Identifying Kotlin Classes in Need of Refactoring: A Comparison of Machine Learning and Rule-Based Approaches

Arda Gökalp Batmaz

***Abstract*—Ensuring code quality in Android applications developed with Kotlin is increasingly vital, yet static analysis tools often rely on fixed thresholds that do not adapt well to varied projects. In this paper, I compare two distinct methodologies, a rule-based approach and a set of machine learning (ML) classifiers, to detect classes in need of refactoring. I collect and analyze key metrics such as cyclomatic complexity, cognitive complexity, and function count from two real-world Kotlin projects: "Let's Describe," a 69-class application with over 9k lines of code, and "CampusApp," a smaller, 38-class social platform. While the rule-based strategy achieves high recall, it also produces numerous false positives by over-flagging non-problematic classes.**

To overcome these shortcomings, I separately evaluate five ML algorithms; Decision Tree, Random Forest, Support Vector Machine, K-Nearest Neighbors, and Naive Bayes using cross-validation on the "Let's Describe" dataset and validation on "CampusApp." Random Forest emerges as the most robust model, consistently demonstrating high recall for problematic classes alongside satisfactory precision. Although K-Nearest Neighbors excels in "CampusApp," it fails to detect several code smells in "Let's Describe," underscoring the role of dataset size and distribution. Future work will include fine-tuning hyperparameters and validating the models on larger, multi-module, and open-source projects to enhance the generalizability of these findings.

Keywords — Detecting Code smells, Kotlin, Android Development, Refactoring, Machine Learning, Rule Based

I. INTRODUCTION

Refactoring has long been recognized as a crucial practice for sustaining software quality, maintainability, and evolution. As applications grow in size and complexity, managing the code and assuring its quality becomes an important topic in order to decrease technical debt [1]. When these considerations are left unmanaged, some software components become "code smells" that lead to significant development overhead, brittle code, and reduced productivity [2]. What's more, according to the CISQ, poor-quality software costs the US economy an estimated $2.8 trillion annually [3]. The way to reduce these costs and ensure the technical quality of the project comes from identifying and refactoring the problematic software components. Effective refactoring not only simplifies code structure but also reduces the cost of future changes, aiding in longer-term stability and scalability.

Over the years, a variety of static analysis tools have helped developers spot "code smells" indicative of refactoring needs. However, such tools often rely on rigid heuristics or threshold checks that may not capture the special interactions among different metrics. This limitation opens an area for machine learning (ML) approaches, which can learn from existing data to identify patterns that strongly correlate with refactoring demands. These ML-based approaches allow developers to identify code smells at both the class and function levels, similar to the rule-based systems in their projects [4].

In the process of detection of such classes, choosing the appropriate software quality metrics and suitable thresholds that can be generalized to work on different kinds of software projects is one of the challenges in this field. There is no all-in-one solution when it comes to identifying code smells. Different technical factors, such as used programming languages, frameworks, and libraries also affect the process of identifying appropriate metrics for similar projects. In addition, the results obtained with the collected metrics must match with the industry standards for the relevant software development field in order to make sure that the built model with these metrics benefits the real-life projects. Before working on building such models, the industrial software projects must be reviewed, the inner dynamics of similar projects must be considered, and feedback must be collected from the developers who are working on that project.

In this study, I propose a multifaceted approach that incorporates both machine learning (ML) techniques and rule-based systems to effectively identify and address code smells at the class level. The methodology is specifically tailored for Android projects written in Kotlin, a domain where software quality assessment is critical but underexplored in the context of code smell detection. While there are lots of different metrics used to evaluate code quality in general-purpose software systems, their implementation to mobile development leads to unique challenges. Android projects often involve distinct characteristics such as lifecycle management, UI complexities, and platform-specific constraints, which demand a more nuanced approach to code smell detection. Despite the growing importance of mobile applications, most existing research in code smell detection focuses on traditional software systems, leaving a gap in methodologies designed specifically for mobile environments. By addressing this gap, the proposed approach aims to create a tailored framework for identifying and mitigating code smells

in Kotlin-based Android projects, thereby improving maintainability and reducing technical debt.

