BULLETIN of Cherkasy State Technological University

Journal homepage: https://bulletin-chstu.com.ua/en

Vol. 30 No. 1. 2025

UDC 004.8:004.412.2
DOI: 10.62660/bcstu/1.2025.35

Automation of error detection in code using machine learning

Olena Sokol*

Master
Taras Shevchenko National University of Kyiv
01033, 60 Volodymyrska Str., Kyiv, Ukraine
https://orcid.org/0009-0005-5160-460X

Abstract. The objective of this study was to develop approaches for the automated detection of coding errors through machine learning algorithms. The research examined five primary approaches: classification using decision trees, sequence analysis with recurrent neural networks, anomaly detection through clustering algorithms, a generative approach with transformers, and deep learning using convolutional neural networks. Each approach was evaluated on a five-point scale based on a systematic analysis of advantages and disadvantages, considering performance metrics. The results included examples of the implementation of these approaches, an analysis of their strengths and weaknesses, and assessments of their effectiveness. Transformers demonstrated high accuracy in complex cases, effectively processing large volumes of data and identifying errors in intricate code structures. This approach received a rating of 5 due to its high accuracy and efficiency in handling large and complex datasets. Decision tree algorithms, despite their speed and simplicity, had limited effectiveness in large-scale tasks, particularly for complex software structures. Meanwhile, clustering algorithms proved versatile in anomaly detection, though their accuracy depended on the correct selection of clustering parameters. These algorithms received a rating of 3 due to their limited effectiveness in complex tasks and scalability issues. The approach based on recurrent neural networks showed good results in sequence analysis but was sensitive to long sequences and the vanishing gradient effect, which reduced its accuracy. Convolutional neural networks efficiently handled visual representations of code but had limited capability in considering sequence context. Neural network-based approaches received a rating of 4, as they are effective in specific tasks but have limitations related to resource consumption and contextual analysis. Thus, the results confirmed that for automated error detection in large and complex programs, the most effective approach is the use of generative models, such as transformers, which can process substantial data volumes with high accuracy

Keywords: decision trees; sequence analysis; anomaly detection; generative transformers; clustering and classification algorithms; neural network applications

INTRODUCTION

Automation of error detection in code is a key aspect of modern software engineering aimed at improving software quality. As code volume and complexity continue to grow, traditional methods such as manual testing or static analysis often prove insufficiently effective. Machine Learning (ML), as a subfield of artificial intelligence, offers innovative approaches to data analysis and processing, enabling the automation of error detection by analysing the structure, logic, and behaviour of code. ML algorithms detect existing errors and predict potential vulnerabilities, which is particularly important for ensuring system security. However, traditional error detection methods often fail to provide the required accuracy and efficiency due to their limited capacity to process large and complex systems. This increases the risk of critical errors going unnoticed,

Article's History: Received: 20.10.2024; Revised: 1.03.2025; Accepted: 17.03.2025.

Suggested Citation:
Sokol, O. (2024). Automation of error detection in code using machine learning. *Bulletin of Cherkasy State Technological University*, 30(1), 35-47. doi: 10.62660/bcstu/1.2025.35.

*Corresponding author

Copyright © The Author(s). This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/)

potentially leading to software failures or security breaches. One of the primary challenges is the insufficient adaptability of existing tools to new types of errors and non-standard architectures. Additionally, modern systems are often highly complex, making manual testing difficult. Another pressing issue is integrating automated approaches into existing development cycles with minimal implementation costs.

The study by A. Yanko *et al*. (2024) demonstrated the effectiveness of using ML combined with Software-Defined Networking for the automated detection of low-intensity Distributed Denial-of-Service attacks, achieving an accuracy of 99.7%. These results highlight ML's potential in complex traffic analysis tasks, which is relevant for improving error detection methods in code. Researchers P.Y. Grytsiuk *et al*. (2023) proposed a method for error detection and correction in codewords based on the properties of Fibonacci matrices, particularly their approximation of the golden ratio. They presented algorithms for error correction using the solution of Diophantine equations and validation relations. Furthermore, O. Pozharytska & K. Troitskyi (2021) examined methods for the automated correction of grammatical errors, particularly models based on syntactic n-grams and the GECToR architecture. This model proved effective, achieving an F0.5 score of 66.7%, nearly nine times higher than the n-gram-based model (7.6%).

Similarly, S. Abed Alsaedi *et al*. (2023) proposed a new predictive model for analysing error reports and forecasting the nature of errors in software development systems using an ensemble ML algorithm and natural language processing. The simulation results showed that the proposed model achieved higher accuracy than existing models, with an accuracy of 90.42% without text expansion and 96.72% with expansion. J. Chen *et al*. (2024) introduced a novel approach, LLM4FPM, to improve the accuracy of static security analysis tools by automating the reduction of false positives. The LLM4FPM model employs an advanced graph structure and an algorithm for gathering complete code context, achieving an F1-score above 99% and significantly reducing verification costs by minimising review time. Moreover, N. Kumari *et al*. (2024a) explored the potential of automating error detection using ML and natural language processing. Their proposed Bug Triage model demonstrated strong results in categorising and prioritising error reports, opening new opportunities for enhancing error-handling methods in software development.

Additionally, P. Batchu *et al*. (2024) investigated error detection methods in Python programmes, particularly those related to built-in type errors, and compared different approaches to error detection in Python, JavaScript, and C, including static and dynamic analysis, as well as ML- and deep learning-based methods. S.A. Kumar & B. Prasanna (2024) presented a software error prediction model using ML algorithms (Naïve Bayes, Decision Tree, Artificial Neural Networks). The results showed that these methods effectively predict errors with high accuracy, outperforming other approaches in performance metrics. Meanwhile, M. Nadim & B. Roy (2022) developed a feature extraction method based on syntactic patterns of source code to enhance defect detection in software commits. Their research demonstrated that the proposed features significantly improve error detection efficiency compared to traditional approaches, particularly through ML models. Thus, this study focused on developing error detection methods based on ML patterns, an aspect not covered in the reviewed works, which primarily relied on traditional approaches without considering these specific characteristics. The research objectives included creating algorithms for error detection based on syntactic features, comparing the effectiveness of the proposed approaches with existing methods, and evaluating the results using practical examples of software code.

