2025 10th International Conference on Computer Science and Engineering (UBMK) | 979-8-3315-9975-1/25/$31.00 ©2025 IEEE | DOI: 10.1109/UBMK67458.2025.11206829

Software Metrics vs. Code Text: Feature Representations for ML- and DL-Based Vulnerability Detection

İbrahim Tarakcı
Department of Computer Engineering
Middle East Technical University
Ankara, Türkiye
itarakci@metu.edu.tr

Selma Nazlıoğlu
Department of Software Engineering
Atılım University
Ankara, Türkiye
selma.suloglu@atilim.edu.tr

***Abstract*—This study evaluates traditional Machine Learning (ML) and Deep Learning (DL) models for software vulnerability detection using two feature representations: software metrics and raw source code text. Experiments focus on C/C++ vulnerabilities using Array Usage and Arithmetic Expression datasets. We assess how vectorization techniques (CountVectorizer, TF-IDF, and Word2Vec) affect the performance of Random Forest and K-Nearest Neighbors (ML models), and Multi-Layer Perceptron and Long Short-Term Memory (DL models) when applied to text-based code. Experiments on both balanced and imbalanced datasets across multiple vulnerability types reveal that representation choice significantly i nfluences model performance. Deep learning models perform better with code text, while traditional machine learning models benefit from software metrics. Vectorization strategy also impacts results, particularly for text-based inputs. Furthermore, class imbalance notably affects model behavior. These findings o ffer practical insights into how feature representation, model architecture, pre-processing, and data balance affect the effectiveness of automated vulnerability detection.**

Keywords*—*software vulnerability detection, machine learning, deep learning, software metrics, source code analysis

I. Introduction

In today's digital landscape, software powers critical infrastructure and everyday technologies, making its security more crucial than ever. Vulnerabilities in source code, flaws that can be exploited by attackers, pose serious risks to software confidentiality, integrity, and availability. Traditional static analysis tools, while widely used, often suffer from high false-positive rates and limited adaptability to modern, complex codebases.

Machine learning (ML) and deep learning (DL) have emerged as promising alternatives, capable of learning from labeled examples to detect vulnerabilities in unseen code. However, their effectiveness largely depends on how the code is represented. Common approaches include software metrics (capturing structural attributes), raw source code text (leveraging natural language processing techniques), and graph-based representations like abstract syntax trees (ASTs) or code property graphs (CPGs), which model syntactic and semantic relationships.

Each representation demands specific pre-processing. Software metrics require extraction and normalization, code text must be tokenized and vectorized (e.g. via CountVectorizer, TF-IDF, or Word2Vec), and graph-based approaches involve constructing code graphs and applying embedding techniques. Another major challenge is class imbalance. Vulnerable code samples are typically much rarer than safe ones, making it difficult for models, particularly DL ones, to learn effectively.

While prior work explores ML [1]–[3] and DL [4], [5] approaches to vulnerability detection, most focus on a single model type or code representation. To our knowledge, no prior work has directly compared metrics and text representations under identical conditions, especially related to vectorization techniques and balancing conditions within a unified framework. Thus, it remains unclear which approaches perform best under different real-world conditions.

Given these considerations, this study investigates how different source code representations impact the performance of ML and DL models in software vulnerability detection. We focus on how representation type, pre-processing strategies, model architecture, and class imbalance collectively influence detection effectiveness. To guide our analysis, we address the following research questions:

- **RQ1.** How does the representation of source code, either as software metrics or as code-as-text, affect the performance of traditional machine learning and deep learning models in vulnerability detection?
- **RQ2.** How do vectorization techniques (CountVectorizer, TF-IDF, Word2Vec) influence the performance of machine learning and deep learning models when using code-as-text?
- **RQ3.** How significantly does class imbalance affect model performance across different code representations and vectorization techniques?
- **RQ4.** Which model family, traditional ML or DL models, performs more effectively for different code representations?

To address these questions, we conduct a comprehensive experimental study using four models (Random Forest, K-Nearest Neighbors, Multi-Layer Perceptron, and Long Short-

979-8-3315-9975-1/25/$31.00 ©2025 IEEE

Authorized licensed use limited to: ULAKBIM UASL - MIDDLE EAST TECHNICAL UNIVERSITY. Downloaded on November 07,2026 at 21:38:32 UTC from IEEE Xplore. Restrictions apply.