II. RELATED WORK

Traditional approaches to code smell detection often rely on static analysis tools and rule-based systems. Tools such as PMD and iPlasma evaluate predefined thresholds for metrics like cyclomatic complexity, lines of code (LOC), and coupling between objects to identify potential issues. However, the subjectivity inherent in these thresholds limits their effectiveness across diverse projects. Studies, such as those by Fontana, have highlighted the importance of complementing heuristic-based methods with machine learning (ML) approaches to enhance detection accuracy and generalizability [5].

The use of ML in code smell detection has gained traction due to its ability to learn from existing datasets and adapt to different contexts. Di Nucci explored ML techniques for detecting code smells like God Class, Data Class, and Feature Envy. Their findings revealed that while ML classifiers such as Random Forest and J48 achieved high accuracy in controlled datasets, their performance often deteriorated in diverse and realistic scenarios, emphasizing the need for robust datasets and fine-tuned models [6].

Similarly, Fontana validated the utility of ML by benchmarking 16 classifiers across multiple projects, demonstrating that supervised learning approaches could achieve over 95% accuracy for specific code smells. However, they also noted the challenge of balancing performance with the interpretability of results, which is critical for developer adoption [5].

The transition from Java to Kotlin in Android development introduces unique challenges and opportunities in code smell detection. Unlike Java, Kotlin emphasizes concise syntax, null safety, and functional programming constructs. Therefore it is necessary to adapt the approaches used on traditional detection methods to align with Kotlin’s unique features. Novendra and Sunindyo addressed this gap by introducing a Kotlin-specific tool for detecting bad smells such as Brain Class and Brain Method using Program Structure Interface (PSI) representation. Their work highlighted the distinct characteristics of Kotlin code and the need for tailored metrics and thresholds [7].

What's more, software quality metrics such as Weighted Methods per Class (WMC), Tight Class Cohesion (TCC), and Access to Foreign Data (ATFD) are widely used to evaluate code quality. However, applying these metrics in mobile development presents unique challenges due to the distinct lifecycle, UI complexity, and performance constraints of Android applications. Recent studies emphasize the importance of considering platform-specific characteristics, such as resource limitations and interaction patterns, in metric selection and threshold determination to ensure effective code is written [8].

While substantial progress has been made in applying machine learning to detect code smells, the focus has largely been on general-purpose systems rather than mobile-specific environments. Moreover, Kotlin is the most popular choice for development in the Android ecosystem. Therefore, in this study I tried to find the metrics and models that are appropriate for finding the classes that require refactoring.

III. MATERIAL AND METHOD

A. Dataset Collection

The dataset used in this study was obtained from real-world Android projects written in Kotlin by EduSyntech Solutions company. Specifically, the main project that is worked on to create models is a mobile app already released on Google Play, and it's called Let's Describe. It's a special education mobile app that allows the children who have learning disorders to learn the objects in their environment by taking photos. In that way these children are learning to vocalize and how to assign the voices with the objects. Let's Describe contains more than 9k lines of code along with 69 classes, which makes it appropriate to work on rule-based and ML-based approaches. The second project is called "CampusApp." It is a small social media app that allows students from the same university to communicate with each other. It contains 38 classes and is used in order to validate the approaches derived from the Let's Describe project.

To effectively implement different code smell detection approaches, a diverse range of software quality metrics was required to analyze the projects comprehensively. These metrics provide critical insights into the technical structure of the Kotlin-based Android projects, enabling the identification of potential code smells and areas that might benefit from refactoring. To collect these metrics seamlessly and ensure compatibility with Kotlin, two widely used static analysis tools were employed: Detekt and SonarQube. Detekt is a tool specifically designed for Kotlin that allows identifying the code smells specific to Kotlin's constructs [9]. SonarQube is a multilingual static analysis and code quality management platform that allows detailed technical analysis on software projects [10].

