Detecting and resolving feature envy through automated machine learning and move method refactoring

Dimah Al-Fraihat[1], Yousef Sharrab[2], Abdel-Rahman Al-Ghuwairi[3], Majed AlElaimat[3], Maram Alzaidi[4]
[1]Department of Software Engineering, Faculty of Information Technology, Isra University, Amman, Jordan
[2]Department of Data Science and Artificial Intelligence, Faculty of Information Technology, Isra University, Amman, Jordan
[3]Department of Software Engineering, Faculty of Prince Al-Hussein Bin Abdallah II for Information Technology, The Hashemite University, Zarqa, Jordan
[4]Department of Computer Science, College of Computers and Information Technology, Taif University, Taif, Kingdom of Saudi Arabia

Article Info

Article history:

Received Oct 9, 2023
Revised Dec 30, 2023
Accepted Jan 9, 2024

Keywords:

Automated machine learning
Code smell
Feature envy
Move method
Refactoring
Software quality

ABSTRACT

Efficiently identifying and resolving code smells enhances software project quality. This paper presents a novel solution, utilizing automated machine learning (AutoML) techniques, to detect code smells and apply move method refactoring. By evaluating code metrics before and after refactoring, we assessed its impact on coupling, complexity, and cohesion. Key contributions of this research include a unique dataset for code smell classification and the development of models using AutoGluon for optimal performance. Furthermore, the study identifies the top 20 influential features in classifying feature envy, a well-known code smell, stemming from excessive reliance on external classes. We also explored how move method refactoring addresses feature envy, revealing reduced coupling and complexity, and improved cohesion, ultimately enhancing code quality. In summary, this research offers an empirical, data-driven approach, integrating AutoML and move method refactoring to optimize software project quality. Insights gained shed light on the benefits of refactoring on code quality and the significance of specific features in detecting feature envy. Future research can expand to explore additional refactoring techniques and a broader range of code metrics, advancing software engineering practices and standards.

Corresponding Author:

Dimah Al-Fraihat
Department of Software Engineering, Faculty of Information Technology, Isra University
Amman, 11622, Jordan
Email: d.fraihat@iu.edu.jo

1. INTRODUCTION

Refactoring is the process of enhancing the readability and usability of software code while preserving its functionality [1]. Changes must be made to the code without changing how it functions. The main goals of refactoring are to make the code more understandable and to make it easier to update-either the design or the actual code-without requiring as much work [2]. Refactoring is advantageous for several reasons. Firstly, it improves code understandability which makes it simpler to maintain and lowers the likelihood of introducing errors [3]. Furthermore, refactoring promotes code reuse by enabling its implementation in different projects or different sections of a program [4]. Refactoring further makes code simpler, which makes it easier to manage as well as more flexible in the long run. In addition to reducing technical debt, this helps to avoid problems and inefficiencies [5]. Another benefit is that by improving the readability and understanding of the codebase, refactoring fosters collaboration among team members [6]. Ultimately, through refactoring, we can enhance program quality, maintainability, and performance, which leads to improved efficiency when working with programs and projects [7].

Software refactoring includes a variety of activities aimed at improving the quality, maintainability, and performance of the software [8]. Through following a structured method to refactoring, developers can ensure that the changes made to the code are safe, effective, and well-documented [9]. Software refactoring encompasses several activities, including identifying “code smells” which involves recognizing areas of code that could be refactored and enhanced or simplified, such as duplicated code, long methods, or complex conditional statements [1], [10]. Twenty-two types of code smells were identified by Beck *et al.* [11], among which is the widely recognized feature envy.

Feature envy is a widespread code smell that occurs when a method in a class uses a disproportionate number of features or data that belongs to another class, instead of using its own data [12]. In other words, the method in one class seems to "envy" the features or data of another class, leading to tight coupling and dependency between the two classes, potentially complicating the maintenance, comprehension, and modification of the code [13]. Feature envy results in close coupling between classes, which may have an impact on the maintainability and readability of the code [14]. To address this issue, refactoring using the move method is proposed as “the process of moving the method that envy features to the class that owns them” [15]. This refactoring can reduce coupling between classes and improve code readability and quality [16]. To apply the move method for refactoring, it is necessary to identify the methods that cause feature envy at first, and after that determine which class owns the features or data being accessed. The methods can then be moved to that class, making any necessary updates to method calls or references in other classes.

Addressing feature envy bad smell though the move method refactoring is an effective approach that can be implemented to improve the quality of software code [17], [18]. The literature proposes various automatic refactoring methods to aid software specialists in identifying and rectifying problematic code through recommended refactoring operations [19], [20]. These methods can be rule-based, data-driven using machine learning, or software-based to optimize predetermined metrics [21]. Every approach has its unique advantages and limitations. Rule-based methods tend to produce satisfactory outcomes, but creating rules for code smells can be challenging as it is a manual and tedious task [22]. Additionally, search-based heuristics endeavor to detect code smells based on predetermined metrics, necessitating the manual establishment of threshold values that the algorithm can use to determine whether a smell has been identified. Automating the detection of refactoring opportunities using machine learning can offer several advantages, including faster and more accurate identification of problematic code, reducing the workload on developers, and improving the overall quality of the codebase. Nevertheless, research indicates that it is important to ensure that the machine learning model is well-trained and tested to avoid introducing new problems or missing important refactoring opportunities. Furthermore, advanced approaches to detecting code smells based on machine learning require further investigation to assess their effectiveness [18], [23].

In this study, we present a machine learning based methodology designed for the identification of feature envy, a prevalent concern in code. Our proposed approach utilizes AutoML techniques to discern instances of code smells within the codebase and subsequently employs the move method refactoring technique. To gauge the impact on software quality, we conducted a comparative analysis of evaluation metrics, assessing their values both pre- and post-refactoring.

The subsequent sections of this paper are structured as follows: in section 2, background information is provided, along with a comprehensive overview of related works in the field. Section 3 outlines the methods and the steps followed in the experiment. Section 4 presents the results and the discussion. Finally, in section 5, the paper concludes by summarizing the key findings, discussing their implications, and exploring potential avenues for future research.

2. BACKGROUND AND RELATED LITERATURE

Code smells are warning signs of potential challenges with software design or code quality, highlighting areas that need more research rather than severe prohibitions. One distinguishing code smell is feature envy, which occurs when a method expresses a stronger interest in the properties of other classes than its own [24]. Refactoring is a technique that emphasizes enhancing the structure of code without modifying its functionality. To prevent adding any problems, it is imperative to use caution when refactoring [25]. The move method strategy, which involves moving a method to a class, is a practical method for resolving the feature envy issue. This decreases dependencies while simultaneously improving the flexibility and maintainability of the code [26]. JDeodorant, a software tool that developers use for detecting and fixing code smells like feature envy, employs algorithms to analyze the code, provide refactoring suggestions, and assist in managing technical debt effectively [27].

There is an approach to recognize chances of refactoring using the move method technique, which involves applying relational topic models (RTM). RTM considers both the structure and text of source code, enabling an understanding of how different methods are interconnected. By treating source code methods as "documents" and analyzing their dependencies, RTM extracts information such as method calls, identifiers, and comments to uncover topics and their relationships. This approach has shown promising results in pinpointing move method possibilities, surpassing other techniques in terms of selecting the most suitable methods to relocate based on topic similarity [28]. The adoption of RTM holds potential for advancing software engineering research and enhancing software quality by harnessing its capability to extract textual insights from source code.

Undoubtedly, identifying code smells and implementing refactoring techniques like the move method are essential steps in enhancing software design, maintainability, and overall code quality. Tools like JDeodorant and approaches like RTM contribute significantly to detecting and resolving code smells, resulting in higher quality code bases. By addressing code smells, developers can improve the readability, modifiability, and performance of the software, ultimately leading to more efficient and maintainable systems. Such practices are crucial in the continuous improvement and evolution of software projects.

Researchers have been working on automatically detecting feature envy for the past few decades, with the main goal of finding and recommending classes where methods have been mistakenly inserted. This study proposes the move method refactoring technique as a solution, concentrating on applying AutoML techniques for feature envy detection. Relocating recognized methods from one class to another includes ensuring they are in line with the data or behavior they most frequently interact with. This makes it simpler to maintain and promotes the code to be of higher quality. This section provides an overview of current methods that may be roughly divided into two categories: those that use machine learning techniques to identify feature envy and those that make use of the move method refactoring technique to improve the quality and maintainability of software code. Researchers are integrating these approaches in an effort to identify and resolve feature envy-related problems in software code, aiming to create robust and easily manageable software systems.

