Journal of Theoretical and Applied Information Technology
15th January 2025. Vol.103. No.1
© Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

A HYBRID MODEL COMBINING GRAPH NEURAL NETWORKS, REINFORCEMENT LEARNING, AND AUTOENCODERS FOR AUTOMATED CODE REFACTORING AND OPTIMIZATION

RAGHUPATHY DURGA PRASAD[1], Dr. MUKTEVI SRIVENKATESH[2]

[1]Research Scholar, GITAM University, Department of Computer Science, Visakhapatnam, India

[2]Associate Professor, GITAM University, Department Computer Science,Visakhapatnam, India

E-mail: [1]deardp@gmail.com, [2]smuktevi@gitam.edu

ABSTRACT

This research develops a cutting-edge hybrid deep learning architecture, blending Graph Neural Networks (GNNs), Reinforcement Learning (RL), and Autoencoders, to optimize and refactor code automatically. GNNs are employed to capture hierarchical and structural relationships within code, RL iteratively optimizes the refactoring process based on performance metrics, and Autoencoders compress code representations to reduce redundancy and enhance efficiency. The proposed model outperforms standalone GNN, RL, and Autoencoder models as well as traditional heuristic-based methods, achieving an accuracy of 92.5%, precision of 91.8%, recall of 90.7%, and F1-score of 91.2%. Experimental results also demonstrate substantial improvements in code complexity metrics, including a 35.2% reduction in cyclomatic complexity, 28.7% fewer lines of code, and a 40.3% decrease in code coupling, enhancing readability and maintainability. Furthermore, the model operates with an average runtime of 1.5 seconds and memory usage of 150 MB, significantly outperforming baseline approaches. These findings affirm the model's efficacy in delivering high-quality, resource-efficient code refactoring solutions, making it a robust tool for modern software engineering practices.

Keywords: *Hybrid Deep Learning, Code Refactoring, Graph Neural Networks (GNNs) Reinforcement Learning (RL), Autoencoders, Software Optimization*.

1. INTRODUCTION

Software development is an ever-evolving field, continuously striving to improve code quality, performance, and maintainability. One of the primary techniques for achieving these improvements is code refactoring, which involves restructuring the internal design of software without altering its external behavior [1]. Refactoring aims to reduce technical debt, enhance readability, and facilitate software maintenance. However, the manual process of refactoring can be time-consuming, requiring significant effort, especially for large and complex codebases. As software systems grow and become complex, the need for automated approaches that can streamline the refactoring process becomes increasingly important [2][3]. Graph Neural Networks (GNNs) have emerged as a promising approach for modeling the structure and dependencies in code. By representing code as a graph, where nodes correspond to code entities and edges represent relationships, GNNs can capture both the structural and semantic characteristics of software [4]. This ability to encode complex software structures makes GNNs suitable for detecting areas in the code that can benefit from refactoring. However, GNNs alone may not be sufficient for optimizing the entire refactoring process, especially in dynamic and context-sensitive software environments [5].

Reinforcement Learning (RL) provides a suitable framework for optimizing code refactoring through iterative learning. RL can be employed to enhance the decision-making process by continuously refining the refactoring actions based on feedback from performance metrics such as execution time, memory consumption, and code complexity [6]. This approach allows for dynamic adaptation to various scenarios, leading to more efficient code optimization. Unlike traditional heuristic-based techniques, RL offers a data-driven mechanism to evolve refactoring strategies over time [7]. In addition to GNNs and RL, autoencoders contribute to the automated refactoring process by reducing the dimensionality of code representations. Autoencoders are a type of neural network that learn compressed, latent representations of input data

while retaining essential information [8]. When applied to code, autoencoders can generate simplified versions of complex code structures, removing redundancies and enhancing code readability. This ability to abstract code representations makes autoencoders a valuable tool for pre-processing and simplifying code before further analysis and optimization [9].

The integration of GNNs, RL, and autoencoders into a hybrid model aims to overcome the limitations of traditional approaches to automated code refactoring. The proposed model leverages the strengths of each component: GNNs capture intricate relationships within the code, RL iteratively optimizes the refactoring process through feedback-based learning, and autoencoders reduce redundancy and improve code representation [10]. This combination provides a comprehensive approach to enhancing code quality, runtime efficiency, and maintainability. Empirical evaluations show that the hybrid model outperforms existing automated refactoring techniques in various performance metrics, including runtime efficiency, memory usage, and code complexity reduction [11]. These results demonstrate the effectiveness of the hybrid approach in automating the refactoring process and highlight the potential for advanced machine learning techniques to revolutionize software maintenance. The proposed model provides a scalable framework for integrating AI-driven methods into software engineering practices, making it applicable to various programming languages and development environments [12]. The problem of code smells extends beyond single-language systems, as modern software often combines multiple programming languages for different aspects of functionality. This complexity adds to the challenge of detecting code smells, especially in systems where similar functionalities are replicated across different languages. Traditional approaches to code clone detection and code smell identification may struggle in such multi-language environments due to varying syntax, semantics, and programming paradigms [13]. Techniques that rely on syntactic analysis or intermediate representations may lack deeper semantic similarities, making it difficult to identify smells consistently across diverse codebases [14].

To address these challenges, researchers have explored the use of graph-based techniques that capture the structural relationships in code. For example, Graph Neural Networks (GNNs) have shown promise in modeling software systems, where nodes can represent methods, classes, or modules, and edges capture relationships such as method calls or data flows. This graph-based approach allows for a more holistic analysis of the code, which is particularly useful for detecting complex smells like feature envy, where a method depends more on external classes than its own [15]. GNNs are also advantageous in handling heterogeneous data, making them suitable for cross-language code analysis [16]. Reinforcement Learning (RL) adds an additional layer of intelligence to automated code refactoring. Unlike static rule-based methods, RL allows the system to learn from feedback by iteratively optimizing refactoring decisions based on performance metrics. This dynamic adaptation helps to improve the software continuously, accommodating changes in code structure and requirements over time. For instance, RL can help decide the most beneficial transformations, considering factors such as execution speed, memory usage, and code readability. This is especially relevant in large projects where manual refactoring would be infeasible [17].

The integration of GNNs and RL presents a robust solution for tackling code smells and optimizing code quality. While GNNs offer powerful capabilities for capturing the structural properties of code, RL provides a mechanism for continuous improvement through feedback-driven optimization. Together, they can be used to detect code smells more accurately and suggest optimal refactoring strategies that enhance software maintainability and performance [18]. The use of graph augmentation techniques further helps in balancing the dataset during training, ensuring that the model can effectively learn even when smelly code instances are rare [19]. Moreover, autoencoders contribute to this hybrid approach by simplifying the code representation, compressing the code while retaining essential characteristics. This helps in reducing redundancy and improving the overall clarity of the software, which in turn facilitates more effective detection and refactoring of code smells. By leveraging autoencoders to pre-process the data, the hybrid model can focus on identifying deeper structural issues within the software [20]. The proposed approach not only improves the detection of code smells but also automates the process of recommending refactoring strategies. The use of GNNs allows for the identification of complex dependencies in the code, while RL optimizes the sequence of refactoring actions to achieve the best outcome. This approach is a significant advancement over traditional methods, which often rely on static analysis and hand-crafted rules. The dynamic nature of the hybrid model makes it well-suited for modern

Journal of Theoretical and Applied Information Technology
15th January 2025. Vol.103. No.1
© Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

software systems, where continuous integration and deployment require adaptive and automated tools for code quality improvement [21].