Term Memory) evaluated on both balanced and imbalanced datasets. We examine two code representations (software metrics and raw code text) and apply three vectorization techniques (CountVectorizer, TF-IDF, and Word2Vec) to convert text-based code into feature vectors. The main contributions of this work are: (i) a comparative analysis of traditional ML and DL models across different code representations and pre-processing strategies, (ii) an investigation of the impact of vectorization techniques in text-based vulnerability detection, and (iii) an examination of how class imbalance affects model performance. Based on these findings, we provide practical insights to guide the selection of models, representations, and pre-processing methods for effective software vulnerability detection.

The remainder of this paper is organized as follows. Section II introduces key concepts related to software vulnerabilities, machine learning, deep learning, and vectorization techniques. Section III reviews relevant prior work. Section IV describes the datasets, models, and evaluation metrics, along with the experimental setup. Section V presents and discusses the results, including answers to the research questions. Finally, Section VI concludes the study and outlines directions for future research.

II. Fundamentals

A. Software Vulnerability and Detection

Software vulnerabilities are weaknesses in code that attackers can exploit to break, misuse, or access a system. These often come from coding mistakes such as logic errors, arithmetic mistakes, unsafe pointer usage, or incorrect array handling. Such problems are especially common in low-level languages like C and C++. Improper use of third-party libraries and APIs can also create security risks. Detecting these issues early in development helps build safer software.

To find vulnerabilities, traditional methods like static analysis tools and formal verification have been used [6], [7]. However, they usually require expert knowledge and may not work well on different codebases. More recently, machine learning (ML) is widely used for detecting software vulnerabilities. Traditional models like Random Forest (RF) and K-Nearest Neighbors (KNN) perform well with structured features like software metrics [1], [2], and some studies have also applied them to source code text by tokenizing and vectorizing it into numerical features [3], [4].In this study, RF is chosen for its strong performance and resistance to overfitting [1], while KNN is used as a simple, fast baseline model [4].

Deep learning (DL) models like MLPs, LSTMs, and CNNs learn complex patterns and have been widely used for this task too [3]–[5]. MLPs are great for structured data, while LSTMs and CNNs are better with text. More recently, Large Language Models (LLMs) like CodeBERT, GraphCodeBERT, and SecBERT have taken this further by understanding deep context in code [8], [9]. This work uses MLP for structured inputs and LSTM for both structured and text-based features to compare how different models handle different code representations.

B. Software Metrics

Software metrics are structured, interpretable features that quantify aspects like size, complexity, and maintainability, and are widely used in vulnerability detection as effective inputs for machine learning models. Two prominent metric families are Halstead's complexity measures [20] and Mc-Cabe's cyclomatic complexity [21]. Halstead metrics rely on counting operators and operands to derive values like pro-gram length, volume, effort, and estimated bugs. McCabe's metric quantifies control flow complexity by counting the number of independent execution paths. Their numerical na-ture makes them well-suited for models such as MLPs. These metrics capture structural and cognitive aspects of code, such as high cyclomatic complexity or Halstead volume, which may indicate vulnerability risk.

C. Vectorization Techniques

Vectorization transforms source code into numerical formats suitable for machine learning and deep learning models. In this study, we focus on three widely used vectorization techniques: CountVectorizer, TF-IDF, and Word2Vec.

CountVectorizer creates a sparse matrix based on token frequencies in each code snippet. Though fast and simple, it does not consider the relative importance of tokens [12]. TF-IDF (Term Frequency–Inverse Document Frequency) improves upon this by down-weighting common terms and emphasizing rare, informative tokens, helping models focus on distinctive features [13]. Word2Vec captures contextual meaning by generating dense embeddings. We trained a Skip-gram model on our dataset. For KNN, RF and MLP, we averaged token vectors to create fixed-length inputs. For LSTM, we preserved token sequences with padding, enabling temporal learning. This flexibility made Word2Vec effective across both model types [14].