There has been a significant amount of research that focused on understanding how code smells affect the quality of software. In the study conducted by Kaur [28], the researchers analyzed and evaluated existing literature on this topic. Their findings suggest that code smells do not have a uniform impact on software quality. Different code smells can have varying effects on various aspects. This literature review underscores the significance of exploring known code smells, considering less commonly discussed quality attributes, collaborating with industry researchers, and analyzing large-scale commercial software systems. These efforts aim to gain deeper insights and improve software development practices.

The study conducted by Reis *et al.* [29] aimed to investigate the identification and visualization of code smells. It had two goals: first, to examine the techniques and tools discussed in previous research for detecting code smells, and second, to analyze the utilization of visual methods in supporting this detection process. To conduct the research, over eighty primary studies were collected from repositories, and a careful selection process was applied to choose the most relevant works. The findings revealed that the approaches used for detecting code smells include search-based, metric-based, and symptom-based techniques. Notably, a significant majority of these studies (83.1%) rely on open-source software for their analyses.

According to a study by Rahman *et al.* [26], they suggest using the move method refactoring approach as a way to improve software design. This approach specifically targets the issue of feature envy, which occurs when methods are placed in the wrong class. The researchers combined factors such as coupling, cohesion, and contextual similarity to provide effective recommendations. They evaluated this approach on seven open-source projects and found that it performed better than the widely used JDeodorant tool in terms of precision, recall, and F measure. Additionally, they found that the accuracy of the approach was influenced by project standards and sizes, highlighting its benefits for software development and maintenance.

The research of [30] aims to explore the current understanding of coupling smells among practitioners. The study identifies defining factors of coupling smells, their impacts, relationships with other smells, and fix options as perceived by practitioners. The results highlight gaps between scientific theory and practice in the detection and management of coupling smells. The article presents five lessons that serve as opportunities and challenges for future research, facilitating a better understanding of practitioner concerns for both scientists and practitioners dealing with coupling smells in software development.

Based on a study highlighted by AbuHassan *et al.* [31], the researchers found 145 studies that examine smell detection in software design and code. They analyzed these studies to answer questions regarding the existing techniques for detecting smells, such as the level of abstraction (design or code), types of smells targeted, metrics used, implementation details, and validation methods. They identified categories of smell detection techniques. Interestingly, they discovered that 57% of the studies did not include any performance measures, 41% omitted information about the programming language being targeted, and 14% of these studies did not validate their detection techniques. When it comes to the level of abstraction, 18% of the studies focused on detecting smells at the design level. This lack of coverage highlights the importance of placing emphasis on identifying smells during the design phase to address them early on.

The research undertaken by Alfadel *et al.* [32] explores the connection between design patterns and code smells in software systems. By analyzing ten Java-based systems using analysis and association rules, the findings reveal that classes that utilize design patterns tend to have occurrences and frequencies of code smells compared to those that do not. Some specific design patterns may coincide with code smells, such as command patterns being associated with God class, Blob, and external duplication smells.

In another study, Al-Obeidallah *et al.* [14] empirically investigated how the adapter design pattern impacts software maintainability. They refactored four subject systems to create versions with both pattern implementation and non-pattern implementation, and compared software metrics between them. The analysis relied on correlations to software maintainability from research. The empirical results indicate that the adapter pattern versions exhibit software metrics, such as [mention specific metrics], suggesting an influence on software maintainability.

The study by Rizwan *et al.* [33] explores the importance of software module coupling in various design aspects, including fault prediction, impact analysis, re-assessment, and software vulnerabilities assessment. The research conducted an examination of coupling metrics and their coverage of significant factors related to coupling. The analysis revealed that although many metrics consider levels of coupling, they often fail to distinguish between these levels. Moreover, most metrics overlook the breadth, hiddenness, and rigidity of data flow, and none of the metrics consider the combined impact of these aspects.

An interesting study conducted by Singh and Kaur [4] introduces an approach to predict code smells using machine learning techniques and software metrics. This approach aims to enhance software quality, improve maintainability, and minimize the risk of faults. The study's findings indicate that tree-based algorithms, specifically random forest, outperform Kernel-based and network-based algorithms in this context. Additionally, the accuracy of these machine learning algorithms is further improved by incorporating algorithm-based feature selection and parameter optimization techniques. To understand the predictions made by the machine learning model, local interpretable model-agnostic explanations are employed. Overall, this research underscores the potential of machine learning techniques in predicting code smells and highlights their valuable role in enhancing software quality.

Building on previous related work, numerous studies have explored code smells, software quality, and design patterns. They have identified positive impacts of design patterns on software maintainability and the co-occurrence of code smells with certain design patterns. Despite these insights, there are still gaps in understanding coupling smells and the early detection of bad smells during the design phase [34]. Recognizing the potential of machine learning techniques in predicting code smells and enhancing software quality [35], our next section presents our methodology to further investigate the impacts of code smells on software quality and explore effective refactoring approaches.

3. METHOD

This study aims to detect feature envy code smells in software code through the utilization of machine learning techniques, subsequently addressing them through Move method refactoring. The methodology employed to achieve the research objectives encompasses the selection of the dataset and corpus retrieval, followed by the choice of a code metrics tool and the generation of measurement metrics. Subsequently, the integration of code metrics and the "Bartosz Walter 2018 842778" dataset is carried out. To enhance classifier performance, imbalanced classes are addressed using SMOTETomek sampling. The steps followed to fulfil the research objectives are detailed as follows:

Step 1: Dataset selection and corpus retrieval

In this study, the selected dataset is "Bartosz Walter 2018 842778" sourced from the "qualitas corpus (QC)". The dataset comprises various types of code smells, class names, and information about the detection tools [36]. Specifically, this dataset focuses on classes that have been identified as having code smells, with the detection process performed using three distinct tools. To denote the characteristics of the dataset, the filenames incorporate essential information. These filenames consist of the base release of the QC and a numerical value (25, 50, or 75). This numerical value signifies the minimum number of detectors (tools) that recognized a specific instance of a code smell within a class. For example, if a code smell, designated as X, is detected by only one out of the three tools, it will be recorded in the file marked with the number 25, but not in files with numbers 50 or 75. Furthermore, the dataset employs a percentage-based representation to indicate the detection level of code smells within classes. If all four tools identify a code smell in a class, the value will be 100%. Similarly, if two out of the four tools detect the code smell, the value will be 50%. Notably, it is essential to mention that the dataset itself lacks metrics for detecting the feature envy code smell. Therefore, the next step involved extracting the necessary metrics from the QC, a comprehensive compilation of open-source Java systems, to supplement the dataset for further analysis and investigation.

Step 2: Selection of code metrics tool and producing measurement metrics

Metrics related to code were collected using the “Understand” tool, developed by SciTools. This tool is known for its functionalities, such as dependency analysis, visualization of call graphs, testing of code standards, and computation of metrics. It is widely acknowledged as a valuable integrated development environment (IDE) and plays a significant role in extracting metrics from the QC dataset. The application of the “understand” tool enables an evaluation of the codebase, facilitating the extraction of code metrics necessary for subsequent analysis. Utilizing the capabilities of the “Understand” tool ensures the effective measurement of code-related attributes within the QC dataset. Additionally, the research process is enhanced with a widely recognized tool that excels in analyzing code. Furthermore, this approach aligns with standards to achieve strong and reliable findings when studying code smells and associated metrics within our dataset. It also ensures the accuracy and robustness of the experiments, allowing for more reliable and valid conclusions regarding code smells and their metrics [20], [37], [38].

Step 3: Integration of code metrics and "Bartosz Walter 2018 842778" dataset

To ensure a comprehensive analysis, the "Bartosz Walter 2018 842778" dataset with the code metrics we collected in step 2 were combined. This merging process involved matching the datasets based on the system ID and class full name. The resulting dataset consists of a total of 16,543 rows, with each column labeled to represent issues comprising 44 unique features. The primary objective of incorporating these metrics into the existing dataset was to simplify the prediction of instances related to feature envy bade smell. The combination of code metrics with the dataset enables a thorough analysis of issues in the code. By including these 44 features derived from code metrics, we gained insights into the structure of the code and potential occurrences of feature envy. Hence, enhancing our comprehensive understanding of code smells and their impact on software quality. The dataset we obtained, comprising code smells and their corresponding code metrics, provides a basis for our analysis of feature envy. This unified dataset is a valuable resource for conducting further statistical analysis and modeling to identify patterns and trends in detecting and addressing feature envy.

Step 4: Application of the SMOTETomek sampling technique to mitigating class imbalance and enhancing machine learning model performance

Analyzing the data reveals a discrepancy in the distribution of classes within our dataset as depicted in Figure 1. To address this issue and verify proper model training, the researchers decided to employ a technique known as "SMOTETomek." This method can effectively address the problem of imbalanced classes and improve the accuracy of our machine learning model.