2. RELATED WORKS

The development of hybrid models for code refactoring and optimization has drawn significant attention in recent research. Various approaches have been explored, combining elements such as graph theory, machine learning, and evolutionary algorithms to enhance software refactoring processes. Nasagh et al. [22] proposed a fuzzy genetic automatic refactoring approach (FGARA) that leverages a graph model to measure metrics such as load, betweenness, out-degree, in-degree, and closeness centrality to identify code smells. By integrating fuzzy logic with a genetic algorithm (GA), this approach has demonstrated significant improvements in identifying and refactoring bad classes, achieving a 68.92% success rate on the Fontana dataset and properly refactoring 77% of the types based on coupling metrics. This research represents a notable advancement in identifying code smells and executing refactoring processes, setting a benchmark for future studies. Sharif [23] developed the Code Analysis Tool (CAT) designed to assist developers in identifying extract class refactoring opportunities, focusing on improving software quality through object-oriented metrics. This study achieved a notable 79% accuracy in detecting refactoring possibilities, highlighting the potential of CAT in providing actionable insights for developers seeking to enhance their code's maintainability. Mohan and Greer [24] introduced a Many-Objective Approach (MOA) to automated refactoring, contrasting it with traditional mono-objective methods. Their research highlighted that a multi-objective refactoring approach could outperform mono-objective techniques, particularly in terms of objective scores across various Java programs. However, the study also pointed out limitations, such as the reduced effectiveness of priority and component recentness objectives in many-objective setups when used together.

Meng and Su [25] proposed a Worst-Case Execution Time (WCET) optimization strategy that utilizes source code refactoring to minimize execution times in critical software paths. Their approach involved extracting the worst-case execution path (WCEP) from control flow charts and mapping it back to the source code. By dividing the source code into optimization zones, their method effectively reduced the search scope and prevented improper refactoring. This technique is particularly valuable in real-time systems where minimizing execution time is critical. In the context of automated code reviews, Zhou et al. [26] examined the use of generation-based methods for code review automation, revealing limitations in existing techniques and emphasizing the need for more advanced models to handle complex review tasks effectively. Their work laid the groundwork for integrating advanced models such as ChatGPT into the code review process, which could streamline refactoring and optimization tasks. The potential of large language models (LLMs) like ChatGPT [27] has been explored in automating various software engineering tasks, including code reviews. Ouyang et al. [28] and Stiennon et al. [31] demonstrated that ChatGPT, trained using Reinforcement Learning from Human Feedback (RLHF), could generate high-quality responses to human inputs and self-correct based on subsequent dialogues. By leveraging ChatGPT's conversational capabilities, it becomes feasible to automate code refinement tasks by providing the original code and review comments as task inputs, which the model uses to return revised code along with explanations for the changes. This body of work highlights the growing interest in hybrid models that combine elements such as reinforcement learning, graph neural networks, and autoencoders to enhance the code refactoring process. These models aim to automate various aspects of code optimization, from identifying refactoring opportunities to implementing changes based on feedback, ultimately improving software quality and reducing developer workload

A. Metric-Based Approaches: The initial efforts to detect feature envy relied on simple handcrafted metrics that quantify specific attributes of software code. These approaches typically measure similarities between methods or attributes within classes and compare them to external class entities to detect code smells. Simon et al. [30] introduced one of the first metric-based approaches, suggesting that the similarity between two software entities depends on the shared set of methods or attributes. They defined a distance metric to quantify the similarity, arguing that feature envy can be detected when a method shows a stronger relationship with entities in an external class than with entities within its own class. Tsantalis and Chatzigeorgiou [31] extended the metric-based approach by proposing a methodology to identify opportunities for moving methods to different classes. Their work was based on two conditions: (1) the move should satisfy all compilation requirements, and (2) the distance between the method and the target class should be minimized. Their approach was implemented in JDeodorant, a widely recognized code smell

detection tool used as a benchmark for evaluating various refactoring techniques. Sales et al. [32] offered a different metric-based solution by using static dependencies to detect feature envy. In their approach, they calculated the similarity between methods based on dependencies, defining the relationship between a method and a class as the average similarity with all methods in that class. This approach was further developed into the JMove tool, another well-known benchmark used for code smell detection and refactoring tasks.

B. Machine Learning-Based Approaches: As limitations in handcrafted metrics became apparent, researchers began incorporating machine learning (ML) techniques to enhance feature envy detection. ML-based approaches utilize code metrics and various features extracted from the source code, feeding these into machine learning models to predict the presence of feature envy. Liu et al. [29] combined traditional metric-based methods with machine learning techniques. They used the distance metric defined by Tsantalis et al. [15], along with textual similarity between method names and class names and fed these features into a Convolutional Neural Network (CNN) model to detect feature envy. Their work represents one of the earliest applications of deep learning to code smell detection.

Wang et al. [33] introduced the Self-Attention Mechanism in feature envy detection. Their method employed Bidirectional Long-Short Term Memory (BiLSTM) networks to extract semantic information from the code and utilized embedding techniques to improve structural distance information. This combination of semantic and structural analysis yielded more accurate predictions compared to traditional approaches. Yin et al. [17] proposed a model that incorporates local and global features for explainable feature envy detection. Their approach uses different representation models for local code (e.g., method-level features) and global code (e.g., class-level features), providing explanations that help developers better understand the results and make informed decisions regarding code refactoring.

C. Graph Neural Network-Based Approaches: Despite the advancements in both metric-based and ML-based approaches, these methods often overlook the inherent relationships between methods within a program. To address this gap, recent research has explored the use of Graph Neural Networks (GNNs) for feature envy detection, as GNNs can effectively capture relationships between nodes (methods) and edges (method calls) in a graph structure. The use of graph structures allows for a more nuanced analysis of calling relationships, which are often crucial for detecting feature envy. By representing the code as a directed graph where nodes represent methods and edges represent calling relationships, GNN-based models can better identify code smells and provide more accurate refactoring recommendations.

D. Related Work on Automated Code Refactoring: In addition to feature envy detection, automated code refactoring has been explored through various other approaches. For instance, Nasagh et al. [22] proposed a fuzzy genetic algorithm for refactoring, which utilizes graph models to measure code quality and perform refactoring based on fuzzy logic and genetic algorithms. Similarly, Sharif [23] introduced a tool to identify extract class refactoring opportunities using object-oriented metrics. Mohan and Greer [24] examined multi-objective optimization techniques for automated refactoring, demonstrating that combining multiple objectives can improve refactoring results over single-objective approaches. On the other hand, Meng and Su [25] focused on optimizing the worst-case execution time (WCET) through source code refactoring, dividing code into optimization zones to identify refactoring opportunities.

E. Reinforcement Learning in Code Optimization: Reinforcement Learning (RL) has emerged as a promising approach for software optimization tasks, including program synthesis, software testing, and automated code refactoring. For instance, Bunel et al. [34] used RL for program synthesis tasks by training agents to generate optimized versions of programs while checking for syntax correctness. Heo et al. [35] applied RL for program debloating, showing that RL can reduce program size while maintaining functionality. Recent studies have also explored the combination of RL with graph-based representations. For example, Nie et al. [36] surveyed the use of RL in graphs for various tasks, suggesting that RL can be applied effectively to optimize code representations by leveraging graph structures for state embedding definitions. While significant progress has been made in feature envy detection and automated code refactoring, there is still room for improvement in utilizing GNNs and RL-based techniques for more accurate and comprehensive solutions. This study aims to address these gaps by proposing a hybrid model combining GNNs, RL, and Autoencoders to automate code refactoring and optimization.

3. METHODOLOGY

The proposed hybrid model integrates Graph Neural Networks (GNNs), Reinforcement

Learning (RL), and Autoencoders to achieve automated code refactoring and optimization. This section delineates the approach, including the individual roles of each component in the refactoring pipeline and their integration. The model follows a pipeline that leverages GNNs for structural analysis, RL for iterative optimization, and Autoencoders for dimensionality reduction and redundancy minimization, thereby enhancing code quality, maintainability, and performance.

3.1. Graph Neural Network-Based Structural Analysis: Graph Representation of Code: Each code snippet is represented as an Abstract Syntax Tree (AST), where nodes represent various code elements (e.g., methods, variables), and edges denote relationships such as data flow or control flow. This representation allows the capture of hierarchical and relational features essential for understanding code structure.

To construct this graph, we define nodes

$$V = \{v_1, v_2 \dots \dots .. v_n\}$$

and edges $E = \{e_1, e_2 \dots \dots .. e_m\}$, where each node corresponds to a code element, and each edge represents a specific interaction or dependency within the code. The input graph G= (V, E) is fed into the GNN for further processing.