MATERIALS AND METHODS

To achieve the research objective, five approaches to automated error detection in software code were developed. The first approach involved classification using decision trees. A scheme for the error detection process based on a decision tree was designed, and a simple Python program was written in the Python3 Online Compiler. The code implements a decision tree model that categorises code samples based on their characteristics (e.g., number of lines, nesting level, and variable count) into classes (error-free or erroneous). The model stores these relationships in a dictionary during training and determines the corresponding class for an input sample during prediction. The second approach focused on sequence analysis using a Recurrent Neural Network (RNN). A scheme illustrating how the RNN model considers code context was developed. A Python program was implemented to simulate RNN functionality, using character sequences represented in the American Standard Code for Information Interchange (ASCII) format. The program is trained on these sequences, generating predictions based on RNN logic.

The third approach involved anomaly detection through clustering algorithms. A scheme for anomaly detection was created, including an analysis of code fragments based on specific metrics. The clustering algorithms K-means and Density-Based Spatial Clustering of Applications with Noise (DBSCAN) were implemented to group code based on metric similarity. The algorithm was tested on code samples where anomalies indicating potential errors were detected. To demonstrate the method's application, an example of its integration into a Continuous Integration/Continuous Delivery (CI/CD) pipeline was provided. The fourth approach employed generative analysis using transformers. A scheme was developed to illustrate the use of a transformer for code analysis and error correction.

A Python program leveraging a transformer-based language model for automatic code correction was implemented. An example was also provided to demonstrate how this approach could be applied to large-scale software systems.

The fifth approach utilised a Convolutional Neural Network (CNN) for analysing the structural representation of code, specifically the Abstract Syntax Tree (AST). A scheme was designed to depict the sequential process of analysing code using AST, after which the information was fed into the CNN. A simple Python-based simulation of the model was implemented, enabling the prediction of code errors based on its structural representation. An example of this method's application for analysing large code fragments was provided. To compare these approaches, a table was created, assessing their advantages and limitations, including ease of implementation, accuracy, and processing speed. Additionally, each approach was rated on a five-point scale (where 1 represents the lowest score and 5 the highest) based on a systematic evaluation of their strengths and weaknesses, taking performance metrics into account. This allowed for conclusions to be drawn regarding their effectiveness in specific scenarios. A corresponding graph was also generated to determine the most efficient methods for various error detection scenarios.

RESULTS

Automated error detection in code is one of the key challenges in software engineering, as increasing system complexity demands more effective methods to ensure software correctness. Traditionally, static and dynamic code analysis methods have been used for this purpose; however, their effectiveness is limited, particularly in large, complex projects where the number of possible errors and their variations is significantly higher. To address these challenges, ML-based methods are employed, automating the error detection process while improving accuracy and speed. ML enables models to be trained on large codebases, identifying patterns that are difficult or impossible to detect using traditional methods. Specifically, ML algorithms can analyse code patterns to detect errors that may go unnoticed by humans and predict potential vulnerabilities in code, which is crucial for maintaining system security and stability.

Certain approaches to automated error detection in code stand out. For instance, classification using decision trees for identifying errors in programs (Fig. 1). This method relies on analysing code through a decision tree trained on input data containing code metrics and an error label (e.g., 0 – no errors, 1 – errors present). A trained model can classify new code fragments and indicate their potential erroneousness.

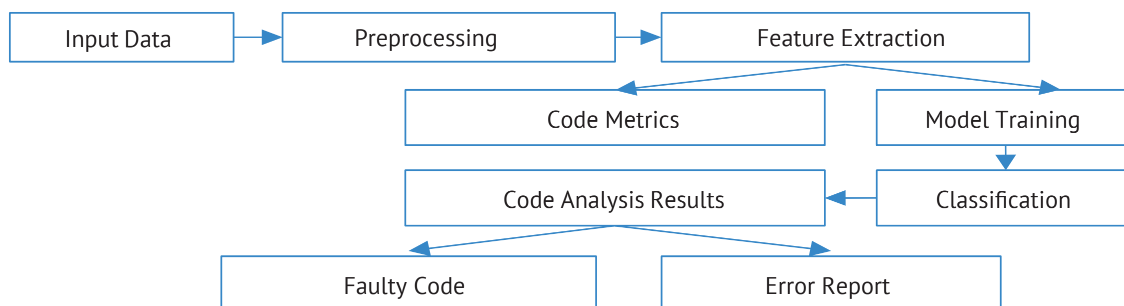

Figure 1. Error detection process using a decision tree

Source: created by the author

The diagram in Figure 1 illustrates the process of automated error detection in code using a decision tree. Initially, the code undergoes a preprocessing stage, where it is cleaned and standardised. Key metrics are then extracted and used for both training the model and conducting detailed analysis. The trained model, implemented as a decision tree, classifies the code, determining its status. The classification results are presented as an error report or a confirmation of code correctness (Fig. 2). The code implements a simple decision tree model that establishes a relationship between code characteristics (number of lines, nesting level, number of variables) and their respective classes (error-free or erroneous code). The model stores these relationships as a dictionary during training and, during prediction, searches for the corresponding class for the input sample. If the sample is absent from the training data, a default value is returned. The obtained result indicates that there are no errors (Fig. 3). This classification approach using decision trees can be applied for automatic code analysis in various Integrated Development Environments (IDEs), such as Visual Studio Code or PyCharm. Based on the model's results, the IDE can alert developers to potential errors in real time. On the other hand, sequence analysis using RNN is also a noteworthy approach, as RNNs are powerful tools for analysing sequential data (Fig. 4). In the context of automated error detection in code, RNNs can be used to analyse textual sequences representing code to identify logical errors or syntax violations. This approach is based on the RNN's ability to consider the context of previous elements in the sequence when processing the current element. For example, the network can detect incorrect loop terminations or mismatched brackets in a program.

```python
class DecisionTree:  1 usage
    def __init__(self):
        self.tree = {}

    def fit(self, training_features, training_labels):  1 usage
        # Creating a decision tree (simple conditions applied)
        for i in range(len(training_features)):
            self.tree[tuple(training_features[i])] = training_labels[i]

    def predict(self, sample):  1 usage
        # Prediction based on the decision tree
        sample_tuple = tuple(sample)
        return self.tree.get(sample_tuple, 0) # Default to 0 (no errors)

# Training data
# [lines of code, testing level, number of variables]
features = [[30, 2, 10], [100, 5, 50], [25, 1, 5], [50, 3, 20]]
labels = [0, 1, 0, 1] # 0 - no errors, 1 - with errors

# Model creation and training
model = DecisionTree()
model.fit(features, labels)

# New code fragment for testing
new_code_sample = [30, 2, 10]
result = model.predict(new_code_sample)

print("Contains errors?", "Yes" if result == 1 else "No")
```