III. Related Work

Software vulnerability detection has been extensively studied using both ML and DL techniques. Two dominant approaches have emerged: representing code through software metrics, and treating code-as-text using embedding and vectorization methods. Several studies have demonstrated that software metrics effectively capture structural properties linked to vulnerabilities. Du, Li, and Yin proposed the LEOPARD framework [15], which uses program metrics to identify potentially vulnerable functions without requiring labeled data. Their work reveals the utility of static features for pre-screening risky code. Zagane et al. applied DL models to Halstead metrics and other structural features for vulnerabil-ity detection in C code [4], while Viszkok et al. found that 35 static code metrics could predict vulnerable JavaScript functions [16].

More recent work has shifted toward textual representations of code and NLP-based techniques. Bagheri and Hegedűs compared Word2Vec, fastText, and BERT embed-

Authorized licensed use limited to: ULAKBIM UASL - MIDDLE EAST TECHNICAL UNIVERSITY. Downloaded on November 07,2026 at 21:38:32 UTC from IEEE Xplore. Restrictions apply.

dings with LSTM models for detecting vulnerabilities in Python [17]. They found that contextual embeddings like BERT yielded superior results. Similarly, Wartschinski et al. proposed VUDENC, using Word2Vec with LSTM to detect real-world Python vulnerabilities, achieving high F1 scores across datasets [18].

Kalouptsoglou et al. showed that text-based models outperform metric-based ones and that combining both yields minimal benefit [19]. However, their study only focused on DL, omitting traditional ML models. In contrast, our work compares both ML and DL models using consistent experimental setups across multiple vectorization methods and balanced vs. imbalanced datasets.

While existing studies tend to focus on either metrics or text-based representations in isolation, no prior work, to our knowledge, has systematically compared both approaches using identical models and conditions. Addressing this gap, our study evaluates CountVectorizer, TF-IDF, and Word2Vec across four model types (KNN, RF, MLP, and LSTM) under varying data balance scenarios. This comprehensive analysis clarifies how representation and model choice affect vulnerability detection performance and offers practical guidance for future research and applications.

IV. Methodology

This section outlines the structured methodology adopted to answer research questions given in Section I, illustrated in Fig. 1. The experimental workflow includes dataset balanc-ing, train-test splitting, pre-processing, model training, and performance evaluation. We used two datasets, Array Us-age and Arithmetic Expression, elaborated in Section IV-A. They are composed of labeled code snippets annotated for vulnerability. To reflect real-world conditions, we first used the datasets in their original imbalanced form. To examine the impact of class balance, we also created balanced versions by applying random undersampling to the majority class (non-vulnerable or clean samples). The distribution of code samples for the balanced and imbalanced versions of each dataset is provided in Table I.

All datasets were split consistently into training and testing sets, with 80% used for training and 20% for testing. These splits were fixed and shared across all models and representations to ensure fair comparisons.

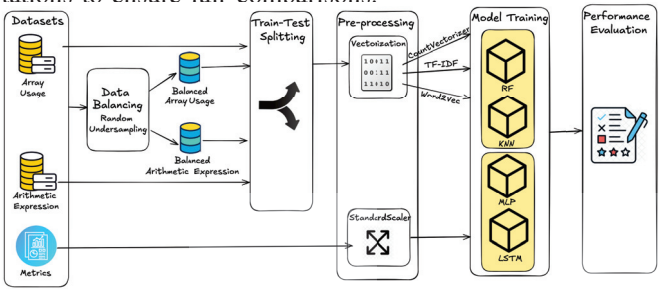


Fig. 1: Methodology Overview

We explored two types of code representations. The first uses software metrics, where numerical features were scaled using StandardScaler. The second treats code-as-text, which was transformed using three vectorization techniques: CountVectorizer, TF-IDF, and Word2Vec.

TABLE I. DISTRIBUTION OF CODE SAMPLES AFTER BALANCING

Dataset		Vulnerable	Clean	Total
Array Usage	Imbalanced	10,926	31,303	42,229
Array Usage	Balanced	10,926	10,926	21,852
Arithmetic Expression	Imbalanced	3,475	18,679	22,154
Arithmetic Expression	Balanced	3,475	3,475	6,950

Four models were trained for each configuration: RF and KNN as traditional ML models, and MLP and LSTM as DL models. These were evaluated across all combinations of dataset type, representation, vectorization method (for text), and class balance. Model performance was evaluated using standard classification metrics, including accuracy, precision, recall, false positive rate, false negative rate, F1-score, and ROC AUC. All results were saved to an Excel file for comparison and further analysis.