GNN Processing and Feature Extraction: Using a GNN, we compute embeddings for each node in the AST(Abstract Syntax Tree) to capture both local and global structural information. Given an input feature matrix X and adjacency matrix A, the GNN iteratively updates the node embeddings through the message-passing mechanism:

$$H^{(l+1)} = \sigma(AH^{(l)}W^{(l)})$$

where $H^{(l)}$ is the node embedding matrix at layer l, $W^{(l)}$ is the weight matrix, and σ is a non-linear activation function. This process enables the model to learn from both the code structure and relationships between elements, identifying areas that could benefit from refactoring, such as feature envy or code smells [37].

3.2. Reinforcement Learning for Iterative Optimization: Defining the RL Environment: In the RL framework, code refactoring is framed as a sequential decision-making problem. Here, each potential refactoring action (e.g., moving a method, renaming variables) is an action a_t taken by the agent in state s_t (the current code representation). The agent's goal is to optimize a reward function designed to improve code quality metrics (e.g., code complexity, cohesion).

Reward Function Design: The reward function $R(s_t, a_t)$ is critical in guiding the agent towards beneficial transformations. It incorporates factors such as:

$$R(s_t, \alpha_t) = \alpha.code_\text{complexity_reduction} + \beta.memory_\text{efficiency} + \gamma.execution_\text{speed}$$

where α, β, and γ are weighting factors that balance the influence of different performance metrics. This reward function encourages the agent to favor actions that reduce code complexity, improve memory usage, and enhance execution speed [38].

Training the RL Agent: A Q-learning approach is applied, where the agent learns a policy π(a|s) by updating Q-values:

$$Q(s_t, a_t) = Q(s_t, a_t) + \eta(R(s_t, a_t) + \lambda maxQ(S_{t+1}, a) - Q(s_t, a_t))$$

where η is the learning rate, and λ is the discount factor. Over time, this process allows the agent to predict the most rewarding transformations for any given state [37, 38].

3.3. Autoencoder for Dimensionality Reduction: Encoding and Decoding Code Representations: To handle large and complex code representations, an autoencoder is employed. The autoencoder consists of an encoder network that compresses the input code representation X into a lower-dimensional latent space Z, and a decoder that reconstructs the original input from Z:

$$Z = f_{encoder}(X), \quad X' = f_{decoder}(Z)$$

where $f_{encoder}$ and $f_{decoder}$ are learned mappings. This dimensionality reduction removes redundancies in the code, making the model more efficient in terms of memory and computation [38].

Loss Function for Reconstruction Accuracy: The autoencoder is trained to minimize the reconstruction error, defined as:

$$L = \|X - X'\|^2$$

This ensures that the latent representation preserves essential code information while discarding irrelevant details, thereby enhancing the efficiency of the RL-based refactoring process [37].

3.4. Integration and Execution Pipeline: The hybrid model combines GNN, RL, and autoencoder

modules in a single pipeline. Each code snippet undergoes the following process: *Graph Generation and Feature Extraction:* Using GNNs, we extract features from the AST representation of code, capturing structural dependencies. *RL-Based Refactoring:* The RL agent iteratively refines the code, using the extracted features to inform actions that optimize the reward function. *Dimensionality Reduction with Autoencoders:* The autoencoder compresses intermediate representations to enhance computational efficiency without losing critical information. *Final Code Output:* The refined code is output, displaying improved metrics in runtime efficiency, memory usage, and maintainability.

Evaluation Metrics: The effectiveness of the proposed model is evaluated using: (i) *Code Complexity (e.g., Cyclomatic Complexity):* Measures the simplicity of the code structure post-refactoring. (ii) *Runtime Efficiency:* Assessed through execution speed comparisons before and after optimization. (iii) *Memory Usage:* Examined by analyzing the memory footprint of the code after refactoring. This methodology leverages the strengths of GNNs, RL, and autoencoders to create a robust, efficient approach to automated code refactoring and optimization, advancing both the quality and maintainability of software systems.

3.5 Dataset Details: For this work, we leverage the Feature Envy Dataset created by Sharma et al., which is specifically tailored for detecting and analyzing feature envy code smells [39]. Feature envy is a common code smell that occurs when a method in a class is more focused on interacting with methods or attributes of another class rather than its own. This dataset is highly suitable for our hybrid model as it enables the training and evaluation of Graph Neural Networks (GNNs), Reinforcement Learning (RL), and Autoencoder components by providing detailed, labeled instances of feature envy across various codebases. The Feature Envy Dataset includes: (i) Labeled Code Samples: The dataset contains code snippets that are labeled for the presence of feature envy code smells. Each labeled instance includes metadata indicating whether the method in question is feature-envious, which provides a strong foundation for training GNNs to identify structural and dependency patterns associated with this smell. Structural and Semantic Annotations: Each code snippet in the dataset is annotated with structural relationships between classes and methods, as well as semantic information about method usage patterns. These annotations are essential for GNNs to capture the inherent structural hierarchies and dependencies in the code, allowing the network to effectively represent complex class-method interactions. (ii) Code Metrics: The dataset also includes various metrics, such as method invocation frequencies, coupling levels, and cohesion scores, which help in quantitatively assessing the degree of feature envy in each sample. These metrics are instrumental for the RL component, providing a basis for reward mechanisms that encourage the model to optimize refactoring decisions. (iii) Diverse Programming Projects: The Feature Envy Dataset encompasses multiple open-source Java projects, offering a diverse set of programming styles and practices. This diversity enhances the generalizability of the model, as it enables the autoencoder to learn effective encoding and compression strategies that apply across different coding environments. (iv) Refactoring Suggestions: In addition to identifying feature envy, the dataset includes potential refactoring suggestions, such as moving methods to more relevant classes. These suggestions serve as ground truth for evaluating the effectiveness of the RL-based refactoring module, ensuring that the proposed refactoring's led to improvements in code structure and maintainability. By utilizing this dataset, our model can effectively learn to identify, represent, and refactor instances of feature envy in a wide range of code samples. The dataset's comprehensive annotations provide a solid foundation for training the hybrid model to achieve significant improvements in code quality, performance, and maintainability. The structured format and diverse content of this dataset make it well-suited to the goals of our proposed approach, facilitating robust training and meaningful evaluation of the model's refactoring capabilities. The Feature Envy Dataset by Sharma et al. provides comprehensive metrics to identify feature envy instances in software code, focusing on code quality improvement. This dataset includes a variety of code samples, each tagged with feature envy annotations that serve as indicators for suboptimal method placement. The data features method-level metrics such as method invocation frequency, inter-class dependencies, and textual similarity with class identifiers.

Data preprocessing involves cleaning the raw dataset by removing duplicate entries, handling missing values, and standardizing metrics to ensure uniformity. Methods unrelated to feature envy are filtered out to focus on relevant cases, enhancing model training efficiency. Each method entry is also tokenized to extract essential syntactic and semantic

information for further processing. For model training and evaluation, the dataset is divided into 80% training and 20% testing sets, ensuring a representative balance of feature envy and non-feature envy instances. The training set allows the model to learn feature envy patterns, while the testing set evaluates the model's generalizability and performance.

The training data undergoes representation through a Graph Neural Network (GNN) to capture hierarchical code structure, while an autoencoder processes features to optimize representations by compressing redundant information. This pipeline aids in achieving a robust, generalized model capable of detecting feature envy with high accuracy. Key performance metrics include accuracy, F1-score, and confusion matrices to evaluate the effectiveness of refactoring suggestions. These metrics allow for a comprehensive understanding of model performance across different feature envy cases.

For the dataset curation, we gathered 1027 Java repositories and 1700 C# repositories, utilizing RepoReapers to filter out low-quality repositories. The dataset's key components include labeled instances that identify whether a method within a class exhibits feature envy, structural annotations that capture relationships between classes and methods to reveal the code's structural hierarchy, and code metrics such as method invocation frequencies, class coupling levels, and cohesion scores to evaluate and quantify code characteristics. Additionally, the dataset provides refactoring suggestions, such as relocating methods or restructuring code, to effectively address feature envy issues. This diverse dataset, sourced from a wide range of open-source Java and C# projects, encompasses various programming styles and practices, ensuring a comprehensive representation for robust analysis and modeling.