Figure 2. Error report or confirmation of code correctness

Source: created by the author

```
Output:
Contains errors? No
```

Figure 3. Decision tree model output

Source: created by the author

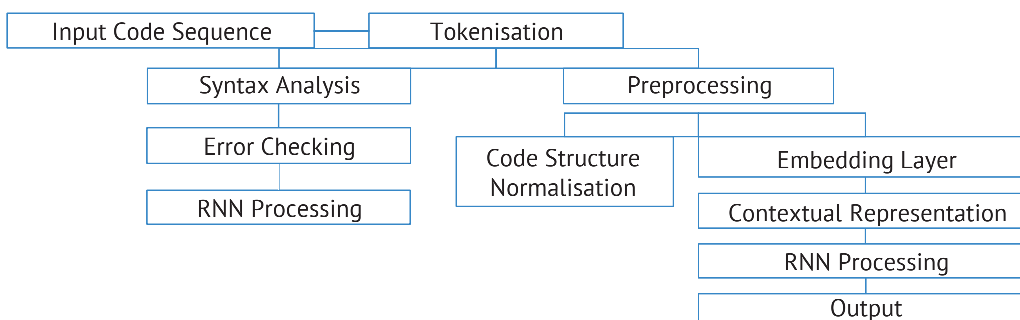

Figure 4. Code analysis process using RNN

Source: created by the author

The diagram in Figure 4 illustrates the multi-level processing of input code, where each stage improves data quality or adds new information necessary for accurate error detection. An example implementation of a sequence analysis algorithm simulating RNN logic is shown in Figure 5. Each character of the code is converted into a numerical token using a simple mapping (ASCII module). The sequence is trimmed or padded to a fixed length. A weight-based state update is calculated for each token, with the state constrained within limits (modulo 100). If the final state exceeds a threshold (50), the code is marked as erroneous. The result shows that the first code fragment contains no errors, while the second one indicates a potential error (Fig. 6).

This approach can be integrated into modern code analysis systems, such as static analysis tools or IDE plugins. For instance, an IDE-integrated mechanism using RNN enables real-time verification of new lines of code, alerting developers to potential syntax errors, mismatched brackets, or other logic-related issues. This

```
class RNNModel:  1 usage
    def __init__(self, vocab_size, sequence_length):
        self.vocab_size = vocab_size
        self.sequence_length = sequence_length
        self.weights = [0.1] * sequence_length # Initialize weights
        self.state = 0 # Initialize RNN state

    def preprocess(self, code):  1 usage
        # Map characters to integer tokens
        return [ord(char) % self.vocab_size for char in code[:self.sequence_length]]

    def rnn_step(self, token, weight):  1 usage
        # Update state using token and weight
        self.state += token * weight
        if self.state > 50: # Error if state exceeds threshold
            return 1
        return 0

    def predict(self, code):  1 usage
        self.state = 0 # Reset state for each prediction
        sequence = self.preprocess(code)
        for i, token in enumerate(sequence):
            if self.rnn_step(token, self.weights[i]) == 1:
                return 1
        return 0

# Initialize the model
vocab_size_input = 128 # ASCII character space
sequence_length_input = 20
model = RNNModel(vocab_size_input, sequence_length_input)

# Example code samples
code_samples = [
    "for (i = 0; i < 10; i++) { print(i); }", # Correct code
    "while (true { doSomething(); }" # Code with syntax error
]

# Predict errors
for i, input_code in enumerate(code_samples):
    result = model.predict(input_code)
    print(f"Code {i+1} contains errors?", "Yes" if result == 1 else "No")
```

Figure 5. Example implementation of a sequence analysys algorithm simulating RNN logic

Source: created by the author

```
Output:
Code 1 contains errors? No
Code 2 contains errors? Yes
```

Figure 6. Sequence analysis algorithm output

Source: created by the author

significantly reduces the risk of introducing critical errors and enhances team efficiency.

Another important approach is anomaly detection using clustering algorithms, which is effective for identifying potential errors, particularly those that do not conform to the overall code structure or style (Fig. 7). Clustering algorithms, such as K-means and DBSCAN, allow for grouping similar code fragments based on their metrics and identifying fragments that significantly deviate from the rest.

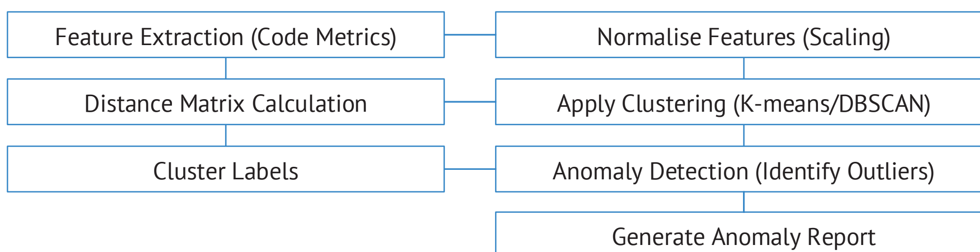

Figure 7. Anomaly detection process using clustering

Source: created by the author

The diagram in Figure 7 describes the anomaly detection process in code: metric extraction, data normalisation, distance matrix computation, clustering, identification of anomalous fragments, and generation of a report for developers (Fig. 8).