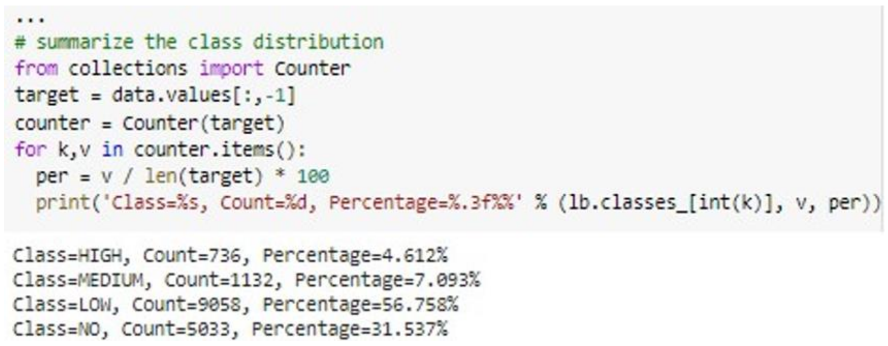
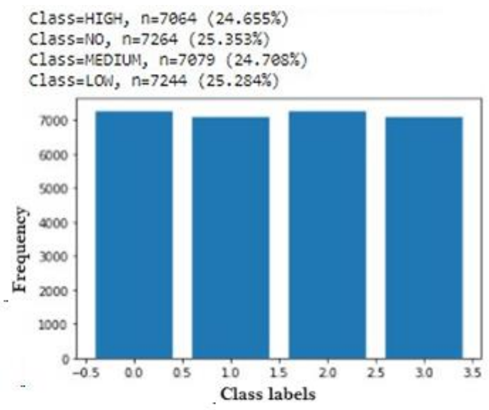

```
...
# summarize the class distribution
from collections import Counter
target = data.values[:,-1]
counter = Counter(target)
for k,v in counter.items():
  per = v / len(target) * 100
  print('Class=%s, Count=%d, Percentage=%.3f%%' % (lb.classes_[int(k)], v, per))
```
```
Class=HIGH, Count=736, Percentage=4.612%
Class=MEDIUM, Count=1132, Percentage=7.093%
Class=LOW, Count=9058, Percentage=56.758%
Class=NO, Count=5033, Percentage=31.537%
```

Figure 1. The unbalanced distribution of classes

The SMOTETomek consists of two techniques: synthetic minority oversampling technique (SMOTE) and Tomek links. SMOTE generates samples to increase the representation of the minority class in the dataset while Tomek links assist in this process. SMOTE creates instances that capture the defining characteristics of the minority class with the goal of addressing class imbalance and creating a more balanced training dataset. Concurrently, Tomek links are employed to identify pairs of instances from different classes that are in close proximity to each other. These pairs are potential sources of noise or misclassification in the data. To improve class separation and refine decision boundaries, instances from the majority class that form Tomek links with instances from the minority class are removed. This process enhances the overall balance of the dataset and makes the decision boundaries between classes more discernible. By integrating the strengths of over-sampling through SMOTE and under-sampling via Tomek links, the SMOTETomek approach proficiently addresses the issue of class imbalance. As a result, the training data becomes more representative of the underlying distribution, leading to an enhanced model performance and more reliable predictions on unseen data.

The application of the SMOTETomek sampling technique aligns with improving the accuracy and generalization capability of the machine learning model. By rectifying the class imbalance, the model is trained on a more equitable and diverse dataset, which is crucial for achieving robust and unbiased performance in real-world scenarios. The balanced classes resulting from this step are depicted in Figure 2, illustrating the effectiveness of the SMOTETomek technique in achieving a more balanced representation of classes in the dataset.



```
Class=HIGH, n=7064 (24.655%)
Class=NO, n=7264 (25.353%)
Class=MEDIUM, n=7079 (24.708%)
Class=LOW, n=7244 (25.284%)
```

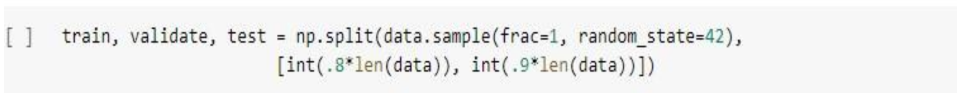
Figure 2. The distribution of balanced classes

Step 5: Model selection

To tackle the classification challenge in this study, involving the categorization of the dataset into four distinct classes ("No," "Low," "Medium," and "High") based on the number of tools detecting bad smells, the researchers opted to utilize the AutoGluon framework. AutoGluon is an open-source AutoML tool specifically designed for training highly accurate machine learning models on raw tabular datasets, such as CSV files, using Python code. The particular module of AutoGluon employed in this study, AutoGluon-Tabular, was chosen for its ensemble techniques and model stacking capabilities, providing advantages in training time efficiency compared to traditional approaches that focus solely on individual model and hyperparameter selection. The decision to adopt AutoGluon-Tabular was reinforced through empirical experiments, wherein it consistently outperformed the best combinations of its competitor tools. This noteworthy finding underscores the efficacy of AutoGluon-Tabular as the preferred choice for addressing the classification problem in the context of this study. By utilizing AutoGluon-Tabular, the researchers aim to enhance the accuracy and efficiency of the classification process, thereby obtaining reliable and high-quality results in predicting the severity of code smells based on the number of tools detecting them. The selection of AutoGluon-Tabular aligns with the research objective of employing cutting-edge methodologies to tackle the classification challenge posed by the dataset's unique characteristics.

Step 6: model training

The dataset was divided into three parts for this study: the training set, the validation set, and the testing set. The training set was used for model training, the validation set for hyperparameter tuning and model selection, and the testing set for evaluating the model's performance on unseen data. Figure 3 illustrates the data splitting code. This division ensured the model's evaluation on independent data, mitigating overfitting and providing a more realistic assessment of its capabilities.



```
[ ]   train, validate, test = np.split(data.sample(frac=1, random_state=42),
                              [int(.8*len(data)), int(.9*len(data))])
```

Figure 3. Snippet of the code for splitting data into three parts (train, validate, and test)

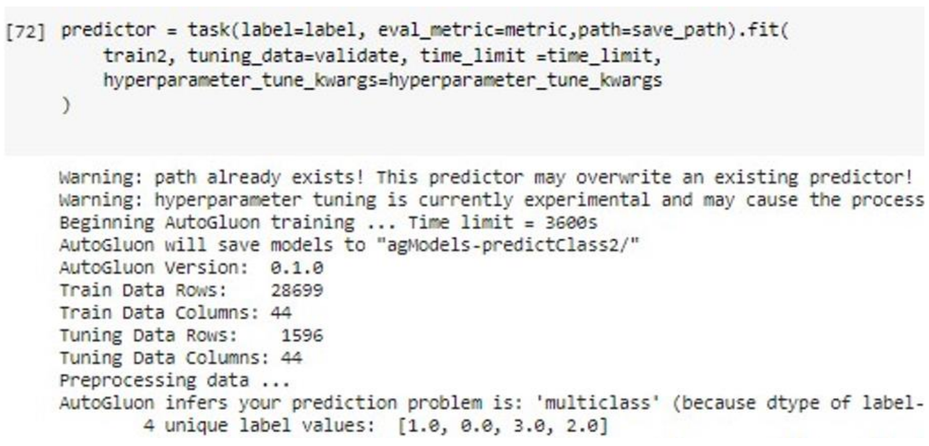
The models were trained using the AutoGluon-TabularPredictor module. AutoGluon automatically identified the task as a multiclass classification problem based on the dataset's characteristics and the number of classes. The objective of this multiclass classification task was to classify the existence of feature envy bad smells. The "TabularPredictor" package from AutoGluon efficiently employs various algorithms and ensemble methods to enhance model performance for multiclass classification problems. Through utilizing multiple machine learning models, the model learns from the data and generates accurate predictions for all classes, ensuring robustness in handling the complexities in multiclass classification.

Throughout the training process, several settings were fine-tuned to optimize the models' performance as depicted in Figure 4. These configurations were carefully selected to achieve the best possible model outcomes and obtain reliable and accurate predictions for the multiclass classification of feature envy instances. These settings encompass the following aspects:

a. The objective function was configured to optimize the model's performance based on specific metrics such as precision, recall, and F1 score. By prioritizing these metrics, the model is geared towards achieving the best results for the given classification problem.
b. AutoGluon employs automated hyperparameter tuning, seeking the most suitable combination of hyperparameters for the models. This process effectively fine-tunes hyperparameters, such as learning rate, to optimize the model's performance.
c. AutoGluon offers a diverse array of machine learning models to choose from. Through automated model selection, the library automatically identifies and utilizes the best-performing models on the training data. This comprehensive model exploration facilitates a thorough examination of various model structures and methodologies, ensuring an informed and effective selection process.