Before the metric collection process with these tools, the metrics that can be candidates for detection processes have been chosen while also considering what kinds of metrics can be collected with these tools. There were two things to consider while doing that: the metrics effectiveness on

mobile-based projects and the appropriateness of these metrics on Kotlin. When the overall codebases of both projects are reviewed, it is concluded that these are single-module projects that are written in MVVM architecture. Therefore, there are UIs, view models, use cases, and repositories in the source code. By using the analysis tools, I collected the following metrics: line count, which indicates the size of a class and potential complexity; function count, reflecting the responsibilities and functionality of the class; statements, measuring the density of logic within the class; number of public functions, which reveals the exposure and potential coupling of a class; type, categorizing the class (enum class, data class, sealed class) to reflect its intended role; private fields, which contribute to encapsulation and internal cohesion; public fields, indicating external accessibility and potential loss of encapsulation; cyclomatic complexity, quantifying the number of independent paths and decision-making logic; cognitive complexity, measuring the difficulty of understanding the code structure; and coupling between objects, which evaluates interdependence and modularity. These metrics are collected properly from both projects. In the process of collection, I didn't collect the interfaces; instead, I set their "number of public functions" parameter as the same number of functions in their interface. Also, I didn't include the metrics of the UI design part because the metrics and technical points that must be focused on in this part are different from other parts of the project. The dataset used in this study, along with the scripts and configuration files are publicly available for usage on GitHub at https://github.com/Arda-Gokalp-Batmaz-AGB/Refactoring_Study_About_Kotlin_Based_Android_Apps. It includes detailed metric outputs, labeling data, and any preprocessing steps applied.

B. Labeling Process

After the data collection, I consulted one of the team members in our company who has experience on native Android development projects for two years and who participated in the development of Let's Describe. I asked him to indicate whether the relevant classes require refactoring or not for each class just by looking at the source code of the Let's Describe project. A class requires refactoring if it has the following properties: the class is a god class, there are functions that are too complex, class code is unreadable, there are unnecessary control statements, the class does too much work, there are necessary functionalities, etc. While also considering these rules, the developer made empirical guesses about whether a class requires refactoring or not. He labeled the class as 1 if it requires refactoring and set it as 0 if it does not require refactoring. At the end of the process, a table with two columns that contains the class name and an indicator that shows whether it is required to be refactored or not is obtained (Figure 1). I also discussed his reasoning about labeling.

Class	# Needs Refactoring
PotentialAnswerGenerator	1
SpeakerModel	1
TTSModel	0
CompletionOpenAIOptions	0
OpenAIImageLabelingPrompt	0
OpenAIServiceImpl	0
OpenAITranslatePromptEnum	0
TokenEnum	0
TranslateRequest	0

Figure 1. A Sample From the Labeling Process

Out of 69 classes in the Let's Describe project, he labeled 12 classes as needing refactoring and labeled 57 of them as not needing refactoring. That shows %17 total classes that need refactoring in the project. When we consider that the ratio of classes that have code smells according to the maturity of the project changes from 4.5% to 50% it is on a reasonable range to find patterns [11].

C. Rule-Based Approach

Before working on machine learning approaches, I properly examined the values of each metric and tried to come up with a rule-based approach that can detect smells by predefined rules. In the dataset, I recognized that most of the classes that have high cognitive complexities tend to be more likely to be the classes that need refactoring because cognitive complexity affects the readability of the code. I set a threshold of 10 for cognitive complexity and observed that 83.33% of the classes marked as "need refactoring" had a cognitive complexity above this threshold. In contrast, only 8.77% of the classes marked as "do not need refactoring" exceeded this threshold. Therefore, it was a good starting point for detecting problematic classes. I tested thresholds ranging from 8 to 12 and found that 10 was the most optimal value. Additionally, another metric strongly associated with classes needing refactoring is cyclomatic complexity. When setting the threshold to 5, I observed that 100% of the classes requiring refactoring had a cyclomatic complexity above this value, while only 21.06% of the unproblematic classes exceeded this threshold. I tried numbers up to 11 because after 11 the results were not very useful. When I set it as 10, I recognized that 83% of classes needing refactoring have cyclomatic complexity higher than 10, and 10% of other classes have cyclomatic complexity higher than 10. I applied the same procedure to the line count metric and identified 65 lines of code as the most appropriate threshold. All classes requiring refactoring had more than 65 lines of code, while only 17.5%

of the classes not requiring refactoring exceeded this threshold. The found thresholds for all metrics when they are treated as individual rules for detecting code smells are shown in (Figure 2). The threshold setting process prioritized correctly identifying the majority of problematic classes while minimizing the misclassification of non-problematic classes.