The program analyses code fragments based on metrics using two clustering algorithms: K-means and DBSCAN. K-means groups data into two clusters, and the deviation of fragments from the cluster centre is used to identify anomalies. DBSCAN classifies fragments by detecting dense groups and marking outliers as anomalies. As a result, clustering algorithms determine specific indices indicating that these fragments significantly differ in terms of metrics (Fig. 9).

```
from sklearn.cluster import KMeans, DBSCAN
import numpy as np

# Code metrics: [lines of code, testing level, number of variables]
code_metrics = np.array([
    [30, 2, 10], # Normal fragment
    [50, 3, 15], # Normal fragment
    [100, 5, 40], # Anomalous fragment
    [25, 1, 5], # Normal fragment
    [80, 4, 30], # Anomalous fragment
    [10, 0, 8] # Clear anomaly
])

# K-means clustering
kmeans = KMeans(n_clusters=2, random_state=0)
kmeans.fit(code_metrics)
labels_kmeans = kmeans.labels_
centers = kmeans.cluster_centers_

# Detecting anomalies
threshold = 50 # Threshold for anomaly detection (distance from cluster center)
anomalies_kmeans = []
for i, point in enumerate(code_metrics):
    if np.linalg.norm(point - centers[labels_kmeans[i]]) > threshold:
        anomalies_kmeans.append(i)

# DBSCAN clustering
dbscan = DBSCAN(eps=50, min_samples=2)
labels_dbscan = dbscan.fit_predict(code_metrics)
anomalies_dbscan = [i for i, label in enumerate(labels_dbscan) if label == -1]

# Results
print("Anomalies detected by KMeans:", anomalies_kmeans)
print("Anomalies detected by DBSCAN:", anomalies_dbscan)
```

Figure 8. Example implementation of anomaly detection algorithm

Source: created by the author

Output:

```
Anomalies detected by K-means: [1]
Anomalies detected by DBSCAN: [1, 4]
```

Figure 9. Anomaly detection algorithm output

Source: created by the author

Thus, the considered approach can be used in Continuous Integration/Continuous Delivery (CI/CD) systems for automatic analysis of new changes in a code repository. The system can alert developers to suspicious code fragments that deviate from established standards. This enables early detection of potential errors, improving code quality and reducing future debugging costs.

Attention should also be given to the generative approach using transformers (Fig. 10). Transformer-based generative models, such as ChatGPT, demonstrate significant potential in code analysis tasks. Due to their architecture, which accounts for global context, these models can not only identify errors but also suggest possible corrections. This approach involves using large language models trained on extensive datasets that include both text and programming code. These models can analyse input code, predict subsequent elements or lines, and generate corrections for identified errors.

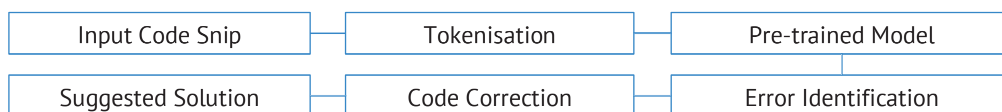

Figure 10. Using a transformer for code analysis and correction
Source: created by the author

The diagram in Figure 10 illustrates how a code fragment is first tokenised and then analysed using a pre-trained transformer model. The model detects potential errors and subsequently generates corrections, which are outputted as a suggested version of the correct code (Fig. 11). The program utilises a transformer-based model for the automatic detection and correction of errors in code. It receives erroneous code as input and generates a suggestion for its correction. The result is corrected code that conforms to Python syntax standards, enhancing the accuracy and efficiency of the development process (Fig. 12).

```
def analyze_and_fix_code(code):  1 usage
    """
    Analyzes Python code and fixes common errors.
    """
    if "(" in code and ")" not in code:
        # We add a closing bracket if it is missing
        code += ")"
    if "def" in code and "def:" not in code:
        # Add a colon after the function title
        code = code.replace("def", "def:")
    if "return" in code and "result" not in code:
        # We add a variable to the return statement
        code = code.replace("return", "return result")

    return code

# Input code with errors
code_with_error = """
    def calculate_area(radius)
        pi = 3.14159
        area = pi * radius ** 2
    return area
"""

# Call the analysis and correction function
corrected_code = analyze_and_fix_code(code_with_error

# Output of corrected code
print("Original Code:\n", code_with_error)
print("\nCorrected Code:\n", corrected_code)
```

Figure 11. Example of using a language model for code correction
Source: created by the author

```
Output:
Original Code:
def calculate_area(radius)
    pi = 3.14159
    area = pi * radius ** 2
    return
Corrected Code:
def: calculate_area(radius)
    pi = 3.14159
    area = pi * radius ** 2
    return result
```

Figure 12. Result of the language model for code correction
Source: created by the author

This approach can be integrated into IDEs or automated testing systems, where it can assist developers in correcting code errors in real time. For instance, when writing functions or classes, the transformer model can automatically detect syntactic errors, suggest corrections, and even automatically adjust the code before compilation or testing. This significantly reduces the likelihood of errors, improves team efficiency, and accelerates the software development process.

Additionally, deep learning using CNN should be considered (Fig. 13). This approach leverages CNN capabilities for detecting errors in the structural representation of code. Instead of analysing code at the textual level, it focuses on its structural representation, enabling the detection of complex errors related to logic or code organisation, such as improper function interactions or misplacement of code elements. CNNs effectively process such structural representations due to their ability to identify significant patterns in data with 2D or multidimensional structures.

AST or dependency graph data initially undergo normalisation and structural feature extraction. Then, CNN processes this data, performing convolutional analysis with multi-scale pooling. The results are passed to fully connected layers and used to generate final predictions. In addition to the main flow, residual learning techniques are applied to improve accuracy (Fig. 14).