Based on the aforementioned settings, the authors aimed to identify the top-performing model for the designated classification task through utilizing the AutoGluon-TabularPredictor library. This strategy capitalizes on automation while maintaining flexibility in adjusting and refining all models, ultimately enhancing their accuracy and predictive capabilities.

The validation portion of the dataset was specifically employed for hyperparameter tuning and model selection. Hyperparameters are adjustable parameters that significantly impact the model's performance. By evaluating the model's performance on the validation set with varying hyperparameter settings, it was possible to identify the most optimal combination of hyperparameters. Conversely, the testing portion of the dataset was exclusively used to evaluate the model's performance on unseen data. This step offered an unbiased assessment of the model's generalization ability to handle novel instances beyond the training data. As a result, this evaluation provided valuable insights into the model's effectiveness and robustness when applied to real-world scenarios.



```
[72] predictor = task(label=label, eval_metric=metric,path=save_path).fit(
         train2, tuning_data=validate, time_limit =time_limit,
         hyperparameter_tune_kwargs=hyperparameter_tune_kwargs
     )

     Warning: path already exists! This predictor may overwrite an existing predictor!
     Warning: hyperparameter tuning is currently experimental and may cause the process
     Beginning AutoGluon training ... Time limit = 3600s
     AutoGluon will save models to "agModels-predictClass2/"
     AutoGluon Version:  0.1.0
     Train Data Rows:    28699
     Train Data Columns: 44
     Tuning Data Rows:    1596
     Tuning Data Columns: 44
     Preprocessing data ...
     AutoGluon infers your prediction problem is: 'multiclass' (because dtype of label-
             4 unique label values:  [1.0, 0.0, 3.0, 2.0]
```

Figure 4. Configuration settings for AutoGluon-TabularPredictor

4. RESULTS AND DISCUSSION

The training process encompasses 23 models, classified into 9 types as depicted in Figure 5, which are as follows: “RFModel,” “KNNModel,” “NNFastAiTabularModel,” “TabularNeuralNetModel,” “CatBoostModel,” “LGBModel,” “XGBoostModel,” “WeightedEnsembleModel,” and “XTModel.” Figure 6 presents the top-performing models derived from the training process which involved the 23 models. These models were evaluated based on their respective performance metrics.

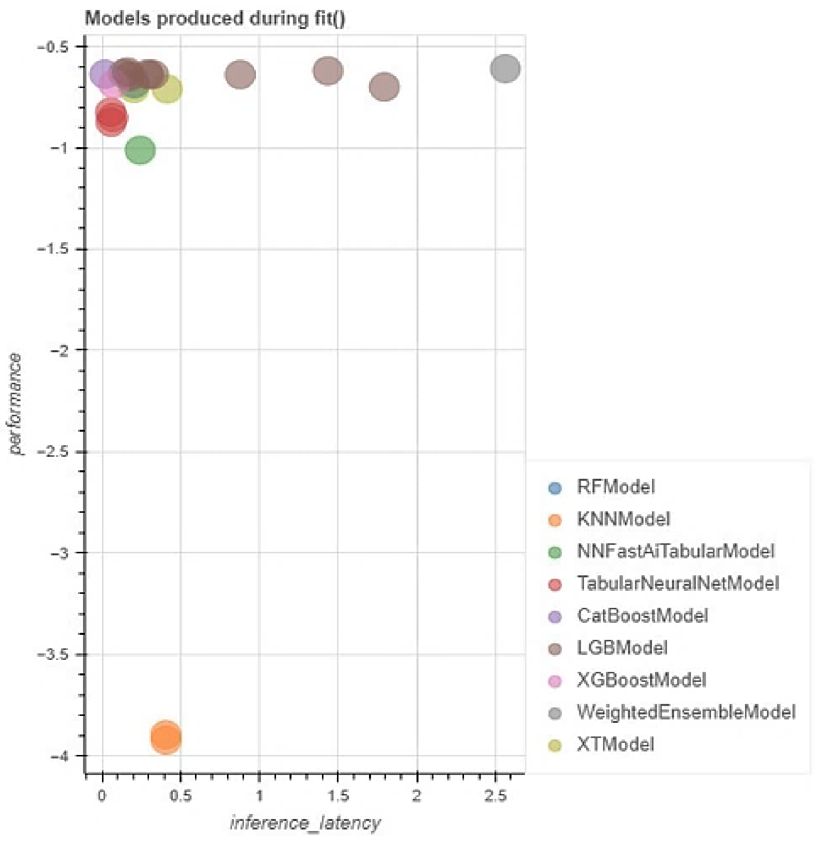
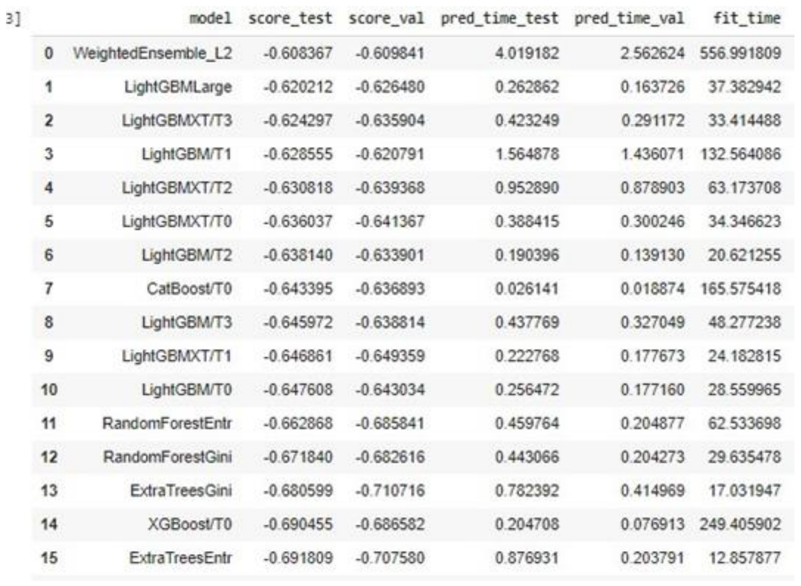


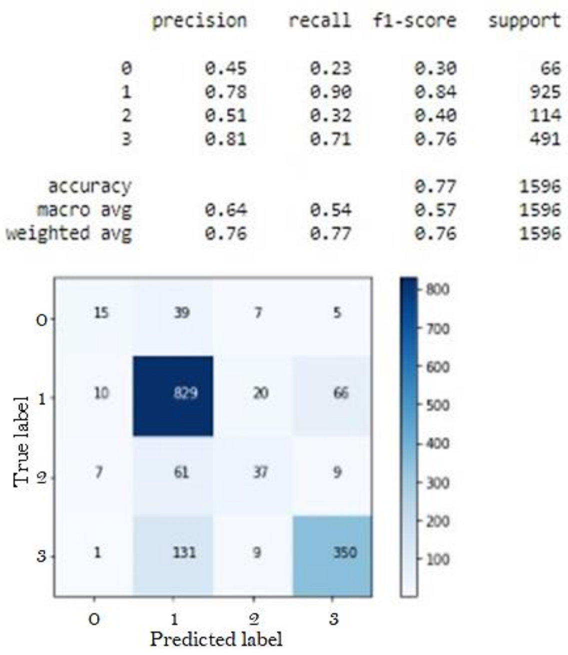
Figure 5. Types of trained models



3]	model	score_test	score_val	pred_time_test	pred_time_val	fit_time
0	WeightedEnsemble_L2	-0.608367	-0.609841	4.019182	2.562624	556.991809
1	LightGBMLarge	-0.620212	-0.626480	0.262862	0.163726	37.382942
2	LightGBMXT/T3	-0.624297	-0.635904	0.423249	0.291172	33.414488
3	LightGBM/T1	-0.628555	-0.620791	1.564878	1.436071	132.564086
4	LightGBMXT/T2	-0.630818	-0.639368	0.952890	0.878903	63.173708
5	LightGBMXT/T0	-0.636037	-0.641367	0.388415	0.300246	34.346623
6	LightGBM/T2	-0.638140	-0.633901	0.190396	0.139130	20.621255
7	CatBoost/T0	-0.643395	-0.636893	0.026141	0.018874	165.575418
8	LightGBM/T3	-0.645972	-0.638814	0.437769	0.327049	48.277238
9	LightGBMXT/T1	-0.646861	-0.649359	0.222768	0.177673	24.182815
10	LightGBM/T0	-0.647608	-0.643034	0.256472	0.177160	28.559965
11	RandomForestEntr	-0.662868	-0.685841	0.459764	0.204877	62.533698
12	RandomForestGini	-0.671840	-0.682616	0.443066	0.204273	29.635478
13	ExtraTreesGini	-0.680599	-0.710716	0.782392	0.414969	17.031947
14	XGBoost/T0	-0.690455	-0.686582	0.204708	0.076913	249.405902
15	ExtraTreesEntr	-0.691809	-0.707580	0.876931	0.203791	12.857877

Figure 6. The top-performing models with strong predictive capabilities

Among the 23 models, the “WeightedEnsemble_L2” emerged as the most effective model in solving the classification problem, exhibiting an accuracy of 77% and a macro average F1-score of 57%. As demonstrated in Figure 7, the model demonstrated strong learning capabilities on the proposed dataset. The results of training the models revealed the 20 most important features that significantly impact the classification process, as indicated in Figure 8.