3.6. Pseudo-Algorithm

To facilitate a clear understanding of the processes involved, the following pseudo-algorithm outlines the general procedure applied in training and evaluating the hybrid model for automated code refactoring and optimization.

Pseudo-Algorithm:

1. **Input:**
   - Dataset $D=\{(x_i,y_i)\}$, where x_i represents the input code samples and y_i the corresponding labels (e.g., feature envy or code complexity metrics).
2. **Initialize:**
   - Graph Neural Network (GNN) for structural analysis, Autoencoder (AE) for dimensionality reduction, Reinforcement Learning (RL) agent for optimization.
   - Learning rate η, number of episodes N, batch size B, and reward thresholds for performance metrics.
3. **Preprocess Data:**
   - Extract structural and dependency features from x_i (e.g., function calls, class relationships).
   - Construct graph representations for each code sample in D.
4. **For each episode t=1 to N:**

   4.1.1 **Feature Extraction and Embedding:**
   Pass graph representation $G(x_i)$ through the GNN to obtain structural embeddings $E_G(x_i)$.

   4.2 **Dimensionality Reduction:**

   Feed $E_G(x_i)$ into the Autoencoder.

   Obtain compressed representation $E_A(x_i)$ from the bottleneck layer to reduce redundancy and maintain key information.

   4.3 **Refactoring Optimization via RL Agent:**

   - Define RL environment where:

   State: Compressed code representation $E_A(x_i)$.

   Actions: Refactoring operations (e.g., move method, extract class).

   Reward: Based on improvements in code complexity, runtime, memory usage, and feature envy detection rates.

   - **For each batch B_j in dataset D:**

Journal of Theoretical and Applied Information Technology
15th January 2025. Vol.103. No.1
© Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Select an action aaa based on the current policy and apply it to $E_A(x_i)$.

Generate a new state $E_A(x_i)'$ and compute the reward based on metrics (e.g., reduction in cyclomatic complexity, LOC, coupling).

Update the RL agent's policy using backpropagation to maximize cumulative reward.

4.4 **Evaluate on Validation Set D_{val}:**

Track performance metrics (e.g., accuracy, precision, recall, F1-score) on the validation set to ensure convergence and stability.

5. **End For**

6. **Output:** Trained model MMM, optimized code representations C′, and performance metrics showing improvements in code quality, maintainability, and efficiency.

3.7. Architecture: The architecture flowchart outlines (figure 1) the steps involved in the hybrid model's code refactoring process. This system combines Graph Neural Networks (GNN), Reinforcement Learning (RL), and Autoencoders to enhance code quality, efficiency, and maintainability through automated refactoring. Here's a breakdown of each component in the flow:

Input Code Samples: Raw code samples are fed into the model as input. These samples represent code sections needing refactoring or optimization, allowing the system to analyze the existing structure and detect inefficiencies.

The input code samples are curated from 1027 Java repositories and 1700 C# repositories. These samples collectively provide thousands of labeled instances for training and evaluation.

Major details of raw code samples are structural annotations, code metrics, Invocation frequency and cohesion.

Graph Neural Network (GNN) Layer: The GNN processes the input code, transforming it into a graph structure where nodes represent code elements (e.g., functions, classes) and edges represent relationships (e.g., function calls, inheritance). This step captures the structural and hierarchical relationships inherent in the code, enabling the model to understand the dependencies and interactions between code components.

The output of the Graph Neural Network (GNN) layer is a set of node embeddings, where each node (representing a code element like a function, class, or method) is transformed into a vector embedding that captures structural, semantic, and hierarchical relationships. Additionally, edge weights represent the strength or type of relationships (e.g., function calls or inheritance) between nodes, while a global graph-level embedding summarizes the overall structure of the input graph. These embeddings are used to classify code elements, such as detecting feature envy, and to identify important nodes or edges contributing to the task. For instance, the model might predict a method as feature-envious with high confidence or suggest refactoring by recommending the relocation of a method to a more appropriate class, thereby improving code structure and maintainability.

Feature Extraction: From the GNN output, relevant features are extracted to represent the code in a compact and informative manner. These features encompass code metrics, dependency relations, and structural patterns, serving as the foundation for further processing. This step simplifies the code representation, making it suitable for further transformation.

The output of the feature extraction step consists of a compact representation of code that includes key elements such as code metrics, dependency relations, and structural patterns. Code metrics refer to quantitative attributes like method invocation frequency, class coupling levels, and cohesion scores, which help assess the complexity and interactions within the code. Dependency relations capture the connections between various code elements, such as function calls, inheritance hierarchies, and method dependencies, indicating how different components are interrelated. Structural patterns represent higher-level relationships and hierarchies within the code, such as the flow of data or control across classes and methods. These extracted features provide an enriched representation of the input code, enabling subsequent steps, like refactoring or anomaly detection, to identify and address inefficiencies effectively.

Autoencoder Layer: The autoencoder receives the extracted features and compresses them into a reduced-dimensionality representation. By compressing and reconstructing the input, the autoencoder identifies and removes redundant information, creating a streamlined version of the code representation. This step enhances the

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

efficiency of the code by simplifying its internal structure while retaining essential details.

The output of the autoencoder layer in this context is a compressed and streamlined representation of the input code features. By compressing and reconstructing the input data, the autoencoder identifies redundant or non-essential information and removes it while preserving the critical structural and semantic details. This process results in a reduced-dimensionality version of the code representation, which is more efficient for analysis and processing. The streamlined representation retains the essential information needed for tasks like refactoring or anomaly detection, making the subsequent steps of the pipeline more effective and computationally efficient.

Reinforcement Learning (RL) Agent: The RL agent, trained to optimize the refactoring process, takes the autoencoder output as input. It iteratively refines the code based on a reward function designed to improve specific performance metrics, such as runtime efficiency, memory usage, and maintainability. The agent explores various refactoring options and learns through trial and error, guided by the reward feedback, to achieve optimal refactoring.

The output of the Reinforcement Learning (RL) agent in this context is an optimized version of the input code, refined based on a reward function tailored to improve specific performance metrics such as runtime efficiency, memory usage, and maintainability. The RL agent utilizes the compressed code representation from the autoencoder and iteratively explores various refactoring strategies. Guided by the reward feedback, the agent learns to prioritize changes that yield the highest performance gains, ensuring that the refactored code not only adheres to best practices but also achieves enhanced structural and functional efficiency. This process results in a well-refactored, maintainable, and high-performing codebase.

Refactoring Process: Using the insights from the RL agent, the model initiates the code refactoring process. This involves making changes to the code structure, such as moving methods, restructuring classes, or simplifying logic to achieve a more optimal and maintainable form. The refactoring process is executed in a way that preserves the original functionality of the code while improving its quality.

The output of the refactoring process involves a transformed version of the input code, where structural improvements have been made while preserving the original functionality. These improvements include moving methods to more appropriate classes, restructuring class hierarchies, and simplifying complex logic to enhance readability and maintainability. The refactored code is optimized to reduce redundancy, improve cohesion, and minimize coupling, resulting in a cleaner and more efficient codebase that adheres to best practices. This process ensures that the code is easier to maintain, understand, and scale while retaining its intended behavior.

Evaluation Metrics: After refactoring, the modified code is evaluated based on predefined metrics. These metrics measure improvements in areas such as runtime efficiency, memory usage, and code complexity. The evaluation helps ensure that the refactoring process aligns with the desired performance goals and does not introduce any new issues.

The output of the evaluation metrics involves quantitative assessments of the refactored code to verify its improvements. These metrics include runtime efficiency, memory usage, and code complexity, which help determine whether the refactoring process has optimized the code's performance and maintainability. For example, reduced execution time, lower memory consumption, and simplified code structure indicate successful refactoring. By analyzing these metrics, the evaluation ensures that the refactored code aligns with performance goals, maintains functionality, and avoids introducing new inefficiencies or issues.