A. Dataset

This study utilizes two primary feature types for vulnerability detection: (i) the source code snippets themselves, and (ii) software metrics computed for those specific snippets. We used a publicly available dataset curated as part of the SySeVR framework [10], which contains labeled C/C++ function-level code snippets annotated for vulnerability de-tection. The dataset includes code slices across various vulnerability categories, such as Array Usage, Arithmetic Expression, Pointer Usage, and API Function Call. For this work, we focused on the Array Usage and Arithmetic Ex-pression subsets, which correspond to common vulnerability types involving improper array indexing and unsafe arith-metic operations. Each snippet is labeled as either vulnerable or non-vulnerable, supporting a binary classification task.

In addition, we used the slice-based software metrics dataset introduced in [4] and available via [11]. This dataset contains precomputed software metrics for the same function-level code snippets used in SySeVR. For each snippet, 18 metrics are provided, capturing characteristics such as size, structure, complexity, and defect likelihood, which are valuable indicators for machine learning-based vulnerability detection.

In this study, the full set of metrics used to represent each code snippet is listed below, which are directly sourced from the software metrics dataset. The last eight are to capture additional aspects of code complexity, including Halstead's metrics and McCabe's cyclomatic complexity. Together, these metrics offer a structured, interpretable representation of code that reflects both its syntactic and cognitive complexity.

- *Empty_Lines*: Number of blank lines
- *Lines_Of_Comments*: Total number of comment lines
- *Lines_Of_Program*: Number of actual code lines (excluding comments/blanks)
- *Physic_Line*: Total number of lines including all content
- *n1_Number_Of_Distinct_Operators*: Number of unique operators in the code

Authorized licensed use limited to: ULAKBIM UASL - MIDDLE EAST TECHNICAL UNIVERSITY. Downloaded on November 07,2026 at 21:38:32 UTC from IEEE Xplore. Restrictions apply.

- *n2_Number_Of_Distinct_Operands*: Number of unique operands in the code
- *n_Program_Vocabulary*: Total number of unique operators and operands
- *N1_Total_Number_Of_Operators*: Total occurrences of operators
- *N2_Total_Number_Of_Operands*: Total occurrences of operands
- *N_Program_Length*: Total occurrences of operators and operands
- *B_Number_of_Delivered_Bugs_1*: Estimated number of delivered bugs (method 1)
- *B_Number_of_Delivered_Bugs_2*: Estimated number of delivered bugs (method 2)
- *D_Difficulty*: Difficulty of understanding the code
- *E_Effort*: Estimated effort required to write or understand the code
- *T_Time_Required_To_Program*: Estimated time to program based on effort
- *V_Volume*: Volume or size of the implementation
- *_N_Calculated_Program_Length*: Expected length of program (theoretical)
- *McCab_Number*: McCabe cyclomatic complexity (number of decision paths)

B. Hyperparameters

To keep the experiments consistent and fair, all models were trained with fixed settings. For traditional ML models, RF used 1000 trees and a random seed of 100, while KNN used 3 neighbors. These choices helped ensure stable and comparable results. The deep learning models, MLP and LSTM, were trained for 25 epochs with a batch size of 64. Both used the same architecture with three hidden layers (24, 12, and 8 units) and ReLU activation, ending with a sigmoid layer for binary classification.

Models were trained using the Adam optimizer with a learning rate of 0.001, and binary cross-entropy as the loss function. A validation split of 20% was used to monitor training performance, and early stopping was employed to prevent overfitting. Additionally, data shuffling was enabled during training to improve model generalization. Word2Vec was trained using the skip-gram architecture with a vector size of 100, window size of 5, min count=1, and a fixed random seed (42). Each code snippet was represented by the average of its token embeddings, or a zero vector if no tokens were in the vocabulary.

C. Experiment Execution

1) Experimental Setup

This study used both balanced and imbalanced versions of two datasets: Array Usage and Arithmetic Expression, resulting in four dataset variations. Balanced versions were created by randomly undersampling the majority class. Two types of code representation were used: software metrics and raw code text. For software metrics, precomputed numerical features were standardized using *StandardScaler* to ensure uniform scaling. For code-as-text representations, three vectorization techniques were applied, namely CountVectorizer, TF-IDF, and Word2Vec. Each model was trained using the same hyperparameters to ensure fair comparison. Combining the four datasets, four models, and four input types (three text-based vectorizations and one metric-based representation) resulted in a total of 64 experiments ($4 \times 4 \times 4 = 64$), each evaluating a unique configuration of dataset, representation, and model.