Metric	Optimal Threshold
Cognitive Complexity	10
Cyclomatic Complexity	10
Line of Code	65
Statements	24
Function Count	3
Number of Public Functions	2
Private Fields	0
Public Fields	1
Coupling Betwen Objects	3

Figure 2. Metrics and Thresholds When Metrics are Considered Individually

During the process of determining optimal thresholds, I observed that certain metrics were not effective in reliably indicating whether a class is a code smell or not. After filtering out these less impactful metrics, the following metrics were selected for combined usage (Figure 3).

Metric	Optimal Threshold
Cognitive Complexity	10
Cyclomatic Complexity	10
Line of Code	65
Statements	24
Function Count	3

Figure 3. Metrics and Thresholds After Unnecessary Ones are Trimmed

When these metrics were combined with their optimal thresholds, the resulting classification report (Figure 4) showed that the method successfully identified all code smells. However, it also exhibited a high rate of misclassifying normal classes as code smells. This issue leads to a significant number of false positives, which could hinder its practicality in real-world projects. Consequently, this approach requires further adjustments to improve its precision and reduce false positives for better usability.

```
=== Rule-Based Classification Report ===
              precision    recall  f1-score   support

           0       1.00      0.75      0.86        57
           1       0.46      1.00      0.63        12

    accuracy                           0.80        69
   macro avg       0.73      0.88      0.75        69
weighted avg       0.91      0.80      0.82        69
```

Figure 4. Results of Classification with All Metrics

I made some changes on the threshold of "statements"; however, I observed that change on this metric does not affect the results that much, and I removed the metric from the rules and recognized that there is a minor improvement in results. (Figure 5)

```
=== Rule-Based Classification Report ===
              precision    recall  f1-score   support

           0       1.00      0.77      0.87        57
           1       0.48      1.00      0.65        12

    accuracy                           0.81        69
   macro avg       0.74      0.89      0.76        69
weighted avg       0.91      0.81      0.83        69
```

Figure 5. Results of Rule-based Classification Without Statement Metric

Similarly, I set different thresholds for the line of code metric, but again nothing much changed; therefore, I removed line count as well, and the results are still identical with the previous reports (Figure 6).

```
=== Rule-Based Classification Report ===
              precision    recall  f1-score   support

           0       0.98      0.81      0.88        57
           1       0.50      0.92      0.65        12

    accuracy                           0.83        69
   macro avg       0.74      0.86      0.77        69
weighted avg       0.90      0.83      0.84        69
```

Figure 6. Results of Rule-based Classification Without Statement and LOC Metrics

There are some adjustments made on thresholds despite the fact that I wasn't able to find rule combinations that give better results. Therefore, while also considering the metrics that I especially focused on, cyclomatic complexity, cognitive complexity, and function count, I began to work on ML-based strategies.

D. ML-Based Approach

In the machine learning-based approach, firstly I thought about what kind of algorithms or models can be beneficial. Also, I considered the case that training data is limited; therefore, it is not very logical to use LLM or neural network-based approaches. I wanted to correctly classify classes as to whether they require refactoring or not. Therefore, I used different supervised classification algorithms to find the most appropriate one. The used algorithms are decision tree, random forest, SVM, KNN, and Naive Bayes.

Before beginning to use algorithms, I wanted to make sure that I was working with correct metrics; therefore, I performed feature selection with RandomForestClassifier, logistic regression, and KBest algorithms. Firstly, I observed that private/public field count does not represent the classes correctly because some classes, such as data classes, are just holding data, so tend to be not a code smell, and using public fields as parameters can mislead the models. Based on the empirical analysis and feature selection techniques, the four metrics identified as most influential in decision-making are cognitive complexity, function count, line count, and cyclomatic complexity. However, from the rule-based approach, I know that in that scenario, line count is not very effective on prediction; also, when I tested and validated each model, I recognized that the line count parameter is decreasing the F1 score for the "class needs refactoring" case drastically. After the feature selection process, I obtained the same metrics I found from the rule-based approach, which are cyclomatic complexity, cognitive complexity, and function count. In the training process, I used the Let's Describe project, which has 69 classes, and 12 of them need refactoring. While training, I used cross-validation to surpass the problem of having a small dataset. I especially examined the recall values of each model.