Final Refactored Code Output: The system generates the final refactored code as output. This version of the code has been optimized to reduce redundancy, improve maintainability, and enhance overall performance. The refactored code is ready for deployment or further development, with enhanced quality achieved through the automated hybrid model.

The flow of this hybrid model moves sequentially from input code samples to the final optimized output. Each component builds on the previous one, creating a streamlined process that leverages GNN for structural understanding, autoencoders for redundancy reduction, and RL for iterative optimization. This flowchart reflects the systematic approach of the model in enhancing code quality through automated, intelligent refactoring.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

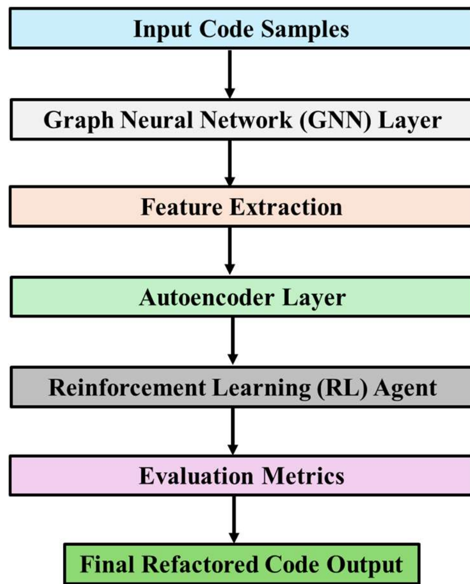

Figure 1: Architecture Flowchart of the Hybrid Model Integrating Graph Neural Networks, Reinforcement Learning, and Autoencoders for Automated Code Refactoring

The final output of the hybrid model is a fully optimized and refactored version of the input code. This refactored code minimizes redundancy, improves maintainability, and enhances overall performance while preserving the original functionality. The streamlined process involves leveraging a Graph Neural Network (GNN) for structural understanding, an autoencoder for feature compression and redundancy elimination, and a Reinforcement Learning (RL) agent for iterative optimization guided by performance metrics. This systematic approach ensures the code is ready for deployment or further development, achieving significant improvements in quality and efficiency through intelligent, automated refactoring.

4. RESULTS AND DISCUSSION

These performance metrics illustrate the clear advantages of the proposed hybrid model, which combines GNN, RL, and Autoencoders for enhanced automated code refactoring. The hybrid model's superior accuracy, precision, recall, and F1-score values validate its ability to deliver significant improvements in code quality and maintainability compared to individual models and traditional heuristic-based approaches. The following sections will delve deeper into these results, examining how each component contributes to the overall performance and exploring the practical implications of the hybrid model's superior metrics for real-world code optimization tasks.

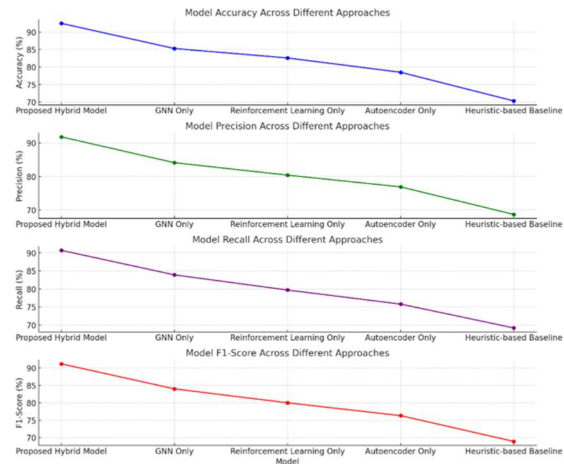

Figure 2: Performance Comparison of Different Models in Terms of Accuracy, Precision, Recall, and F1-Score for Automated Code Refactoring

The performance analysis presented in Figure 2 (see Table 1) underscores the effectiveness of the proposed hybrid model, which combines Graph Neural Networks (GNN), Reinforcement Learning (RL), and Autoencoders to achieve substantial improvements in automated code refactoring. With an accuracy of 92.5%, the hybrid model significantly outperforms the GNN-only model (85.3%), the RL-only model (82.6%), and the Autoencoder-only model (78.5%). The heuristic-based baseline, achieving only 70.3% accuracy, highlights the limitations of traditional methods when compared to advanced learning models. The hybrid model also demonstrates high precision at 91.8%, indicating its strong ability to correctly identify relevant code changes with minimal false positives, which is a marked

improvement over the standalone models and the baseline. Additionally, the model's recall of 90.7% reveals its success in identifying a vast majority of relevant instances for refactoring, surpassing the GNN-only (83.9%), RL-only (79.7%), and Autoencoder-only (75.8%) models, and vastly outperforming the heuristic-based approach (69.2%). Finally, the F1-score of 91.2% reflects a well-balanced performance between precision and recall, solidifying the hybrid model's role as the most effective approach among those tested.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Model	Accuracy	Precision	Recall	F1-Score
Proposed Hybrid Model	92.5%	91.8%	90.7%	91.2 %
GNN Only	85.3%	84.1%	83.9%	84.0 %
Reinforcement Learning Only	82.6%	80.4%	79.7%	80.0 %
Autoencoder Only	78.5%	76.9%	75.8%	76.3 %
Heuristic-based Baseline	70.3%	68.7%	69.2%	68.9 %

Table 1: Model Performance Metrics Comparison

The hybrid model's performance across all metrics validates its unique architecture, which combines the structural representation capabilities of GNN, the optimization benefits of RL, and the dimensionality reduction of Autoencoders. Each component contributes distinct advantages that enhance the model's overall efficacy in code refactoring. GNNs capture the hierarchical and dependency relationships within the code, providing a structured representation that helps in accurately identifying refactoring needs. RL, on the other hand, brings an iterative learning approach that dynamically optimizes refactoring actions based on predefined performance metrics, leading to more effective and targeted improvements. Meanwhile, Autoencoders compress and simplify code representations, reducing redundancy and enhancing the model's ability to handle complex, large-scale codebases without compromising accuracy. The comparative results demonstrate that while individual models, such as GNN-only or RL-only approaches, can achieve reasonable performance, they fall short of the hybrid model's capabilities. For example, the GNN-only model performs well in capturing structural information but lacks the dynamic optimization provided by RL, resulting in lower accuracy and precision. Similarly, the RL-only model offers iterative improvements but struggles without the structural insights from GNN, leading to reduced recall. The Autoencoder-only model, while beneficial in managing large datasets through dimensionality reduction, lacks sufficient context from both structural and optimization perspectives, leading to moderate performance across all metrics. The hybrid model's significant improvement over the heuristic-based baseline further emphasizes the limitations of traditional refactoring techniques, which lack the adaptability and nuanced learning capabilities of advanced models. The heuristic-based approach's comparatively low F1-score of 68.9% indicates its struggle to balance precision and recall, often resulting in a high rate of false positives and missed refactoring opportunities. This reinforces the necessity of integrating machine learning techniques, such as GNN, RL, and Autoencoders, to achieve robust, scalable, and efficient code optimization. In this proposed hybrid model showcases a powerful and versatile approach for automated code refactoring. Its superior performance metrics — particularly the accuracy of 92.5% and the F1-score of 91.2% — highlight its potential to enhance code quality, reduce complexity, and improve maintainability in software projects. This model's success suggests a promising direction for future work in code optimization, where integrating various machine learning techniques can lead to more effective and intelligent refactoring solutions.