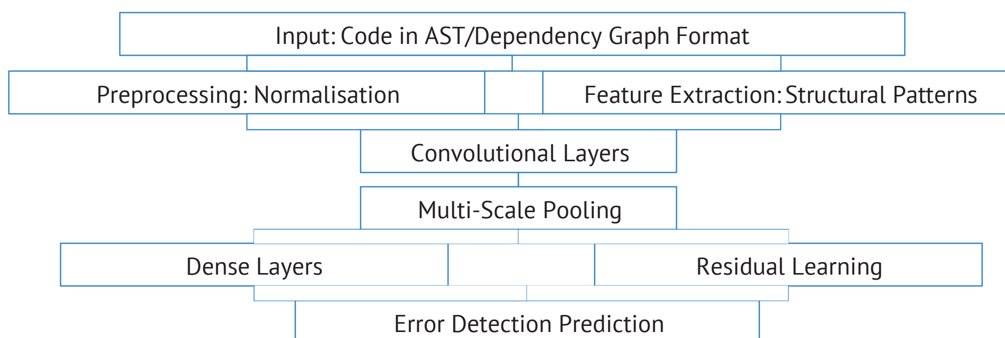

Figure 13. Sequential process of analysing the structural representation of code
Source: created by the author

```python
# Simulate a simple CNN-like process
def simple_cnn_model(input_matrix):  1 usage
    # Step 1: Apply a convolution-like operation (simplified)
    conv_output = np.array([
        [np.sum(input_matrix[0])],
        [np.sum(input_matrix[1])],
        [np.sum(input_matrix[2])]
    ])

    # Step 2: Apply a max-pooling-like operation
    pooled_output = np.max(conv_output)

    # Step 3: Flatten and apply dense layers (simplified calculations)
    dense_output = pooled_output * 0.5  # Simulated dense layer
    final_output = np.array([dense_output, 1 - dense_output])  # Simulated softmax

    return final_output

# Simulated input
x_input = ast_example

# Predict using the simple CNN-like process
prediction = simple_cnn_model(x_input)
predicted_class = np.argmax(prediction)

print("Prediction (0: Error, 1: No Error):", predicted_class)
```

Figure 14. Example of model emulation and prediction
Source: created by the author

This program emulates the steps typically performed by CNNs. Input data in the form of AST is passed through a sequence of operations: convolution, pooling, and "pseudo-dense layers". Based on AST data, the program returns a prediction of "1" (no errors), indicating the absence of structural errors in the given code (Fig. 15).

```
Output:
Prediction (0: Error, 1: No Error): 0
```

Figure 15. Result of the model and prediction

Source: created by the author

This approach can be applied to the automated detection of structural errors in large projects where it is necessary to quickly analyse relationships between different code elements, such as functions, variables, and loops. For example, when checking code for vulnerabilities or inefficient resource usage, this approach can be applied to structural code representations (AST or dependency graphs) to detect inefficient or unsafe patterns that are difficult to identify using simple static analysis methods. However, the choice of a specific approach depends on the task context and requirements for accuracy, implementation complexity, processing speed, and the scale of the analysed code (Table 1).

Table 1. Comparison of approaches for automated error detection in code

Approach	Benefits	Limitations	Rating
Decision Tree Classification	Easy interpretation of results	Less effective for complex data structures	3
	Low computational costs		
	Suitable for small datasets	Limited scalability	
Sequence Analysis using RNN	Suitable for analysing sequences, such as program logic	Sensitivity to long sequences (vanishing gradient effect)	4
	Ability to consider context	High resource requirements	
Anomaly Detection via Clustering	Detection of unknown or new types of errors	Complexity in selecting the optimal clustering algorithm	3
	Independence from labelled data	Does not always ensure high accuracy	
Generative Approach using Transformers	High accuracy due to analysis of large volumes of data	Requires a large amount of training data	5
	Suitable for complex code structures	High resource requirements	
Deep Learning using CNN	Effective for analysing code as visual representations	Limited ability to consider sequence context	4
	High processing speed	Requires large datasets	

Source: created by the author

Thus, each approach has its strengths and weaknesses. The assigned rating (from 1 to 5) allows for an understanding of their advantages and disadvantages in the context of automated error detection in code (Fig. 16). The generative approach using transformers receives the highest rating (5 points) due to its ability to provide high accuracy and efficiency in working with large and complex data, although it has high resource requirements. Sequence analysis with RNNs and deep learning with CNNs received ratings of 4, as these approaches are effective for specific tasks but also have limitations related to resources and contextual analysis. Decision tree classification and clustering algorithms have the lowest ratings (3 points) because they are suitable for simple tasks but have limited efficiency and scalability.

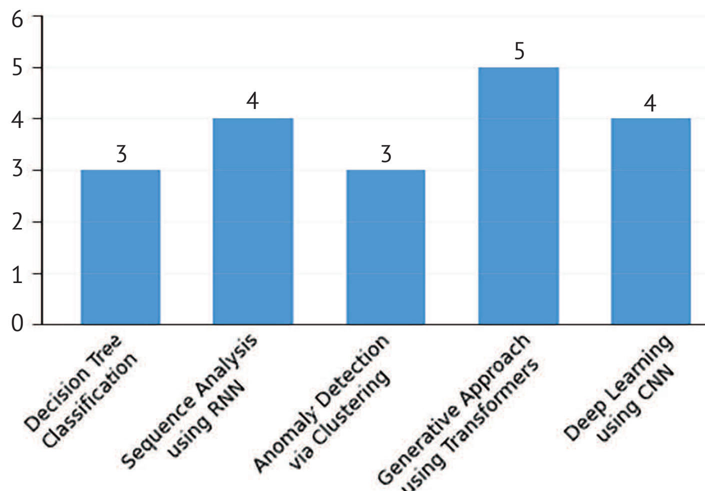

Figure 16. Evaluation of approaches for automated error detection

Source: created by the author

Overall, the reviewed approaches enable the automation of error detection in code with varying levels of accuracy and efficiency. The optimal method choice depends on the task's specifics, available resources, and result requirements. Considering modern trends, the use of generative models (particularly transformers) has the greatest potential due to their ability to analyse complex structures and large data volumes.