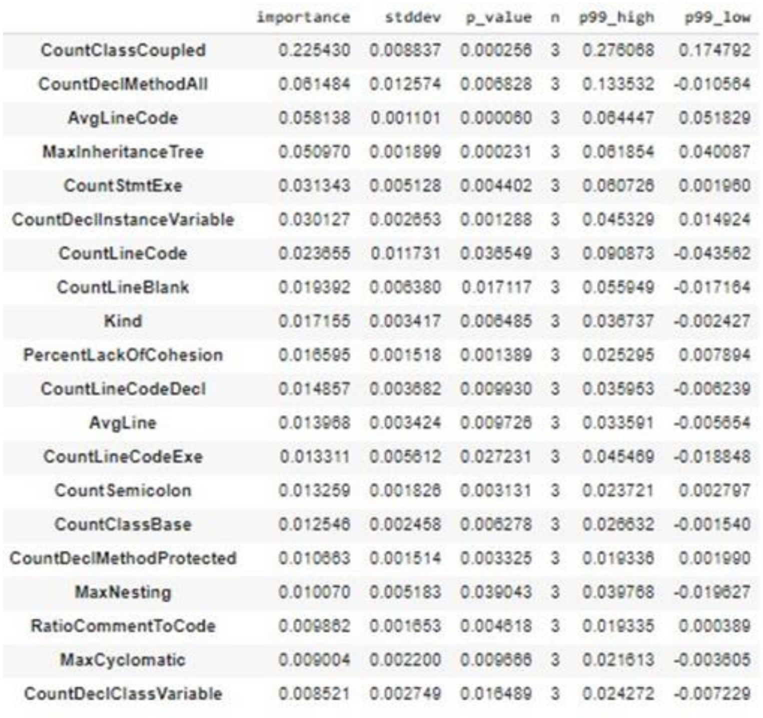


```
              precision    recall  f1-score   support

           0       0.45      0.23      0.30        66
           1       0.78      0.90      0.84       925
           2       0.51      0.32      0.40       114
           3       0.81      0.71      0.76       491

    accuracy                           0.77      1596
   macro avg       0.64      0.54      0.57      1596
weighted avg       0.76      0.77      0.76      1596
```

Figure 7. The confusion matrix for the top best model “WeightedEnsmble_L2”

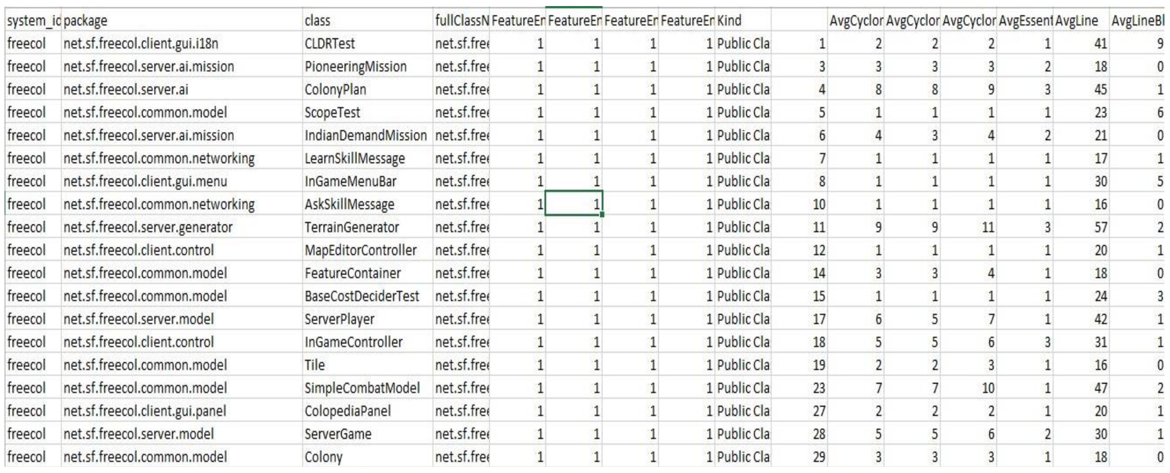


	importance	stddev	p_value	n	p99_high	p99_low
CountClassCoupled	0.225430	0.008837	0.000256	3	0.276068	0.174792
CountDeclMethodAll	0.061484	0.012574	0.006828	3	0.133532	-0.010564
AvgLineCode	0.058138	0.001101	0.000060	3	0.064447	0.051829
MaxInheritanceTree	0.050970	0.001899	0.000231	3	0.061854	0.040087
CountStmtExe	0.031343	0.005128	0.004402	3	0.060726	0.001960
CountDeclInstanceVariable	0.030127	0.002653	0.001288	3	0.045329	0.014924
CountLineCode	0.023655	0.011731	0.036549	3	0.090873	-0.043562
CountLineBlank	0.019392	0.006380	0.017117	3	0.055949	-0.017164
Kind	0.017155	0.003417	0.006485	3	0.036737	-0.002427
PercentLackOfCohesion	0.016595	0.001518	0.001389	3	0.025295	0.007894
CountLineCodeDecl	0.014857	0.003682	0.009930	3	0.035953	-0.006239
AvgLine	0.013968	0.003424	0.009726	3	0.033591	-0.005654
CountLineCodeExe	0.013311	0.005612	0.027231	3	0.045469	-0.018848
CountSemicolon	0.013259	0.001826	0.003131	3	0.023721	0.002797
CountClassBase	0.012546	0.002458	0.006278	3	0.026632	-0.001540
CountDeclMethodProtected	0.010663	0.001514	0.003325	3	0.019336	0.001990
MaxNesting	0.010070	0.005183	0.039043	3	0.039768	-0.019627
RatioCommentToCode	0.009862	0.001653	0.004618	3	0.019335	0.000389
MaxCyclomatic	0.009004	0.002200	0.009666	3	0.021613	-0.003605
CountDeclClassVariable	0.008521	0.002749	0.016489	3	0.024272	-0.007229

Figure 8. The 20 most influential features impacting the classification process

3.1. Feature envy code smell detection results

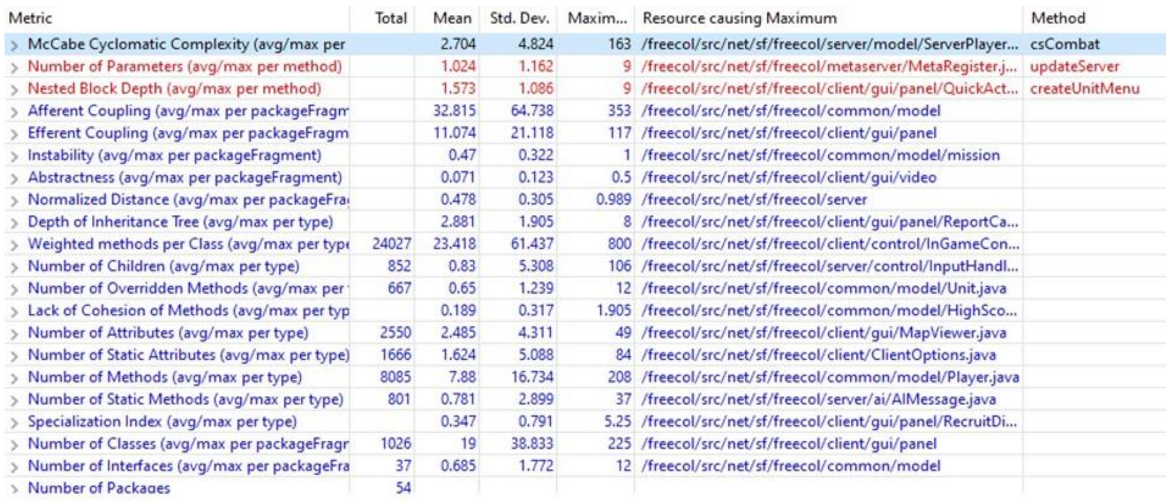
The best-performing model obtained from the model training process was employed to predict the level of feature envy code smell in a novel system named “FreeCol”. This system was sourced from the QC and its corresponding metrics were calculated using the “Understand” tool as shown in Figure 9. Harnessing the capabilities of the identified top model, the investigation focused on accurately detecting and quantifying instances of feature envy in the “FreeCol” system.