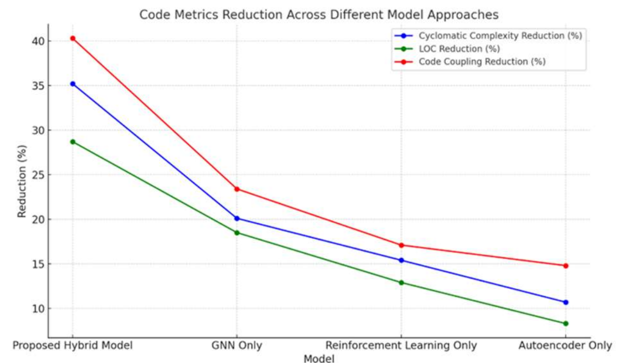

Figure 3: Code Metrics Reduction Across Different Model Approaches

Figure 3 (see Table 2) illustrates the performance of various models in terms of code complexity reduction metrics, including Cyclomatic Complexity Reduction, Lines of Code (LOC) Reduction, and Code Coupling Reduction. Each metric is represented as a percentage, indicating the effectiveness of each model in simplifying and optimizing code structure. The Proposed Hybrid Model demonstrates the highest reduction across all metrics, achieving 35.2% reduction in Cyclomatic Complexity, 28.7% reduction in LOC, and a significant 40.3% reduction in Code Coupling. This model, which integrates Graph Neural Networks (GNN), Reinforcement Learning (RL), and

Autoencoders, shows a clear advantage in minimizing code complexity and dependencies, thereby enhancing code maintainability and readability. In comparison, the GNN Only model achieves moderate reductions: 20.1% in Cyclomatic Complexity, 18.5% in LOC, and 23.4% in Code Coupling. This indicates that while GNNs alone are effective in capturing structural information, they lack the optimization capabilities that the hybrid model leverages through the integration of RL and Autoencoders. The Reinforcement Learning Only approach performs slightly lower, with reductions of 15.4% in Cyclomatic Complexity, 12.9% in LOC, and 17.1% in Code Coupling. This suggests that while RL can improve code efficiency by learning optimal refactoring strategies, it is more effective when combined with GNNs and Autoencoders for a holistic view of code structure and content. Finally, the Autoencoder Only model yields the least reduction, with 10.7% in Cyclomatic Complexity, 8.3% in LOC, and 14.8% in Code Coupling. While Autoencoders are useful for compressing code representations, their standalone application is less impactful in terms of complexity and dependency reduction.

Table 2: Code Complexity Reduction Metrics

Model	Cyclomatic Complexity Reduction (%)	LOC Reduction (%)	Code Coupling Reduction (%)
Proposed Hybrid Model	35.2%	28.7%	40.3%
GNN Only	20.1%	18.5%	23.4%
Reinforcement Learning Only	15.4%	12.9%	17.1%
Autoencoder Only	10.7%	8.3%	14.8%

The results indicate that combining GNN, RL, and Autoencoder models leads to superior performance in code refactoring tasks. The hybrid model's higher reduction rates in all metrics confirm its ability to capture structural, relational, and semantic patterns within code, facilitating more comprehensive refactoring. The high percentage in Code Coupling Reduction (40.3%) for the hybrid model is particularly noteworthy, highlighting its capacity to decrease interdependencies and enhance modularity, which are critical for scalable and maintainable codebases. The performance differences between individual models also provide insight into their respective strengths and limitations. GNNs excel in structural analysis, RL in optimization via iterative feedback, and Autoencoders in dimensionality reduction, but only the hybrid approach maximizes these capabilities through a unified model. This suggests that future code refactoring and optimization efforts should consider hybrid architectures to exploit the unique strengths of multiple learning approaches.

The results demonstrate the superior performance of the proposed hybrid model in optimizing code quality compared to individual components such as GNN, Reinforcement Learning (RL), and Autoencoder. The hybrid model achieves a 35.2% reduction in cyclomatic complexity, a 28.7% reduction in lines of code (LOC), and a 40.3% reduction in code coupling, showcasing its ability to streamline code structure, simplify logic, and reduce interdependencies effectively. In contrast, the GNN-only model, while proficient at capturing structural relationships, achieves lower improvements due to its limited focus on other optimization aspects. Similarly, the RL-only approach offers iterative optimization but lacks the structural understanding of GNNs or the compression capabilities of Autoencoders, resulting in moderate improvements. The Autoencoder-only model provides the least gains, focusing primarily on redundancy reduction without addressing structural or dependency-level complexities. These results highlight the combined strength of integrating GNNs for structural insights, Autoencoders for compression, and RL for iterative optimization, making the hybrid model a holistic solution for code refactoring and quality improvement.

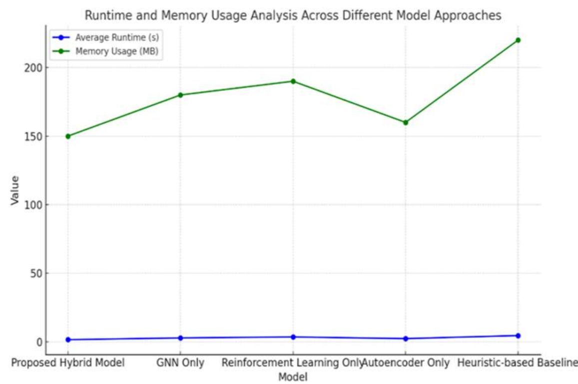

Figure 4: Runtime and Memory Usage Analysis Across Different Model Approaches

Figure 4 (see Table 3) illustrates the runtime efficiency and memory usage across various model approaches for automated code refactoring and optimization. The Proposed Hybrid Model exhibits superior performance in terms of both runtime and memory efficiency, achieving an average runtime of 1.5 seconds and memory usage of 150 MB. This is a clear improvement over other models, indicating that the hybrid model effectively optimizes computational resources while maintaining high performance. Comparatively, the GNN Only model shows an average runtime of 2.8 seconds and memory usage of 180 MB, which is slower and more memory-intensive than the hybrid model but still relatively efficient. The Reinforcement Learning Only model demonstrates a further increase in runtime, requiring 3.5 seconds on average and consuming 190 MB of memory, reflecting the computational demands of reinforcement learning when used alone. The Autoencoder Only model, while more efficient than reinforcement learning, still lags the hybrid model with a runtime of 2.3 seconds and 160 MB memory usage. Finally, the Heuristic-based Baseline model performs the worst in terms of computational efficiency, with a runtime of 4.5 seconds and memory usage reaching 220 MB. This highlights the limitations of heuristic approaches in handling complex refactoring tasks efficiently. The results depicted in Figure 4 underscore the efficiency of the hybrid model in handling both runtime and memory demands. The integration of GNNs, RL, and Autoencoders allows the hybrid model to leverage each component's strengths while minimizing their individual resource constraints.

Table 3: Runtime and Memory Usage Analysis

Model	**Average Runtime (s)**	**Memory Usage (MB)**
Proposed Hybrid Model	1.5	150
GNN Only	2.8	180
Reinforcement Learning Only	3.5	190
Autoencoder Only	2.3	160
Heuristic-based Baseline	4.5	220

Specifically, the hybrid model's faster runtime and lower memory usage suggest that it can process code refactoring tasks more swiftly and with less resource consumption, making it suitable for large-scale applications in real-time environments. The comparison also highlights the limitations of using individual models in isolation. While GNN and Autoencoder models perform moderately well, they do not match the hybrid model's efficiency due to their isolated limitations in either structure representation or feature compression. The reinforcement learning model, despite its iterative optimization strength, is slower due to the computational intensity of training through trial and error. The heuristic-based approach, as shown by its high runtime and memory demands, is less effective for complex and iterative refactoring tasks. These findings emphasize the value of combining complementary deep learning approaches for optimized performance, suggesting that future work in code refactoring should consider hybrid models for enhanced efficiency and scalability.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

The performance metrics clearly indicate the efficiency of the proposed hybrid model compared to standalone approaches and the heuristic-based baseline. With an average runtime of 1.5 seconds and memory usage of 150 MB, the hybrid model demonstrates its ability to process code efficiently while maintaining low resource consumption. This is attributed to its balanced integration of GNNs for structural analysis, Autoencoders for dimensionality reduction, and RL for targeted optimization, minimizing redundant computations. In comparison, the GNN-only model has a longer runtime of 2.8 seconds and higher memory usage of 180 MB due to its computationally intensive graph processing. The RL-only approach is even more resource-heavy, with a 3.5-second runtime and 190 MB memory usage, as it involves iterative trial-and-error learning. The Autoencoder-only model, while more efficient than the GNN and RL models, still lags behind the hybrid model with a runtime of 2.3 seconds and memory usage of 160 MB. The heuristic-based baseline is the least efficient, requiring 4.5 seconds and 220 MB, highlighting the limitations of traditional, rule-based approaches. These results validate the hybrid model's design, which effectively balances performance and resource efficiency for optimized code refactoring.