2) Experiment Execution

All experiments were implemented using Python 3.11. ML models (RF and KNN) were implemented using the scikit-learn library (v1.7), which was also used for applying CountVectorizer, TF-IDF via *sklearn.feature extraction.text*, and for feature scaling using *StandardScaler*. DL models (MLP and LSTM) were developed using the TensorFlow and Keras libraries, while Gensim was used to train Word2Vec embeddings on the code snippets. The pandas library (v2.3) was employed throughout the process for data manipulation, result aggregation, and exporting outputs to Excel.

Each experiment used 80% training and 20% testing split, with fixed random seeds to ensure consistent preprocessing and model initialization. All experiments were conducted on Google Colab Pro, using a T4 GPU with 15 GB of memory, 12.7 GB of RAM, and 235 GB of available disk space.

V. Results and Discussion

This section presents the experimental results obtained from all model configurations and discusses them in relation to the research questions. Table II summarizes the F1 scores across different combinations of model type, code representation (software metrics or source code-as-text), and vectorization technique. Results are reported for four dataset variants: imbalanced array usage (IAU), balanced array usage (BAU), imbalanced arithmetic expression (IAE), and balanced arith-metic expression (BAE). The F1 score is used as the primary evaluation metric because it balances precision and recall. This is particularly important in software vulnerability de-tection, where both false positives and false negatives can have serious consequences.

TABLE II. F1 SCORE COMPARISON ACROSS MODELS, CODE REPRESENTATIONS, VECTORIZATION TECHNIQUES, AND DATASETS

Model	Rep.	Vectorizer	IAU	BAU	IAE	BAE
RF	Metric	-	0.774	0.8401	0.8283	0.8725
RF	Code	Count	0.7244	0.8662	0.8241	0.9187
RF	Code	TF-IDF	0.7015	0.8517	0.7957	0.9050
RF	Code	Word2Vec	0.7356	0.8569	0.8176	0.9074
KNN	Metric	-	0.735	0.8053	0.7796	0.8495
KNN	Code	Count	0.6815	0.8339	0.7200	0.8802
KNN	Code	TF-IDF	0.6679	0.8236	0.7141	0.8705
KNN	Code	Word2Vec	0.6659	0.8166	0.7386	0.8707
MLP	Metric	-	0.5166	0.7400	0.5140	0.7865
MLP	Code	Count	0.7790	**0.8782**	0.8506	**0.9299**
MLP	Code	TF-IDF	0.7514	0.8453	0.8504	0.9060
MLP	Code	Word2Vec	0.7091	0.7967	0.7305	0.8503
LSTM	Metric	-	0.4935	0.7327	0.5174	0.7866
LSTM	Code	Count	0.7834	0.8761	**0.8567**	0.9123
LSTM	Code	TF-IDF	0.7375	0.8428	0.8073	0.9096
LSTM	Code	Word2Vec	**0.8002**	0.8669	0.6751	0.8077

Authorized licensed use limited to: ULAKBIM UASL - MIDDLE EAST TECHNICAL UNIVERSITY. Downloaded on November 07,2026 at 21:38:32 UTC from IEEE Xplore. Restrictions apply.

A. Answers to Research Questions

RQ1: *How does the representation of source code, either as software metrics or as code-as-text, affect the performance of traditional machine learning and deep learning models in vulnerability detection?*

Using code-as-text consistently yields better performance for DL models compared to software metrics. This trend holds across all datasets and vectorization methods. For instance, the MLP model achieves an F1 score of *0.9299* on the *Balanced Arithmetic Expression* dataset using code-as-text with CountVectorizer, compared to *0.7865* with software metrics. Similarly, on the *Imbalanced Array Usage* dataset, LSTM reaches an F1 score of *0.7834* with CountVectorizer, while scoring only *0.4935* using software metrics. These results suggest that textual representations of code provide richer sequential and contextual information, which deep learning models are better equipped to capture and learn from.