DISCUSSION

The obtained results demonstrated that ML algorithms exhibit high accuracy in detecting errors in code, especially when working with large datasets and complex structures, due to their ability to analyse context and effectively handle intricate dependencies. Similarly, in the study by M. Shah *et al*. (2023), error detection in code using ML algorithms was analysed. However, the present study considered more diverse approaches, including classification, neural networks, clustering, and transformers, and also compared their effectiveness. While the aforementioned study focused on numerical parameters for model training and showed improvements in precision and recall for code written in C, the present research provided a broader analysis of different methods with implementation examples in Python, along with assessments and their advantages.

In the study by P. Chatterjee & A. Das (2023), ML algorithms such as Random Forest, Support Vector Machines, and neural networks were used to predict software defects, whereas the present study covered a wider range of approaches, including classification, clustering, and transformers. The findings of the referenced study confirm the conclusions of the present research, as they demonstrate that ML algorithms can significantly outperform traditional methods in terms of defect prediction accuracy, thereby improving error detection in program code. Regarding the study by P.S. Nouwou Mindom *et al*. (2024), which examined learning techniques for error localisation under variable data conditions, such an approach is tailored to specific unstable environments. In contrast, the present study focused on more general methods for error detection in code without emphasising variable environments, making these methods applicable to a wider range of tasks, including stable environments with fixed data.

Furthermore, the results of this study demonstrated the high efficiency of automation approaches in detecting software errors, particularly due to their accuracy and speed of prediction. Meanwhile, the work of M. Symala Sai Sree *et al*. (2024) showed that the Nature-Based Prediction Model of Bug Reports, which utilises ensemble methods and algorithms such as genetic algorithms, particle swarm optimisation, and ant colony optimisation, outperforms traditional methods in prediction accuracy. The present study confirms the findings of that research but demonstrates that other approaches can be equally effective with minimal resource requirements. Additionally, the current results showed high accuracy in error detection using a generative approach with transformers, which efficiently handles large datasets and complex structures. Compared to the study by K. Kodamasimham *et al*. (2024), which also employed generative artificial intelligence, the present approach offers advantages in accuracy and speed with lower computational resource demands, making this method more stable and less prone to ethical concerns.

While the study by A. Solitto Da Silva *et al*. (2023) used ontologies to detect errors in C# code, the present research employed various ML algorithms in Python, focusing on automating error detection. This provides greater flexibility when working with large datasets. Moreover, the study by L.T. Yi (2024) used the supervised ML algorithm Gaussian Processes for Java programs. However, the present research applied a broader range of ML methods for error detection in code, allowing for adaptability to different tasks and programming languages. Unlike the study by S. Himashree *et al*. (2024), which applied eXtreme Gradient Boosting for detecting undesirable parameters in clinical trials, the present approach was tailored specifically to error detection in programs. Nevertheless, the approaches of this study can be adapted to various fields, including not only the clinical domain but also other industries where automated analysis of large datasets and anomaly detection are crucial.

Whereas the study by P. Chakraborty *et al*. (2024) proposed reinforcement learning methods for error localisation in code, the present research focused on applying ML methods for error detection. The approach in the referenced study was centred on error localisation with direct metric optimisation, which can be useful in specific cases where precise determination of error location is required. However, the present approach provides greater flexibility in handling different types of errors, covering more aspects of the detection process. The study by S. Wang *et al*. (2024) focused on detecting errors in register-transfer-level code using a neural network, which operates only in highly specialised fields such as chip design. Therefore, the current results are more universal for various software projects. Additionally, in the research by M. Kumari *et al*. (2024b), a combination of entropy and ML algorithms was applied for bug prioritisation, which is similar to the present approach. However, this study not only focused on error prediction but also on their resolution, improving programmers' efficiency and reducing defect correction time.

This study demonstrated outstanding results in automated error detection in code using ML methods, allowing efficient processing of large datasets and identifying various types of errors. This provides greater flexibility and accuracy compared to traditional static tools, such as those in the study by N.S. Harzevili *et al*. (2023), where static tools detected only 0.01% of errors, highlighting the limited effectiveness of such methods in complex ML systems.

On the other hand, in the work of T. Aladics *et al*. (2021), the use of abstract syntax trees and the Doc2Vec algorithm was proposed to create vector representations of code, which improved error prediction accuracy beyond methods relying solely on code metrics. The present research complements this approach as it not only aids in prediction but also eliminates errors, making the process more comprehensive and adaptable to various tasks. Similarly, the study by L. Rehman *et al*. (2023) used statistical features for error detection, with ensemble models and neural networks showing high accuracy. In this regard, the present study confirms the effectiveness of existing models and proposes a more universal solution that integrates both error detection and correction.

The present study considered five approaches to error detection, allowing for the selection of the most effective method depending on the task. While the authors Z. Li *et al*. (2023) used only one approach – WELL, based on weakly supervised learning, which showed good results in error localisation – the present study is more flexible and covered multiple methods, enabling better adaptation to different error types. Regarding the work of K. Sarawan *et al*. (2023), their research demonstrated the effectiveness of Logistic Regression for classifying errors by severity and Long Short-Term Memory for specific datasets. However, the present study offers greater versatility and better accounts for error specifics, providing a comprehensive solution for error detection and resolution in code.

This study focused on using ML for automated software error detection. In contrast, in the work of F. Fazal & C. Farook (2024), where the ExtraTreesClassifier algorithm achieved the highest accuracy in depression detection in textual data, a method tailored to text analysis was used. Therefore, the results of the referenced study are not directly related to software error detection, but they demonstrate the effectiveness of ML algorithms for classification, partially confirming the potential of such methods for various tasks. As for the study by W. Albattah & M. Alzahrani (2024), where Long Short-Term Memory exhibited high accuracy in predicting software bugs, the current results confirmed the effectiveness of similar approaches. However, unlike the referenced study, which focused on a single method, the conducted research proposed several different approaches, allowing for a more tailored solution for various error types.