system_ic	package	class	fullClassN	FeatureEn	FeatureEn	FeatureEn	FeatureEn	Kind		AvgCyclor	AvgCyclor	AvgCyclor	AvgEssent	AvgLine	AvgLineBl
freecol	net.sf.freecol.client.gui.i18n	CLDRTest	net.sf.free	1	1	1	1	Public Cla	1	2	2	2	1	41	9
freecol	net.sf.freecol.server.ai.mission	PioneeringMission	net.sf.free	1	1	1	1	Public Cla	3	3	3	3	2	18	0
freecol	net.sf.freecol.server.ai	ColonyPlan	net.sf.free	1	1	1	1	Public Cla	4	8	8	9	3	45	1
freecol	net.sf.freecol.common.model	ScopeTest	net.sf.free	1	1	1	1	Public Cla	5	1	1	1	1	23	6
freecol	net.sf.freecol.server.ai.mission	IndianDemandMission	net.sf.free	1	1	1	1	Public Cla	6	4	3	4	2	21	0
freecol	net.sf.freecol.common.networking	LearnSkillMessage	net.sf.free	1	1	1	1	Public Cla	7	1	1	1	1	17	1
freecol	net.sf.freecol.client.gui.menu	InGameMenuBar	net.sf.free	1	1	1	1	Public Cla	8	1	1	1	1	30	5
freecol	net.sf.freecol.common.networking	AskSkillMessage	net.sf.free	1	1	1	1	Public Cla	10	1	1	1	1	16	0
freecol	net.sf.freecol.server.generator	TerrainGenerator	net.sf.free	1	1	1	1	Public Cla	11	9	9	11	3	57	2
freecol	net.sf.freecol.client.control	MapEditorController	net.sf.free	1	1	1	1	Public Cla	12	1	1	1	1	20	1
freecol	net.sf.freecol.common.model	FeatureContainer	net.sf.free	1	1	1	1	Public Cla	14	3	3	4	1	18	0
freecol	net.sf.freecol.common.model	BaseCostDeciderTest	net.sf.free	1	1	1	1	Public Cla	15	1	1	1	1	24	3
freecol	net.sf.freecol.server.model	ServerPlayer	net.sf.free	1	1	1	1	Public Cla	17	6	5	7	1	42	1
freecol	net.sf.freecol.client.control	InGameController	net.sf.free	1	1	1	1	Public Cla	18	5	5	6	3	31	1
freecol	net.sf.freecol.common.model	Tile	net.sf.free	1	1	1	1	Public Cla	19	2	2	3	1	16	0
freecol	net.sf.freecol.common.model	SimpleCombatModel	net.sf.free	1	1	1	1	Public Cla	23	7	7	10	1	47	2
freecol	net.sf.freecol.client.gui.panel	ColopediaPanel	net.sf.free	1	1	1	1	Public Cla	27	2	2	2	1	20	1
freecol	net.sf.freecol.server.model	ServerGame	net.sf.free	1	1	1	1	Public Cla	28	5	5	6	2	30	1
freecol	net.sf.freecol.common.model	Colony	net.sf.free	1	1	1	1	Public Cla	29	3	3	3	1	18	0

Figure 9. FreeCol system metrics using the understand tool

The code metrics for the "FreeCol" system were assessed prior to any code refactoring using two distinct tools, namely the "Metric" plugin and "CKJM." The "Metric" plugin serves as a dedicated tool providing metrics calculation and dependency analysis capabilities specifically designed for the Eclipse platform. Conversely, "CKJM" is a program that calculates Chidamber and Kemerer object-oriented metrics through the processing of compiled Java files' bytecode. The results of the code metric measurements for the FreeCol system before refactoring are presented in Figure 10.



Metric	Total	Mean	Std. Dev.	Maxim...	Resource causing Maximum	Method
McCabe Cyclomatic Complexity (avg/max per		2.704	4.824	163	/freecol/src/net/sf/freecol/server/model/ServerPlayer...	csCombat
Number of Parameters (avg/max per method)		1.024	1.162	9	/freecol/src/net/sf/freecol/metaserver/MetaRegister.j...	updateServer
Nested Block Depth (avg/max per method)		1.573	1.086	9	/freecol/src/net/sf/freecol/client/gui/panel/QuickAct...	createUnitMenu
Afferent Coupling (avg/max per packageFragm		32.815	64.738	353	/freecol/src/net/sf/freecol/common/model	
Efferent Coupling (avg/max per packageFragm		11.074	21.118	117	/freecol/src/net/sf/freecol/client/gui/panel	
Instability (avg/max per packageFragment)		0.47	0.322	1	/freecol/src/net/sf/freecol/common/model/mission	
Abstractness (avg/max per packageFragment)		0.071	0.123	0.5	/freecol/src/net/sf/freecol/client/gui/video	
Normalized Distance (avg/max per packageFra		0.478	0.305	0.989	/freecol/src/net/sf/freecol/server	
Depth of Inheritance Tree (avg/max per type)		2.881	1.905	8	/freecol/src/net/sf/freecol/client/gui/panel/ReportCa...	
Weighted methods per Class (avg/max per type	24027	23.418	61.437	800	/freecol/src/net/sf/freecol/client/control/InGameCon...	
Number of Children (avg/max per type)	852	0.83	5.308	106	/freecol/src/net/sf/freecol/server/control/InputHandl...	
Number of Overridden Methods (avg/max per	667	0.65	1.239	12	/freecol/src/net/sf/freecol/common/model/Unit.java	
Lack of Cohesion of Methods (avg/max per typ		0.189	0.317	1.905	/freecol/src/net/sf/freecol/common/model/HighSco...	
Number of Attributes (avg/max per type)	2550	2.485	4.311	49	/freecol/src/net/sf/freecol/client/gui/MapViewer.java	
Number of Static Attributes (avg/max per type)	1666	1.624	5.088	84	/freecol/src/net/sf/freecol/client/ClientOptions.java	
Number of Methods (avg/max per type)	8085	7.88	16.734	208	/freecol/src/net/sf/freecol/common/model/Player.java	
Number of Static Methods (avg/max per type)	801	0.781	2.899	37	/freecol/src/net/sf/freecol/server/ai/AIMessage.java	
Specialization Index (avg/max per type)		0.347	0.791	5.25	/freecol/src/net/sf/freecol/client/gui/panel/RecruitDi...	
Number of Classes (avg/max per packageFragr	1026	19	38.833	225	/freecol/src/net/sf/freecol/client/gui/panel	
Number of Interfaces (avg/max per packageFra	37	0.685	1.772	12	/freecol/src/net/sf/freecol/common/model	
Number of Packages	54					

Figure 10. Code metrics for the FreeCol system before refactoring

The metrics utilized in our study for the purpose of results comparison encompass the following:

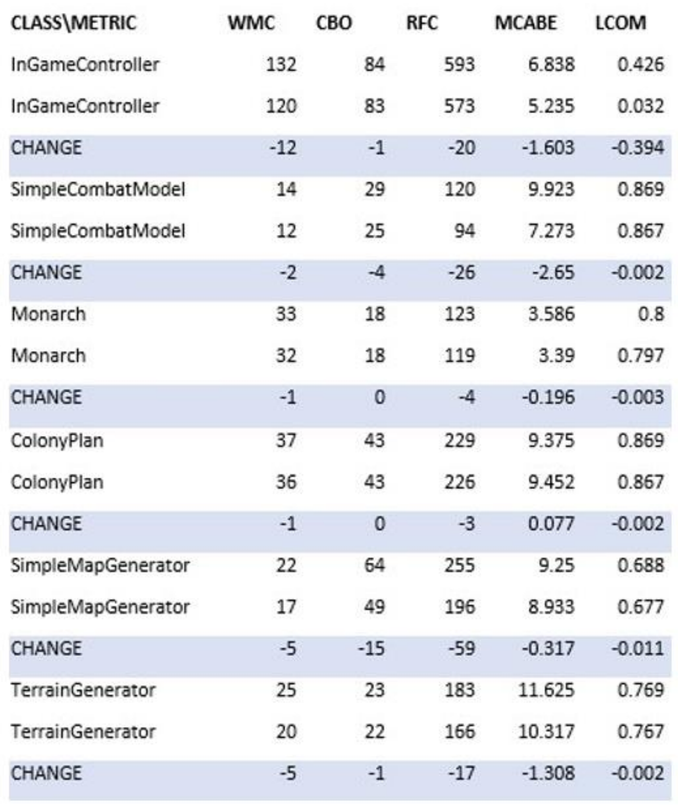
a. Weighted methods per class (WMC): WMC is an object-oriented metric introduced by Chidamber and Kemerer to measure complexity within a class. It quantifies the number of methods in a class, assigning weights to each method based on its significance in terms of complexity.
b. Coupling between object classes (CBO): CBO represents the number of classes coupled to a given class in the software system. While some degree of coupling is necessary for system functionality, excessive coupling can lead to difficulties in maintainability and reusability.
c. Response for a class (RFC): RFC measures the number of distinct methods and constructors invoked by a class. It quantifies the variety of methods executed when an object of that class receives a message (i.e., when a method is invoked for that object). A high RFC metric for a method can signify potential issues in terms of understandability, debugging, and testing of the class, thus affecting its maintainability.

d. Lack of cohesion in methods (LCOM): LCOM is a measure that indicates the number of not connected method pairs within a class, representing independent parts with no cohesion. It quantifies the difference between the number of method pairs not sharing instance variables and the number of method pairs with common instance variables.
e. McABE complexity: McABE complexity is a software metric used to gauge the complexity of a program. It quantifies the number of linearly independent paths through a program's source code, providing a quantitative assessment of its complexity. This metric was developed by Thomas J. McCabe, Sr. in 1976.