In contrast, for traditional ML models, the effectiveness of code representation appears to be more influenced by dataset balance. Under imbalanced conditions, software metrics generally produce better results. For instance, on the *Imbalanced Arithmetic Expression* dataset, RF achieves an F1 score of *0.8283* using software metrics, slightly outperforming the *0.8241* it achieves with code text via CountVectorizer. KNN also performs marginally better with software metrics in the imbalanced setting. However, when the datasets are balanced, code-as-text tends to outperform metrics for both RF and KNN.

RQ2: *How do vectorization techniques (CountVectorizer, TF-IDF, Word2Vec) influence the performance of machine learning and deep learning models when using code-as-text?*

Among the vectorization techniques, CountVectorizer provides the most consistent and superior performance across most datasets. For instance, the MLP model achieves an F1 score of *0.9299* on the *Balanced Arithmetic Expression* dataset using CountVectorizer, outperforming both TF-IDF (*0.9060*) and Word2Vec (*0.8503*). This trend also holds for the *Balanced Array Usage* and *Imbalanced Arithmetic Expression* datasets, where CountVectorizer consistently enables top-performing results across different models. These findings suggest that raw frequency-based representations are generally more effective than TF-IDF weighting or pretrained embeddings for source code in vulnerability detection.

However, an exception to this trend is observed in the *Imbalanced Array Usage* dataset. In this case, the LSTM model combined with Word2Vec achieves the highest F1 score of *0.8002*, outperforming all other model and vectorizer combinations for that dataset, including those using CountVectorizer. This result highlights the effectiveness of sequential models like LSTM, especially when paired with Word2Vec embeddings, in capturing contextual and semantic patterns in source code, particularly in scenarios involving imbalanced data distributions. It also suggests that while CountVectorizer is robust and consistently strong across most settings, specific dataset characteristics may favor the deeper contextual understanding provided by Word2Vec and LSTM architectures.

RQ3: *How significantly does class imbalance affect model performance across different code representations and vectorization techniques?*

Class imbalance negatively impacts F1 scores in all models, code representations, and vectorization methods. Across the board, results on balanced datasets consistently outperform their imbalanced counterparts. For example, RF with code and CountVectorizer achieves an F1 score of *0.9187* on the *Balanced Arithmetic Expression* dataset, while it drops to *0.8241* on the imbalanced version. This highlights the importance of dataset balancing to improve the reliability and fairness of vulnerability detection models.

RQ4: *Which model family, traditional ML or DL models, performs more effectively for different code representations?* When using software metrics as input features, traditional machine learning models consistently outperform deep learning models across all datasets. For example, Random Forest achieves the highest F1 scores on every dataset with metrics, while KNN also performs better than deep learning models like MLP and LSTM. In contrast, MLP and LSTM struggle with metric-based input, often showing significantly lower performance. This suggests that deep learning models are not well-suited for structured, low-dimensional data like software metrics. Instead, traditional models such as Random Forest and KNN are more effective at capturing patterns in this type of representation.

When using software metrics, traditional machine learning models such as RF perform better than DL models. For instance, on the *Balanced Arithmetic Expression* dataset, RF achieves an F1 score of *0.8725*, whereas MLP and LSTM reach *0.7865* and *0.7866*, respectively. However, when using code-as-text, DL models clearly outperform traditional ones. This suggests that while traditional models are more effective on structured metric data, DL models better leverage the semantic richness of textual code representations.

Deep learning models work best with code-as-text for all datasets, showing that this type of representation captures more useful information. For traditional machine learning models, software metrics give better results when the data is imbalanced, but code-as-text works better when the data is balanced. This means the best representation choice depends on both the model used and the balance of the dataset.

B. Threats to Validity

Several factors may influence the reliability and generaliz-ability of this study's findings. A primary limitation lies in the datasets used. While the selected C and C++ datasets include realistic examples of vulnerable code, they may not fully capture the diversity of coding styles and vulnerability types present in other programming languages such as Java, Python, or JavaScript. Consequently, the conclusions drawn here may not generalize to all languages or development environments.

The scope of the study was also limited to four models, RF,

Authorized licensed use limited to: ULAKBIM UASL - MIDDLE EAST TECHNICAL UNIVERSITY. Downloaded on November 07,2026 at 21:38:32 UTC from IEEE Xplore. Restrictions apply.