Following this process, I asked my other team member, who has one year of backend development experience to review the "CampusApp" project and label the classes they believe require refactoring. Out of 38 classes, they labeled 2 classes as requiring refactoring, indicating that 5.2% of the classes have issues. This ratio aligns with the typical proportion observed in production projects. So I used this project to validate and test my approaches. After this testing, I found the best options that can be used on similar projects.

IV. RESULTS

Firstly, I evaluated the performance of a rule-based approach using the following thresholds: cognitive complexity set at 10, cyclomatic complexity at 10, and function count at 3. On the "Let's Describe" project, the classification report showed that this method correctly identified 92% of the classes labeled as "1" (requiring refactoring), which is a strong result. However, 50% of the classes labeled as "1" were actually "0" (not requiring refactoring). This indicates that while the model is effective at detecting problematic classes, its high false-positive rate could increase developer workload, making it less practical for real-world applications (Figure 7). Also, I tested the model on the "CampusApp" project that consists of 38 classes, and 2 of them are labeled as needing refactoring. The model again underperformed; this time the misclassifying rate for label 1 is 17%, so I found out that in the current situation it is not logical to work on a rule-based model, and I tried ML-based approaches (Figure 8).

```
=== Rule-Based Classification Report ===
              precision    recall  f1-score   support

           0       0.98      0.81      0.88        57
           1       0.50      0.92      0.65        12

    accuracy                           0.83        69
   macro avg       0.74      0.86      0.77        69
weighted avg       0.90      0.83      0.84        69
```

Figure 7. Let's Describe Project Classification Report for Rule-Based Approach

```
=== Rule-Based Classification Report for CampusApp ===
              precision    recall  f1-score   support

           0       1.00      0.72      0.84        36
           1       0.17      1.00      0.29         2

    accuracy                           0.74        38
   macro avg       0.58      0.86      0.56        38
weighted avg       0.96      0.74      0.81        38
```

Figure 8. CampusApp Project Classification Report for Rule-Based Approach

In the ML-based approach, after performing feature selection and hyperparameter tuning, I trained the models using cross-validation on the "Let's Describe" project's metric data and tested their performance on the "CampusApp" metrics. In comparing the algorithms, I prioritized recall and precision over accuracy, as identifying all code smell classes (high recall) is crucial to ensure no problematic classes are overlooked. At the same time, I acknowledged the trade-off

with precision, where some false positives are acceptable to achieve comprehensive detection of problematic classes.

I started with tree-based algorithms. The random forest and decision tree are similar and some of the most common classifier algorithms therefore before other algorithms I tested them. The decision tree algorithm didn't perform well on distinguishing the labels on the Let's Describe project (Figure 9). Despite the training performance of the decision tree is not good, it successfully found two code smell classes from the CampusApp project however again there is a very bad precision for label 1 and that leads to many more false positives (Figure 10).

Classification Report for Decision Tree on Let's Describe :

	precision	recall	f1-score	support
0	0.78	0.78	0.78	9
1	0.60	0.60	0.60	5
accuracy			0.71	14
macro avg	0.69	0.69	0.69	14
weighted avg	0.71	0.71	0.71	14

Figure 9. Let's Describe Project Classification Report for Decision Tree Approach

Classification Report for Decision Tree on CampusApp:

	precision	recall	f1-score	support
0	1.00	0.92	0.96	36
1	0.40	1.00	0.57	2
accuracy			0.92	38
macro avg	0.70	0.96	0.76	38
weighted avg	0.97	0.92	0.94	38

Figure 10. CampusApp Project Classification Report for Decision Tree Approach

However, for the random forest case, after I properly found the best parameters and trained on the Let's Describe project, the results were better compared to the decision tree. It correctly finds all classes that are labeled as "1," and the classes that are identified as "1" have a 0.71 success rate, which is a huge improvement on recall values. Also, it has 86% accuracy (Figure 11). It also works better on CampusApp's data with higher precision on label "1" (Figure 12)