The incorporation of these metrics in our study facilitates a comprehensive evaluation of the software system's characteristics, aiding in the comparison of different systems and identifying potential areas for improvement in terms of complexity, coupling, and cohesion, thereby contributing to enhanced software quality and maintainability.

3.2. Application of the refactoring method results

A subset of classes predicted as “high” was subjected to refactoring using the move method technique, facilitated by the Eclipse IDE and assisted by the JDeodorant plugin. JDeodorant, as an Eclipse plugin, serves to detect design problems in Java software, aiding in the identification of opportunities for code improvement. Subsequently, the metrics were re-evaluated after the refactoring process, and the differences in metrics were meticulously calculated and analyzed, as represented in Figure 11. The first line in Figure 11 corresponds to the metrics before the refactoring, while the second line illustrates the metrics after the refactoring. The difference between the two sets of metrics is denoted as "CHANGE."



CLASS\METRIC	WMC	CBO	RFC	MCABE	LCOM
InGameController	132	84	593	6.838	0.426
InGameController	120	83	573	5.235	0.032
CHANGE	-12	-1	-20	-1.603	-0.394
SimpleCombatModel	14	29	120	9.923	0.869
SimpleCombatModel	12	25	94	7.273	0.867
CHANGE	-2	-4	-26	-2.65	-0.002
Monarch	33	18	123	3.586	0.8
Monarch	32	18	119	3.39	0.797
CHANGE	-1	0	-4	-0.196	-0.003
ColonyPlan	37	43	229	9.375	0.869
ColonyPlan	36	43	226	9.452	0.867
CHANGE	-1	0	-3	0.077	-0.002
SimpleMapGenerator	22	64	255	9.25	0.688
SimpleMapGenerator	17	49	196	8.933	0.677
CHANGE	-5	-15	-59	-0.317	-0.011
TerrainGenerator	25	23	183	11.625	0.769
TerrainGenerator	20	22	166	10.317	0.767
CHANGE	-5	-1	-17	-1.308	-0.002

Figure 11. Comparison of metrics before and after refactoring the feature envy code smell

The outcomes of the analysis demonstrated that the refactoring process, specifically employing the move method technique to address feature envy, yielded enhancements in code quality. Notably, the refactoring led to a reduction in coupling, an increase in cohesion, and a decrease in the overall complexity. These results underscore the efficacy of the applied refactoring method in optimizing the software design, resulting in improved code maintainability and overall software quality.

Following the completion of our experiment, a novel dataset comprising code metrics was generated to facilitate the classification of feature envy bad smells. Moreover, a comprehensive evaluation was conducted involving the training and testing of 23 models using AutoGluon to identify the optimal classification model. Among the 23 trained models, the "WeightedEnsemble_L2" model emerged as the most effective, exhibiting a substantial increase in accuracy from 58% to 77%. Additionally, the research outcomes highlighted the 20 most influential features significantly impacting the classification of feature envy instances. These features play a pivotal role in accurately distinguishing and classifying instances of the code smell, thereby contributing valuable insights into the underlying patterns of feature envy.

In the context of addressing the feature envy bad smell in the code, the move method refactoring technique was employed. Precise metrics were computed both before and after the refactoring process. The findings of this study demonstrate that the application of the move method refactoring has successfully yielded notable improvements in the quality of the code, particularly evidenced by a reduction in coupling and complexity, as well as an enhancement in cohesion. These results underscore the efficacy of the refactoring approach in enhancing software design, promoting better code maintainability, and elevating overall code quality.

5. CONCLUSION

This study introduced a new dataset that combines code metrics for the identification of feature envy issues using various tools. The dataset has been used in training machine learning models for classifying feature envy instances. By utilizing the AutoGluon, we found that the best model for our dataset is WeightedEnsmble_L2. This was made possible by leveraging its hyperparameter tuning capabilities, which significantly enhanced the performance of our model. Afterwards, we applied the move method refactoring technique to address instances of feature envy and assessed its impact on code quality metrics. The results indicated that refactoring feature envy through move method using AutoML has improved the code quality, reduced coupling, increased cohesion, and decreased complexity.

In this study, we aim to focus on key areas for future research efforts. First, we want to broaden our experimentation by incorporating software systems, different code metrics exploring various types of code issues and investigating different ways to improve the code. This wider approach will give us an understanding of code quality and how different strategies for improving it work. Additionally, we plan to enhance the quality and usefulness of our dataset by including other relevant metrics and code issues, as well as introducing new software systems. This will make our dataset more reliable and representative for conducting analyses. Moreover, we will explore other AutoML libraries like AutoKeras and H2O to train our models and compare their performance with AutoGluon. This comparison will help us understand the strengths and weaknesses of AutoML approaches. We will also use techniques to fine tune the models' parameters in order to improve accuracy on our dataset, aiming for the best possible configurations and improved predictive performance. Lastly, based on the suitable model, from our dataset we plan to develop a specialized tool that can automatically detect and fix any code issues that arise.

DECLARATIONS

Data Availability Statement: The data presented in this study are available and can be accessed at (https://github.com/MajedOlimat/MajedOlimat).

REFERENCES

[1] A. A. B. Baqais and M. Alshayeb, “Automatic software refactoring: a systematic literature review,” *Software Quality Journal*, vol. 28, no. 2, pp. 459–502, Jun. 2020, doi: 10.1007/s11219-019-09477-y.

[2] M. Fowler, *Refactoring: improving the design of existing code*, 2nd edition. Addison-Wesley Professional, 2018.

[3] F. L. Caram, B. R. D. O. Rodrigues, A. S. Campanelli, and F. S. Parreiras, “Machine learning techniques for code smells detection: a systematic mapping study,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 29, no. 02, pp. 285–316, Feb. 2019, doi: 10.1142/S021819401950013X.

[4] S. Singh and S. Kaur, “A systematic literature review: refactoring for disclosing code smells in object oriented software,” *Ain Shams Engineering Journal*, vol. 9, no. 4, pp. 2129–2151, Dec. 2018, doi: 10.1016/j.asej.2017.03.002.

[5] M. Agnihotri and A. Chug, “A systematic literature survey of software metrics, code smells, and refactoring techniques,” *Journal of Information Processing Systems*, vol. 16, no. 4, pp. 915–934, 2020.

[6] B. Walter, F. A. Fontana, and V. Ferme, “Code smells and their collocations: A large-scale experiment on open-source systems,” *Journal of Systems and Software*, vol. 144, pp. 1–21, Oct. 2018, doi: 10.1016/j.jss.2018.05.057.

[7] H. Cervantes and R. Kazman, “Software Archinaut: a tool to understand architecture, identify technical debt hotspots and manage evolution,” in *Proceedings of the 3rd International Conference on Technical Debt*, Jun. 2020, pp. 115–119, doi: 10.1145/3387906.3388633.

[8] F. A. Fontana, E. Mariani, A. Mornioli, R. Sormani, and A. Tonello, “An experience report on using code smells detection tools,” in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, Mar. 2011, pp. 450–457, doi: 10.1109/ICSTW.2011.12.

[9] E. Tempero *et al.*, “The qualitas corpus: a curated collection of Java code for empirical studies,” in *2010 Asia Pacific Software Engineering Conference*, Nov. 2010, pp. 336–345, doi: 10.1109/APSEC.2010.46.

[10] F. A. Fontana and S. Spinelli, "Impact of refactoring on quality code evaluation," in *Proceedings of the 4th Workshop on Refactoring Tools*, May 2011, pp. 37–40, doi: 10.1145/1984732.1984741.

[11] K. Beck, M. Fowler, and G. Beck, "Bad smells in code," *Refactoring: Improving the design of existing code*, vol. 1, pp. 75–88, 1999.

[12] M. Alzahrani, "Measuring class cohesion based on client similarities between method pairs: An improved approach that supports refactoring," *IEEE Access*, vol. 8, pp. 227901–227914, 2020, doi: 10.1109/ACCESS.2020.3046109.