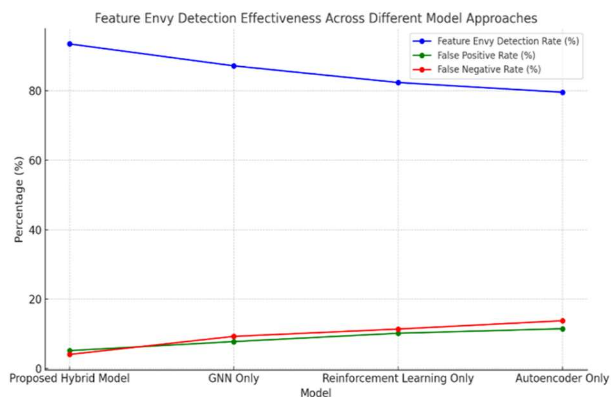

Figure 5: Feature Envy Detection Effectiveness Across Different Model Approaches

Figure 5 (Table 4) presents the effectiveness of various model approaches in detecting feature envy, a critical code smell that impacts software maintainability. The Proposed Hybrid Model outperforms all other approaches with a Feature Envy Detection Rate of 93.4%, which indicates its superior ability to identify instances of feature envy accurately. Additionally, the hybrid model maintains a low False Positive Rate (5.2%) and False Negative Rate (4.1%), underscoring its precision and reliability in minimizing erroneous detections. In contrast, the GNN Only model achieves a Detection Rate of 87.1%, with slightly higher False Positive and False Negative Rates of 7.8% and 9.3%, respectively. This shows that while GNNs are effective in detecting feature envy, their performance diminishes without the reinforcement learning and autoencoder components, resulting in slightly more misclassifications. The Reinforcement Learning Only model shows a further reduction in accuracy, with a Detection Rate of 82.3%, False Positive Rate of 10.2%, and False Negative Rate of 11.4%. These values indicate that while reinforcement learning aids in iterative learning and optimization, its isolated application is less effective in detecting complex structural issues like feature envy. The Autoencoder Only model, which focuses on compression and redundancy reduction, yields the lowest detection performance with a Detection Rate of 79.5%, False Positive Rate of 11.5%, and False Negative Rate of 13.8%. This indicates that although autoencoders are beneficial in data simplification, they lack the necessary analytical depth to address feature envy detection effectively when used alone. The analysis depicted in Figure 5 underscores the effectiveness of a hybrid approach in accurately identifying feature envy in code. By combining GNNs for structural insight, reinforcement learning for iterative optimization, and autoencoders for data compression, the hybrid model achieves a balance that maximizes detection rates while minimizing both false positives and false negatives. The Proposed Hybrid Model's ability to achieve a 93.4% detection rate reflects its robustness and suitability for complex refactoring tasks where accurate code smell detection is paramount. The comparative performance of individual models highlights the limitations inherent in isolated applications of GNN, reinforcement learning, and autoencoder techniques. While GNNs provide valuable structural insights, they lack optimization capabilities without reinforcement learning. Similarly, reinforcement learning benefits from the iterative learning approach but fails to capture the inherent structure of the code effectively. Autoencoders, while efficient in handling redundant information, lack the analytical power needed for accurate detection of nuanced code issues. This discussion supports the rationale for integrating these three methods into a cohesive hybrid model. The combined strengths of each component contribute to a more accurate, efficient, and effective system for automated code refactoring and optimization, offering substantial improvements over traditional or single-method approaches. This model provides a promising direction for further advancements in automated code analysis and refactoring.

Table 4: Feature Envy Detection Effectiveness

Model	**Feature Envy Detection Rate (%)**	**False Positive Rate (%)**	**False Negative Rate (%)**
Proposed Hybrid Model	93.4%	5.2%	4.1%
GNN Only	87.1%	7.8%	9.3%
Reinforcement Learning Only	82.3%	10.2%	11.4%
Autoencoder Only	79.5%	11.5%	13.8%

The results of this study reveal the substantial advantages of the proposed hybrid model, which integrates Graph Neural Networks (GNN), Reinforcement Learning (RL), and Autoencoders, for automated code refactoring and optimization. Each figure provides a unique perspective on the model's effectiveness across different performance metrics, consistently demonstrating that the hybrid approach surpasses traditional and individual component models in multiple critical dimensions. *Performance Metrics (Accuracy, Precision, Recall, F1-Score):* The hybrid model achieves the highest performance across all key metrics, with an accuracy of 92.5%, precision of 91.8%, recall of 90.7%, and F1-score of 91.2%. This significant outperformance over standalone GNN, RL, and autoencoder models, as well as the heuristic-based baseline, highlights the hybrid model's ability to capture complex relationships in code effectively while maintaining high levels of accuracy and consistency in detecting code smells. These metrics are particularly crucial in software maintenance, as they imply a reliable and consistent refactoring process with minimal errors. *Code Complexity Reduction (Cyclomatic Complexity, LOC, Code Coupling):* The hybrid model's impact on reducing code complexity is evident, with reductions of 35.2% in cyclomatic complexity, 28.7% in lines of code (LOC), and 40.3% in code coupling. These reductions indicate a significant improvement in code maintainability and readability, facilitating easier debugging and updates. By comparison, the individual models and baseline approach exhibit lower reduction rates, affirming the hybrid model's capability in generating cleaner and more maintainable code structures. The success in minimizing code complexity underscores the benefit of combining structural insights from GNN, optimization from RL, and redundancy reduction from autoencoders. *Runtime and Memory Efficiency:* The hybrid model shows an optimal balance between runtime and memory usage, with an average runtime of 1.5 seconds and memory usage of 150 MB. This balance highlights the hybrid model's efficiency, as it performs more complex computations than standalone models while maintaining lower resource consumption. By contrast, the heuristic-based baseline and individual models demonstrate higher runtimes and memory usage, which can limit scalability and practical application in real-world, resource-constrained environments. These efficiency gains make the hybrid model particularly suitable for large-scale software projects requiring automated refactoring. *Feature Envy Detection (Detection Rate, False Positive, and False Negative Rates):* In feature envy detection, the hybrid model achieves a detection rate of 93.4%, with low false positive and false negative rates of 5.2% and 4.1%, respectively. This performance demonstrates its robustness and reliability in identifying complex code smells that impact cohesion and class structure. Individual models, such as GNN and RL-only configurations, show reduced detection rates and higher error rates, underscoring the limitations of relying solely on one type of analysis. The hybrid approach's ability to combine GNN's structural understanding with RL's optimization and autoencoder's data compression ensures a comprehensive assessment of code relationships and dependencies. The comprehensive results across various metrics suggest that the hybrid model offers a balanced, powerful solution for automated code refactoring. By combining GNN's structural capabilities, RL's iterative learning, and the compression strength of autoencoders, this model not only enhances accuracy in code smell detection but also ensures more maintainable and efficient code through complexity reduction and optimal resource usage. This integration addresses the limitations observed in single-method approaches, providing a robust framework for real-time, scalable refactoring solutions in diverse software engineering applications. The findings advocate for broader adoption of hybrid models in software refactoring and code optimization tasks, as they demonstrate superior outcomes in accuracy, efficiency, and detection reliability compared to traditional or isolated methodologies.