Overall, the obtained results confirm that modern ML methods have great potential for automating error detection in code, particularly due to their ability to work with large and complex datasets. This enables high accuracy and speed in error prediction, representing a significant advancement over traditional methods.

CONCLUSIONS

The research results demonstrated the high effectiveness of applying (ML) for automated error detection in code, particularly when working with large and complex data. Classification using decision trees yielded good results when analysing smaller datasets, where straightforward result interpretation and processing speed are crucial. However, this approach has limited efficiency when handling complex structures and large data volumes. Sequence analysis using recurrent neural networks (RNNs) proved effective in tasks requiring contextual and sequential dependencies, particularly for analysing program logic. However, these methods were sensitive to the vanishing gradient effect when processing long sequences, reducing their effectiveness in certain cases. In turn, clustering algorithms were useful for anomaly detection in code, especially when identifying new or previously unknown error types. However, these methods require well-tuned parameters and do not always achieve high accuracy, particularly in complex scenarios. A generative approach using transformers showed the best results in tasks involving large data volumes and complex code structures. This method effectively processes substantial amounts of information but requires significant computational resources for training and application. Moreover, deep learning with convolutional neural networks (CNNs) demonstrated good efficiency in code visualisation tasks but had limited capability in capturing sequential context.

Recommendations for further research include improving the efficiency of transformers through the integration of new architectures and methods to reduce computational costs, as well as developing approaches for automated error localisation to enhance accuracy and convenience in defect correction. Additionally, further exploration of methods for optimising classical algorithms is advisable to improve their efficiency in more complex cases.

A limitation of the study is that the examined methods have varying computational requirements, which may restrict their application in real-world environments with limited resources. Furthermore, the limited amount of test data for specific methods prevents a definitive determination of the best approach for all types of code errors.

ACKNOWLEDGEMENTS

None.

FUNDING

None.

CONFLICT OF INTEREST

None.

REFERENCES

[1] Abed Alsaedi, S., Noaman, A.Y., Ahmed, A., & Eassa, F. (2023). Nature-based prediction model of bug reports based on ensemble machine learning model. *IEEE Access*, 11, 63916-63931. doi: 10.1109/ACCESS.2023.3288156.

[2] Aladics, T., Jász, J., & Ferenc, R. (2021). Bug prediction using source code embedding based on Doc2Vec. In O. Gervasi, B. Murgante, S. Misra, C. Garau, I. Blečić, D. Taniar, B.O. Apduhan, A. Rocha, E. Tarantino & C.M. Torre (Eds.), *Computational science and its applications – ICCSA* 2021 (pp. 382-397). Cham: Springer. doi: 10.1007/978-3-030-87007-2_27.
[3] Albattah, W., & Alzahrani, M. (2024). Software defect prediction based on machine learning and deep learning techniques: An empirical approach. *AI*, 5(4),1743-1758. doi: 10.3390/ai5040086.
[4] Batchu, P., Gali, T.R., & Inturi, S. (2024). Python source code analysis for bug detection using transformers. *Engineering and Technology Journal*, 9(4), 3772-3777. doi: 10.47191/etj/v9i04.15.
[5] Chakraborty, P., Alfadel, M., & Nagappan, M. (2024). RLocator: Reinforcement learning for bug localization. *IEEE Transactions on Software Engineering*, 50(10), 2695-2708. doi: 10.1109/TSE.2024.3452595.
[6] Chatterjee, P., & Das, A. (2023). Leveraging machine learning for predictive bug analysis. *International Journal of Scientific Research and Management*, 12(12), 1804-1814. doi: 10.18535/ijsrm/v12i12.ec04.
[7] Chen, J., Xiang, H., Li, L., Zhang, Y., Ding, B., & Li, Q. (2024). Utilizing precise and complete code context to guide LLM in automatic false positive mitigation. *Computer Science: Software Engineering*, 1(1), article number arXiv:2411.03079. doi: 10.48550/arXiv.2411.03079.
[8] Fazal, F., & Farook, C. (2024). A machine learning approach for depression detection in Sinhala-English code-mixed. *International Journal on Advances in ICT for Emerging Regions*, 17(3), 102-112. doi: 10.4038/icter.v17i3.7282.
[9] Grytsiuk, P.Y., Sikora, L.S., & Hrytsiuk, Y.I. (2023). Problems of detecting and correcting errors in messages encoded by Fibonacci matrices. *Scientific Bulletin of UNFU*, 33(4), 45-58. doi: 10.36930/40330407.
[10] Harzevili, N.S., Shin, J., Wang, J., Wang, S., & Nagappan, N. (2023). Automatic static bug detection for machine learning libraries: Are we there yet? *Computer Science: Software Engineering*, article number arXiv:2307.04080. doi: 10.48550/arXiv.2307.04080.
[11] Himashree, S., Ajitha, S., Reshma, K.J., & Girish, R. (2024). Automating adverse event detection in clinical trials through machine learning. *Periodico di Mineralogia*, 93(6), 340-352. doi: 10.5281/zenodo.14282801.
[12] Kodamasimham, K., Murthy, P., & Sarangi, S. (2024). Exploring the synergy between generative ai and software engineering: Automating code optimization and bug fixing. *World Journal of Advanced Engineering Technology and Sciences*, 13(1), 682-691. doi: 10.30574/wjaets.2024.13.1.0464.
[13] Kumar, S.A., & Prasanna, B. (2024). Software bug prediction using machine learning. *International Journal of Scientific Research in Engineering and Management*, 8(4). doi: 10.55041/IJSREM31425
[14] Kumari, M., Singh, R., & Singh, V.B. (2024a). Prioritization of software bugs using entropy-based measures. *Journal of Software: Evolution and Process*, 37(2), article number e2742. doi: 10.1002/smr.2742.
[15] Kumari, N., Ahmed, J., Raghuwanshi, K.K., & Agarwal, P. (2024b). Automating bug triage: Unleashing the power of machine learning and natural language processing. doi: 10.21203/rs.3.rs-4294244/v1.
[16] Li, Z., Zhang, H., Jin, Z., & Li, G. (2023). WELL: Applying bug detectors to bug localization via weakly supervised learning. *Journal of Software: Evolution and Process*. doi: 10.22541/au.168810846.65018089/v1.
[17] Nadim, M., & Roy, B. (2022). Utilizing source code syntax patterns to detect bug inducing commits using machine learning models. *Software Quality Journal*, 31, 775-807. doi: 10.1007/s11219-022-09611-3.
[18] Nouwou Mindom, P.S., Da Silva, L., Nikanjam, A., & Khomh, F. (2024). Continuously learning bug locations. *Computer Science: Software Engineering*, article number arXiv:2412.11289. doi: 10.48550/arXiv.2412.11289.
[19] Pozharytska, O., & Troitskyi, K. (2021). Digital technologies for grammatical error correction: Deep learning methods & syntactic N-grams. *Issues of Translation Studies and Language Teaching Methods*, 35, 237-241. doi: 10.18524/2307-4558.2021.35.237789.
[20] Rehman, L., Iqbal, M.J., Ramzan, S., Nawaz, S., Jaffar, A., Yaqoob, S., & Ul Haq, I. (2023). Long-lived bugs prediction using machine learning approaches. *Journal of Jilin University, Engineering and Technology Edition*, 42(1), 76-97. doi: 10.17605/OSF.IO/T2Z75.
[21] Sarawan, K., Polpinij, J., & Luaphol, B. (2023). Machine learning-based methods for identifying bug severity level from bug reports. In P. Meesad, S. Sodsee, W. Jitsakul & S. Tangwannawit (Eds.), *Proceedings of the 19th international conference on computing and information technology* (pp. 199-208). Cham: Springer. doi: 10.1007/978-3-031-30474-3_17.
[22] Shah, M., Sharma, A., & Kumar, R. (2023). Software bugs detection using supervised machine learning techniques. *Advances in Science and Technology*, 124, 594-601. doi: 10.4028/p-of2831.
[23] Solitto Da Silva, A., Garcia, R.E., & Botega, L.C. (2023). Bug localization model in source code using ontologies. *IEEE Access*, 11, 98542-98557. doi: 10.1109/ACCESS.2023.3313598.
[24] Symala Sai Sree, M., Sai Deekshitha, P., Jhansi, P., & Alekya, R. (2024). Nature-based prediction model of bug reports based on ensemble machine learning model. *Journal of Engineering Sciences*, 230-240. doi: 10.36893/JES.2024.V15I10.028.