[13] L. Sonnleithner, R. Rabiser, and A. Zoitl, "Bad smells in industrial automation: sniffing out feature envy," in *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug. 2022, pp. 346–349, doi: 10.1109/SEAA56994.2022.00061.

[14] M. G. Al-Obeidallah, D. G. Al-Fraihat, A. M. Khasawneh, A. M. Saleh, and H. Addous, "Empirical investigation of the impact of the adapter design pattern on software maintainability," in *2021 International Conference on Information Technology (ICIT)*, Jul. 2021, pp. 206–211, doi: 10.1109/ICIT52682.2021.9491719.

[15] F. Khan, S. Kanwal, S. Alamri, and B. Mumtaz, "Hyper-parameter optimization of classifiers, using an artificial immune network and its application to software bug prediction," *IEEE Access*, vol. 8, pp. 20954–20964, 2020, doi: 10.1109/ACCESS.2020.2968362.

[16] D.-L. Miholca, G. Czibula, and V. Tomescu, "COMET: a conceptual coupling based metrics suite for software defect prediction," *Procedia Computer Science*, vol. 176, pp. 31–40, 2020, doi: 10.1016/j.procs.2020.08.004.

[17] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code smells and refactoring: a tertiary systematic review of challenges and observations," *Journal of Systems and Software*, vol. 167, Sep. 2020, doi: 10.1016/j.jss.2020.110610.

[18] Z. Kurbatova, I. Veselov, Y. Golubev, and T. Bryksin, "Recommendation of move method refactoring using path-based representation of code," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, Jun. 2020, pp. 315–322, doi: 10.1145/3387940.3392191.

[19] A. Kumar Dipongkor *et al.*, "Reduction of multiple move method suggestions using total call-frequencies of distinct entities," *International Journal of Information Engineering and Electronic Business*, vol. 12, no. 4, pp. 21–29, Aug. 2020, doi: 10.5815/ijieeb.2020.04.03.

[20] A.-R. Al-Ghuwairi *et al.*, "Visualizing software refactoring using radar charts," *Scientific Reports*, vol. 13, no. 1, Nov. 2023, doi: 10.1038/s41598-023-44281-6.

[21] 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*, Sep. 2018, pp. 385–396, doi: 10.1145/3238147.3238166.

[22] M. Agnihotri and A. Chug, "Application of machine learning algorithms for code smell prediction using object-oriented software metrics," *Journal of Statistics and Management Systems*, vol. 23, no. 7, pp. 1159–1171, Oct. 2020, doi: 10.1080/09720510.2020.1799576.

[23] 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?," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2018, pp. 612–621, doi: 10.1109/SANER.2018.8330266.

[24] C. S. Tavares, M. A. S. Bigonha, and E. Figueiredo, "Quantifying the effects of refactorings on bad smells," in *Anais Estendidos do XI Congresso Brasileiro de Software: Teoria e Prática (CBSoft 2020)*, Oct. 2020, pp. 100–106, doi: 10.5753/cbsoft_estendido.2020.14615.

[25] M. Y. Mhawish and M. Gupta, "Predicting code smells and analysis of predictions: using machine learning techniques and software metrics," *Journal of Computer Science and Technology*, vol. 35, no. 6, pp. 1428–1445, Nov. 2020, doi: 10.1007/s11390-020-0323-7.

[26] M. M. Rahman, M. R. Rahman, and B. M. M. Hossain, "Recommendation of move method refactoring to optimize modularization using conceptual similarity," *International Journal of Information Technology and Computer Science*, vol. 9, no. 6, pp. 34–42, Jun. 2017, doi: 10.5815/ijitcs.2017.06.05.

[27] N. Erickson *et al.*, "Autogluon-tabular: robust and accurate automl for structured data," *arXiv preprint arXiv:2003.06505.*, 2020.

[28] A. Kaur, "A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes," *Archives of Computational Methods in Engineering*, vol. 27, no. 4, pp. 1267–1296, Sep. 2020, doi: 10.1007/s11831-019-09348-6.

[29] J. Pereira dos Reis, F. Brito e Abreu, G. de Figueiredo Carneiro, and C. Anslow, "Code smells detection and visualization: a systematic literature review," *Archives of Computational Methods in Engineering*, vol. 29, no. 1, pp. 47–94, Jan. 2022, doi: 10.1007/s11831-021-09566-x.

[30] A. Singjai, G. Simhandl, and U. Zdun, "On the practitioners' understanding of coupling smells- a grey literature based Grounded-theory study," *Information and Software Technology*, vol. 134, Jun. 2021, doi: 10.1016/j.infsof.2021.106539.

[31] A. AbuHassan, M. Alshayeb, and L. Ghouti, "Software smell detection techniques: a systematic literature review," *Journal of Software: Evolution and Process*, vol. 33, no. 3, Mar. 2021, doi: 10.1002/smr.2320.

[32] M. Alfadel, K. Aljasser, and M. Alshayeb, "Empirical study of the relationship between design patterns and code smells," *PLOS ONE*, vol. 15, no. 4, Apr. 2020, doi: 10.1371/journal.pone.0231731.

[33] M. Rizwan, A. Nadeem, and M. A. Sindhu, "Theoretical evaluation of software coupling metrics," in *2020 17th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, Jan. 2020, pp. 413–421, doi: 10.1109/IBCAST47879.2020.9044548.

[34] K. Ali, M. Alzaidi, D. Al-Fraihat, and A. M. Elamir, "Artificial intelligence: benefits, application, ethical issues, and organizational responses," 2023, pp. 685–702.

[35] D. Al-Fraihat, M. Alzaidi, and M. Joy, "Why do consumers adopt smart voice assistants for shopping purposes? a perspective from complexity theory," *Intelligent Systems with Applications*, vol. 18, May 2023, doi: 10.1016/j.iswa.2023.200230.

[36] R. Terra, L. F. Miranda, M. T. Valente, and R. S. Bigonha, "Qualitas.class corpus," *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 5, pp. 1–4, Aug. 2013, doi: 10.1145/2507288.2507314.

[37] M. El-Shebli, Y. Sharrab, and D. Al-Fraihat, "Correction: prediction and modeling of water quality using deep neural networks," *Environment, Development and Sustainability*, Jun. 2023, doi: 10.1007/s10668-023-03500-w.

[38] Y. Sharrab, N. T. Almutiri, M. Tarawneh, F. Alzyoud, A.-R. F. Al-Ghuwairi, and D. Al-Fraihat, "Toward smart and immersive classroom based on AI, VR, and 6G," *International Journal of Emerging Technologies in Learning (iJET)*, vol. 18, no. 02, pp. 4–16, Jan. 2023, doi: 10.3991/ijet.v18i02.35997.

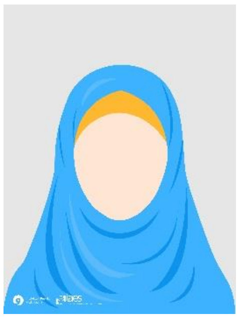
BIOGRAPHIES OF AUTHORS

Dimah Al-Fraihat received her PhD in computer science from the University of Warwick, United Kingdom. Currently, she is an assistant professor at the Software Engineering Department, Faculty of Information Technology, Isra University, Jordan. Her research interests include software engineering, requirements engineering, design patterns, software testing, refactoring, data mining, computer-based applications, technology enhanced learning, and deep learning. She can be contacted at email: d.fraihat@iu.edu.jo.

Yousef Sharrab received his Ph.D. in computer engineering from Wayne State University, USA, in 2017. He currently holds the position of assistant professor at the Department of Computer Science, Isra University. His primary research interests encompass deep learning, computer vision, speech recognition, artificial intelligence, and software engineering. He can be contacted at email: sharrab@iu.edu.jo.

Abdel-Rahman Al-Ghuwairi received his Ph.D. in computer science from New Mexico State University, USA, in 2013. He is currently an associate professor at the Software Engineering Department of Hashemite University, Jordan. His research interests encompass software engineering, cloud computing, requirements engineering, information retrieval, big data, and database systems. He can be contacted at email: Ghuwairi@hu.edu.jo.

Majed AlElaimat received his M.Sc. in software engineering from the Hashemite University, Jordan. Currently, He is a lecturer and a researcher at the Hashemite University. His research interests encompass software engineering, cloud computing, and requirements engineering. He can be contacted at email: MajedElaimat@hu.edu.jo.



Maram Alzaidi received her Ph.D. in computer science from the University of Warwick, UK. Currently, she is an assistant professor at the Faculty of Computer and Information Technology at Taif University, Saudi Arabia. Her research interests include computer-based applications, educational technology, mobile technology, technology enhanced learning, and deep learning. She can be contacted at email: mszaidi@tu.edu.sa.