on Software Engineering*, 44(1):44–69.

Jehad Al Dallal and Lionel C Briand. 2012. A precise method-method interaction-based cohesion metric for object-oriented classes. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(2):1–34.

Mamdouh Alenezi, Mohammed Akour, and Osama Al Qasem. 2020. Harnessing deep learning algorithms to predict software refactoring. *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, 18(6):2977–2982.

Mauricio Aniche, Erick Maziero, Rafael Durelli, and Vinicius HS Durelli. 2020. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering*, 48(4):1432–1450.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Baichuan. 2023. Baichuan 2: Open large-scale language models. *arXiv preprint arXiv:2309.10305*.

Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2014. Automating extract class refactoring: an improved method and its evaluation. *Empirical Software Engineering*, 19:1617–1664.

Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2010. Playing with refactoring: Identifying extract class opportunities through game theory. In *2010 IEEE International Conference on Software Maintenance*, pages 1–5. IEEE.

Kent Beck, Martin Fowler, and Grandma Beck. 1999. Bad smells in code. *Refactoring: Improving the design of existing code*, 1(1999):75–88.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*.

Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. 2016. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–12.

Stephen R Foster, William G Griswold, and Sorin Lerner. 2012. Witchdoctor: Ide support for real-time auto-completion of refactorings. In *2012 34th international conference on software engineering (ICSE)*, pages 222–232. IEEE.

Martin Fowler and Kent Beck. 1997. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Xueliang Guo, Chongyang Shi, and He Jiang. 2019. Deep semantic-based feature envy identification. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, pages 1–6.

Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2011. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304.

Anh Ho, Anh MT Bui, Phuong T Nguyen, and Amleto Di Salle. 2023. Fusion of deep convolutional and lstm recurrent neural networks for automated detection of code smells. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 229–234.

Sudan Jha, Raghvendra Kumar, Mohamed Abdel-Basset, Ishaani Priyadarshini, Rohit Sharma, Hoang Viet Long, et al. 2019. Deep learning approach for software maintainability metrics prediction. *Ieee Access*, 7:61840–61855.

Foutse Khomh, Stephane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. 2011. Bdtex: A gqm-based bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, 84(4):559–572.

Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. 2020. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, 167:110610.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Welf Löwe and Thomas Panas. 2005. Rapid construction of software comprehension tools. *International Journal of Software Engineering and Knowledge Engineering*, 15(06):995–1025.

Lech Madeyski and Tomasz Lewowski. 2020. Mlcq: Industry-relevant code smell data set. In *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, pages 342–347.

Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Esma Aimeur. 2012. Support vector machines for anti-pattern detection. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 278–281.

Ruchika Malhotra, Bhawna Jain, and Marouane Kessentini. 2023. Examining deep learning's capability to spot code smells: a systematic literature review. *Cluster Computing*, 26(6):3473–3501.

Iman Hemati Moghadam, Mel Ó Cinnéide, Faezeh Zarepour, and Mohamad Aref Jahanmir. 2021. Refdetect: A multi-language refactoring detection tool based on string alignment. *IEEE Access*, 9:86698–86727.

Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Francoise Le Meur. 2009. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36.

Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2013. Detecting bad smells in source code using change history information. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 268–278. IEEE.

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. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. 2021. Code smell detection by deep direct-learning and transfer-learning. *Journal of Systems and Software*, 176:110936.

Tushar Sharma and Marouane Kessentini. 2021. Qscored: A large dataset of code smells and quality metrics. In *2021 IEEE/ACM 18th international conference on mining software repositories (MSR)*, pages 590–594. IEEE.

André Silva, Sen Fang, and Martin Monperrus. 2023. Repairllama: Efficient representations and fine-tuned adapters for program repair. *arXiv preprint arXiv:2312.15698*.

Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. 2012. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156.

Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 48(3):930–950.

Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 87–98.