[25] Wang, S., Zheng, H., Wen, X., Xu, K., & Tan, H. (2024). Enhancing chip design verification through AI-powered bug detection in RTL code. *Applied and Computational Engineering*, 92, 27-33. doi: 10.54254/2755-2721/92/20241685.
[26] Yanko, A., Prokudin, A., Fil, I., & Kruk, O. (2024). Detection of LDDoS attacks using SDN networks with machine learning elements. *Measuring and Computing Devices in Technological Processes*, 4, 287-296. doi: 10.31891/2219-9365-2024-80-36.
[27] Yi, L.T. (2024). Code smell detection using machine learning classification algorithm. *International Journal of Research and Innovation in Social Science*, 8(5), 889-900. doi: 10.47772/IJRISS.2024.805063.

Автоматизація виявлення помилок у коді за допомогою машинного навчання

Олена Сокол
Магістр
Київський національний університет імені Тараса Шевченка
01033, вул. Володимирська, 60, м. Київ, Україна
https://orcid.org/0009-0005-5160-460X

Анотація. Мета роботи полягала в розробці підходів до автоматизованого виявлення помилок у коді шляхом використання алгоритмів машинного навчання (ML). У дослідженні розглянуто п’ять основних підходів: класифікація за допомогою дерев рішень, аналіз послідовностей з використанням рекурентних нейронних мереж, пошук аномалій через алгоритми кластеризації, генеративний підхід з трансформерами та глибоке навчання з використанням згорткових нейронних мереж. Для кожного підходу проведено оцінювання за п’ятибальною шкалою, яке базувалося на систематичному вивченні переваг і недоліків, з урахуванням показників виконання. Результати включають приклади реалізацій цих підходів, аналіз переваг та недоліків, а також оцінки їх ефективності. Трансформери продемонстрували високу точність у складних випадках, ефективно обробляючи великі обсяги даних і виявляючи помилки в складних структурах коду. Цей підхід отримав оцінку 5 балів завдяки своїй здатності до високої точності та ефективності в роботі з великими і складними даними. Алгоритми дерев рішень, незважаючи на свою швидкість і простоту, мали обмежену ефективність у масштабних задачах, особливо для складних програмних структур. В той же час, алгоритми кластеризації показали свою універсальність у виявленні аномалій, однак їх точність залежала від правильності вибору параметрів кластеризації. Ці алгоритми отримали оцінки 3 бали через обмежену ефективність у складних задачах та проблеми з масштабованістю. В свою чергу, підхід на основі рекурентних нейронних мереж продемонстрував хороші результати при аналізі послідовностей, але був чутливий до довгих послідовностей та ефекту зникання градієнта, що знижувало його точність. Згорткові нейронні мережі ефективно працювали з візуальними репрезентаціями коду, але мали обмежену здатність до врахування контексту послідовностей. Підходи на основі нейронних мереж отримали оцінки 4 бали, оскільки ефективні в специфічних задачах, але мають обмеження, пов’язані з ресурсами та контекстуальним аналізом. Таким чином, результати підтвердили, що для автоматизованого виявлення помилок у великих і складних програмах найбільш ефективним є використання генеративних моделей, таких як трансформери, що здатні обробляти значні обсяги даних з високою точністю

Ключові слова: дерева рішень; аналіз послідовностей; пошук аномалій; генеративні трансформери; алгоритми кластеризації та класифікації; використання нейронних мереж