Accuracy : 0.86
Classification Report for Random Forest on Let's Describe :

	precision	recall	f1-score	support
0	1.00	0.78	0.88	9
1	0.71	1.00	0.83	5
accuracy			0.86	14
macro avg	0.86	0.89	0.85	14
weighted avg	0.90	0.86	0.86	14

Figure 11. Let's Describe Project Classification Report for Random Forest Approach

Accuracy : 0.9736842105263158
Classification Report for Random Forest on CampusApp:

	precision	recall	f1-score	support
0	1.00	0.97	0.99	36
1	0.67	1.00	0.80	2
accuracy			0.97	38
macro avg	0.83	0.99	0.89	38
weighted avg	0.98	0.97	0.98	38

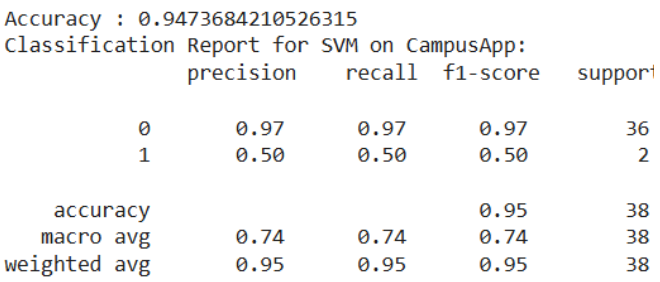
Figure 12. CampusApp Project Classification Report for Random Forest Approach

After applying hyperparameter tuning with a focus on improving "recall," SVM was trained using the optimal parameters. The results on the training dataset demonstrated a well-balanced performance compared to other algorithms (Figure 13). However, when tested on the "CampusApp" project, SVM underperformed relative to the other approaches (Figure 14), indicating potential overfitting or sensitivity to variations in project-specific data.

Accuracy : 0.86
Classification Report for SVM on Let's Describe:

	precision	recall	f1-score	support
0	0.89	0.89	0.89	9
1	0.80	0.80	0.80	5
accuracy			0.86	14
macro avg	0.84	0.84	0.84	14
weighted avg	0.86	0.86	0.86	14

Figure 13. Let's Describe Project Classification Report for SVM Approach



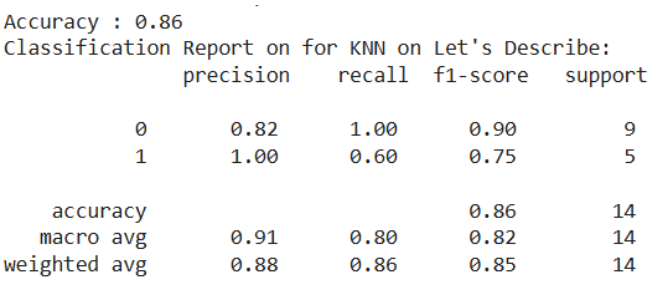
```
Accuracy : 0.9473684210526315
Classification Report for SVM on CampusApp:
              precision    recall  f1-score   support

           0       0.97      0.97      0.97        36
           1       0.50      0.50      0.50         2

    accuracy                           0.95        38
   macro avg       0.74      0.74      0.74        38
weighted avg       0.95      0.95      0.95        38
```

Figure 14. CampusApp Project Classification Report for SVM Approach

Using the KNN algorithm, I observed that it performed well in avoiding misclassifying label "0" classes as "1" on the training set. However, it struggled to identify all classes labeled as "1" in the dataset, leading to missed detections (Figure 15). Interestingly, during testing on the "CampusApp" project, KNN achieved perfect classification, correctly labeling all classes without any misclassifications (Figure 16).