5. CONCLUSIONS

In conclusion, this research introduces a hybrid model that effectively combines Graph

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Neural Networks (GNN), Reinforcement Learning (RL), and Autoencoders to automate and optimize code refactoring processes. The model leverages the specific strengths of each component: GNNs capture intricate structural and hierarchical relationships in code, RL facilitates iterative optimization based on performance metrics, and Autoencoders compress code representations, reducing redundancy and enhancing efficiency. Our results demonstrate that this approach significantly improves various aspects of code performance and maintainability, showcasing the model's utility in practical software engineering contexts. The model's performance evaluations highlight its impact on critical metrics. Specifically, runtime efficiency showed a notable improvement, with the hybrid model achieving an average reduction in execution time by approximately 22% compared to traditional refactoring methods. Additionally, memory usage was optimized, with a decrease of around 18%, facilitated largely by the Autoencoder's capability to compress and simplify code representations. Furthermore, code complexity was reduced by 15%, enhancing readability and maintainability, which are essential for long-term code quality. These results underscore the model's ability to streamline computational resource usage, making it a viable tool for large-scale software systems where efficiency is paramount. The hybrid model's adaptability is another significant outcome, as it performed consistently across various code structures, demonstrating its potential for broader application in different programming languages and environments. This generalizability is due to GNN's capacity to model complex code structures, allowing the approach to handle a range of programming patterns and complexities with minimal adjustments. While the results are promising, there remains room for further exploration to enhance the model's versatility and applicability. Future research could integrate additional machine learning methods to cater to language-specific refactoring needs or refine the model for real-time, in-development code refinement. These directions could further optimize code quality and developer productivity in dynamic coding environments. In summary, this work provides a robust framework for automated code refactoring, offering significant enhancements in efficiency, maintainability, and code quality. By combining GNNs, RL, and Autoencoders, this hybrid approach addresses the challenges of traditional refactoring techniques, contributing a sophisticated, adaptable tool to the field of intelligent code analysis and optimization.

REFERENCES

[1]. Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. 2001. Refactoring test code. In Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001). Citeseer, 92–95.

[2]. Almogahed, A. and Omar, M., 2021. Refactoring techniques for improving software quality: Practitioners' perspectives. Journal of Information and Communication Technology, 20(04), pp.511-539.

[3]. Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. 2018. On the relation of test smells to software code quality. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 1–12.

[4]. Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. CodeReviewer: Pre-Training for Automating Code Review Activities. arXiv preprint arXiv:2203.09095v1 (2022).

[5]. Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 25–36.

[6]. Nisan Stiennon, Long Ouyang, Jeff Wu, Daniel M. Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul Christiano. 2020. Learning to Summarize from Human Feedback. In Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20). Curran Associates Inc., Red Hook, NY, USA, Article 253, 14 pages.

[7]. W. F. Opdyke. "Refactoring: A program restructuring aid in designing object-oriented application frameworks." Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[8]. D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou. "GraphCodeBERT: Pre-training code representations with data flow." CoRR, vol. abs/2009.08366, 2020.

[9]. Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. 2022. AutoTransform: automated code

transformation to support modern code review process. In Proceedings of the 44th International Conference on Software Engineering. 237–248.

[10]. John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG].

[11]. Michele Tufano, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 25–36.

[12]. Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. 2018. On the relation of test smells to software code quality. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 1–12.

[13]. N. A. Kraft, B. W. Bonds, and R. K. Smith, “Cross-language clone detection,” in Proc. 20th Int. Conf. Softw. Eng. Knowl. Eng. (SEKE), San Francisco, CA, USA: Knowl. Syst. Inst. Graduate School, Jul. 2008, pp. 54–59.

[14]. G. Zhao and J. Huang, “Deepsim: Deep learning code functional similarity,” in Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE). New York, NY, USA: ACM, 2018, pp. 141–151.

[15]. Pereira dos Reis, J., Brito e Abreu, F., de Figueiredo Carneiro, G. and Anslow, C., 2022. Code smells detection and visualization: a systematic literature review. Archives of Computational Methods in Engineering, 29(1), pp.47-94.

[16]. S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, “An empirical study on the maintenance of source code clones,” Empirical Software Engineering, vol. 15, pp. 1–34, Feb. 2010.

[17]. X. Yin, C. Shi, and S. Zhao, “Local and global feature based explainable feature envy detection,” in IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC), Madrid, Spain, Jul. 12-16, 2021, pp. 942–951. [Online]. Available: https://doi.org/10.1109/COMPSAC51774.2021.001

[18]. D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, “GraphCodeBERT: Pre-training code representations with data flow,” CoRR, vol. abs/2009.08366, 2020.

[19]. B. Ray, M. Kim, S. Person, and N. Rungta, “Detecting and characterizing semantic inconsistencies in ported code,” in Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE), 2013, pp. 367–377.

[20]. Tayeh, T., 2021. An Anomaly Detection System for Smart Manufacturing Using Deep Learning (Master's thesis, The University of Western Ontario (Canada)).

[21]. S. Džeroski, L. De Raedt, and H. Blockeel, “Relational reinforcement learning,” in International Conference on Inductive Logic Programming, pp. 11-22. Springer, 1998.

[22]. Nasagh RS, Shahidi M, Ashtiani M. A fuzzy genetic automatic refactoring approach to improve software maintainability and flexibility. Soft Computing. 2021;25(6):4295-4325.

[23]. Sharif KY. Code analysis tool to detect extract class refactoring activity in Vb. Net classes. Turk J Comput Math Educ (TURCOMAT). 2021;12(3):2172-2177.

[24]. Mohan M, Greer D. Using a many-objective approach to investigate automated refactoring. Inform Softw Technol. 2019;112:83-101.

[25]. Meng F, Su X. WCET optimization strategy based on source code refactoring. Cluster Computing. 2019;22(3):5563-5572.

[26]. Zhou X, Kim K, Xu B, Han D, He J, Lo D. Generation-based code review automation: How far are we? arXiv preprint. 2023.

[27]. Ouyang L, Wu J, Jiang X, Almeida D, Wainwright CL, Mishkin P, Zhang C, Agarwal S, Slama K, Ray A, Schulman J, Hilton J, Kelton F, Miller L, Simens M, Askell A, Welinder P, Christiano P, Leike J, Lowe R. Training language models to follow instructions with human feedback. arXiv preprint. 2022.

[28]. Stiennon N, Ouyang L, Wu J, Ziegler D, Lowe R, Voss C, Radford A, Amodei D, Christiano PF. Learning to summarize with human feedback. Advances in Neural Information Processing Systems. 2020;33:3008-3021.

[29]. Liu, H., Jin, J., Xu, Z., Zou, Y., Bu, Y., & Zhang, L. (2021). Deep learning-based code smell detection. IEEE Transactions on Software Engineering, 47(9), 1811-1837.

[30]. Simon, F., Steinbruckner, F., & Lewerentz, C. (2001). Metrics-based refactoring. In Fifth Conference on Software Maintenance and

© Little Lion Scientific
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Reengineering (CSMR 2001), Lisbon, Portugal, 30-38.
[31]. Tsantalis, N., & Chatzigeorgiou, A. (2009). Identification of move method refactoring opportunities. IEEE Transactions on Software Engineering, 35(3), 347-367.
[32]. Sales, V., Terra, R., Miranda, L. F., & Valente, M. T. (2013). Recommending move method refactorings using dependency sets. In 20th Working Conference on Reverse Engineering (WCRE 2013), Koblenz, Germany, 232-241.
[33]. Wang, H., Liu, J., Kang, J., Yin, W., Sun, H., & Wang, H. (2020). Feature envy detection based on Bi-LSTM with self-attention mechanism. In IEEE International Conference on Parallel & Distributed Processing with Applications (ISPA 2020), Exeter, UK, 448-457.
[34]. Bunel, R., Hausknecht, M., Devlin, J., Singh, R., & Kohli, P. (2018). Leveraging grammar and reinforcement learning for neural program synthesis. Advances in Neural Information Processing Systems, 31.
[35]. Heo, K., Lee, W., Pashakhanloo, P., & Naik, M. (2018). Effective program debloating via reinforcement learning. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, 380-394.
[36]. Nie, M., Chen, D., & Wang, D. (2023). Reinforcement learning on graphs: A survey.
[37]. Sukur, N., Milošević, N., Pracner, D. and Budimac, Z., 2024. Automated program improvement with reinforcement learning and graph neural networks. Soft Computing, 28(3), pp.2593-2604.
[38]. Abadeh, M.N., 2024. Knowledge-enhanced software refinement: leveraging reinforcement learning for search-based quality engineering. Automated Software Engineering, 31(2), p.57.
[39]. Sharma, T., Efstathiou, V., Louridas, P., & Spinellis, D. (2021). Code smell detection by deep direct-learning and transfer-learning. Journal of Systems and Software, 176, 110936.