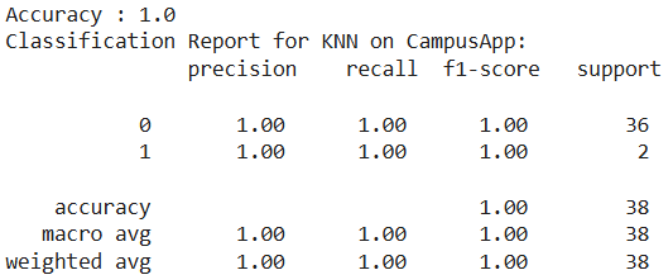


```
Accuracy : 0.86
Classification Report on for KNN on Let's Describe:
              precision    recall  f1-score   support

           0       0.82      1.00      0.90         9
           1       1.00      0.60      0.75         5

    accuracy                           0.86        14
   macro avg       0.91      0.80      0.82        14
weighted avg       0.88      0.86      0.85        14
```

Figure 15. Let's Describe Project Classification Report for KNN Approach



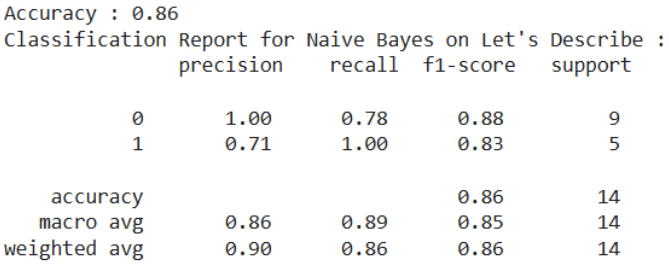
```
Accuracy : 1.0
Classification Report for KNN on CampusApp:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        36
           1       1.00      1.00      1.00         2

    accuracy                           1.00        38
   macro avg       1.00      1.00      1.00        38
weighted avg       1.00      1.00      1.00        38
```

Figure 16. CampusApp Project Classification Report for KNN Approach

Naive bayes performed identical with the random forest on training data (Figure 17). However it misclassified more for label 1 compared to the other algorithms on CampusApp project (Figure 18).

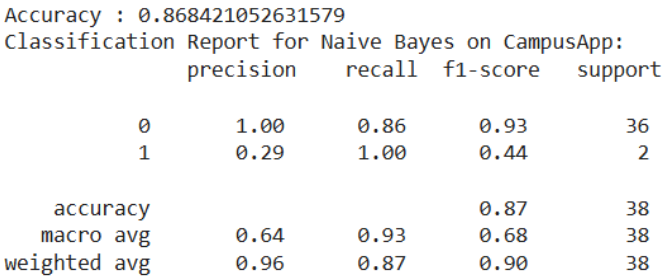


```
Accuracy : 0.86
Classification Report for Naive Bayes on Let's Describe :
              precision    recall  f1-score   support

           0       1.00      0.78      0.88         9
           1       0.71      1.00      0.83         5

    accuracy                           0.86        14
   macro avg       0.86      0.89      0.85        14
weighted avg       0.90      0.86      0.86        14
```

Figure 17. Let's Describe Project Classification Report for Naive Bayes Approach



```
Accuracy : 0.868421052631579
Classification Report for Naive Bayes on CampusApp:
              precision    recall  f1-score   support

           0       1.00      0.86      0.93        36
           1       0.29      1.00      0.44         2

    accuracy                           0.87        38
   macro avg       0.64      0.93      0.68        38
weighted avg       0.96      0.87      0.90        38
```

Figure 18. CampusApp Project Classification Report for Naive Bayes Approach

After training and testing each model, I observed that the precision and recall values for label "0" were consistently within a reasonable range, with at least 70% across the experiments. However, for label "1," these values exhibited significant variation between models and datasets. To better analyze and compare the results, I organized the precision and recall values for label "1" into separate tables for the "Let's Describe" and "CampusApp" projects (Figures 19-20). This detailed breakdown highlights the differences in model performance, particularly in identifying problematic classes.

	Rule-Based	**Decision Tree**	**Randomforest**	**SVM**	**KNN**	**Naive Bayes**
Precision	0.50	0.6	0.71	0.8	1	0.71
Recall	0.92	0.6	1.00	0.8	0.6	1.00
f1-score	0.65	0.6	0.83	0.8	0.75	0.83

Figure 19. Results for Label 1 in Let's Describe

	Rule-Based	Decision Tree	Randomforest	SVM	KNN	Naive Bayes
Precision	0.17	0.4	0.67	0.5	1	0.29
Recall	1.00	1	1.00	0.5	1	1.00
f1-score	0.29	0.57	0.80	0.5	1	0.44

Figure 20. Results for Label 1 in CampusApp

V. DISCUSSION AND CONCLUSION

In this study, in order to find the classes that require refactoring on Kotlin-based Android projects I collected metrics from the Let's Describe project and a smaller project called CampusApp. I used a rule-based approach along with the different AI algorithms to find the best fitting solution for real life projects. The main problem with these approaches was misclassifying the most of non problematic classes as problematic and sometimes missing the problematic classes. Among all the implemented approaches, I found out that KNN and Random forest algorithms outperforms other approaches in terms of performance on both Let's Describe project's metrics and CampusApp's metrics. However, despite KNN performing best on the CampusApp's metrics, it wasn't able to identify class labels with "1" correctly on Let's Describe data while it was training. The reason behind that can be KNN algorithm have much more bias to the label "0" compared to the other algorithms therefore in the CampusApp data set which contains code smell classes about 5% it performed very well but for Let's Describe which it's 17% of classes have code smell it does not performs as much as that due to the bias on label "0". So numbers can be misleading. In the random forest side, despite it having problems on misclassifying some of the label "0" classes as label "1" when we also consider its training performance on Let's Describe it is a much more reliable choice compared to the other approaches. Therefore random forest is the best option out of other options.

In conclusion, while Random Forest emerged as the most effective option for detecting code smells in Kotlin-based Android projects within this study, the limited dataset and the small number of metrics used pose challenges for generalizing its applicability to medium- to large-scale, multi-modular projects. Additionally, the analyzed projects are taken from the same company therefore they share similar coding practices, which could introduce bias in the results. To improve the robustness and generalizability of the approach, future work should focus on testing it on more diverse, open-source projects with higher complexity and varied coding styles. This would provide deeper insights and pave the way for developing more comprehensive and scalable solutions.

REFERENCES

[1] A. Crudu, “The Importance of Code Refactoring in Software Maintenance,” Moldstud.com, Mar. 18, 2024. https://moldstud.com/articles/p-the-importance-of-code-refactoring-in-software-maintenance.

[2] “Opsera | What is Code Smell & How to Avoid it,” Opsera.io,2024. https://www.opsera.io/blog/what-is-code-smell.

[3] G. Andersen, “The Benefits of Code Refactoring and Optimization in Software Projects,” Moldstud.com, Jan. 27,2024. https://moldstud.com/articles/p-the-benefits-of-code-refactoring-and-optimization-in-software-projects.

[4] F. Pecorelli, F. Palomba, D. D. Nucci, and A. D. Lucia, “Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection,” May 2019, doi: https://doi.org/10.1109/icpc.2019.00023.

[5] F. A. 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, Jun. 2015, doi: https://doi.org/10.1007/s10664-015-9378-4.

[6] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, “Detecting code smells using machine learning techniques: Are we there yet?,” 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 612–621, Mar. 2018, doi: https://doi.org/10.1109/saner.2018.8330266.

[7] Radinal Dwiki Novendra and Wikan Danar Sunindyo, “Emerging Trends in Code Quality: Introducing Kotlin-Specific Bad Smell Detection Tool for Android Apps,” IEEE Access, vol. 12, pp. 63895–63903, Jan. 2024, doi: https://doi.org/10.1109/access.2024.3397055.

[8] Radinal Dwiki Novendra and Wikan Danar Sunindyo, “Emerging Trends in Code Quality: Introducing Kotlin-Specific Bad Smell Detection Tool for Android Apps,” IEEE Access, vol. 12, pp. 63895–63903, Jan. 2024, doi: https://doi.org/10.1109/access.2024.3397055.

[9] “Hello from detekt | detekt,” Detekt.dev, 2024. https://detekt.dev/

[10] “Clean Code Tools for Writing Clear, Readable & Understandable Secure Quality Code,” Sonarsource.com, 2024. https://www.sonarsource.com/

[11] Mahmoud Alfadel, Khalid Aljasser, and M. Alshayeb, “Empirical study of the relationship between design patterns and code smells,” PLoS ONE, vol. 15, no. 4, pp. e0231731–e0231731, Apr. 2020, doi: https://doi.org/10.1371/journal.pone.0231731.