KNN, MLP, and LSTM, and three vectorization techniques: CountVectorizer, TF-IDF, and Word2Vec. More advanced architectures, such as transformer-based models or ensemble methods, were not explored and could provide additional insights or performance gains.

Finally, while fixed hyperparameters and random seeds were used to support reproducibility, some variability in results may still arise due to non-deterministic operations, particularly during GPU-based training. This is a common limitation in deep learning experiments and should be considered when interpreting the reported results.

VI. Conclusion

This study presents a detailed comparison of machine learning and deep learning methods for software vulnerability detection, using two types of code representations: software metrics and code-as-text. The experiments explored how different vectorization techniques (CountVectorizer, TF-IDF, and Word2Vec) affect model performance. The impact of class imbalance was also examined. Results show that the choice of representation and vectorization method significantly influences model effectiveness. Traditional machine learning models performed better with software metrics, whereas deep learning models achieved stronger results with textual code inputs. Additionally, all models showed improved performance on balanced datasets compared to imbalanced ones, underscoring the importance of class balance in designing datasets for vulnerability detection.

Our approach can be valuable in real-world software development by integrating into CI/CD pipelines to automatically detect vulnerabilities early, helping teams fix issues before production. Traditional ML models using software metrics work well when structured code data is available and quick, interpretable results are needed. DL models with raw code text perform better for complex patterns and evolving coding styles. By choosing the appropriate model and feature representation based on data conditions, teams can enhance static analysis tools with learning-based detection. This approach adapts to new coding styles and uncovers hidden or hard to detect vulnerabilities often missed by conventional methods.

This study can be extended in several directions. One promising direction is to combine software metrics and source code text to enable multi-modal learning, potentially allowing models to capture both structural and semantic characteristics of code more effectively. Another direction is to apply advanced transformer-based language models fine-tuned for code understanding, which could enhance performance for both text-based and metric-based representations. Incorporating explainability techniques would also improve the interpretability of model predictions and support real-world adoption. Finally, expanding the evaluation to include larger, more diverse datasets, especially those involving real-world Java, Python, or C# projects with labeled vulnerabilities, would strengthen the generalizability and robustness of the findings.

References

[1] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira, "Vulnerable Code Detection Using Software Metrics and Machine Learning," IEEE Access, vol. 8, pp. 219174–219198, Nov. 2020.

[2] K. Z. Sultana, V. Anu, and T.-Y. Chong, "Using software metrics for predicting vulnerable classes and methods in Java projects: A machine learning approach," Journal of Software: Evolution and Process, vol. 33, no. 3, p. e2303, Aug. 2021.

[3] B. Chernis and R. Verma, "Machine Learning Methods for Software Vulnerability Detection," in Proc. 4th ACM Int. Workshop on Security and Privacy Analytics, Mar. 2018, pp. 31–39.

[4] M. Zagane, M. K. Abdi, and M. Alenezi, "Deep Learning for Software Vulnerabilities Detection Using Code Metrics," IEEE Access, vol. 8, pp. 74562–74570, Apr. 2020.

[5] F. Subhan, X. Wu, L. Bo, X. Sun, and M. Rahman, "A deep learning-based approach for software vulnerability detection using code metrics," IET Software, vol. 16, no. 5, pp. 516–526, Jul. 2022.

[6] J. Yeboah and S. Popoola, "Efficacy of Static Analysis Tools for Software Defect Detection on Open-Source Projects," in Proc. 2023 Int. Conf. Computational Science and Computational Intelligence (CSCI), Mar. 2023, pp. 1588–1593.

[7] N. Tihanyi, T. Bisztray, M. A. Ferrag, B. Cherif, R. A. Dubniczky, R. Jain, and L. C. Cordeiro, "Vulnerability Detection: From Formal Verification to Large Language Models and Hybrid Approaches: A Comprehensive Overview," arXiv preprint arXiv:2503.10784, Jul. 2025. [Online]. Available: https://arxiv.org/abs/2503.10784

[8] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in Findings of the Association for Computational Linguistics: EMNLP 2020, Nov. 2020, pp. 1536–1547.

[9] 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. Clement, D. Drain, N. Sundaresan, J. Yin, and M. Zhou, "GraphCodeBERT: Pre-training Code Representations with Data Flow," in Proc. Int. Conf. Learning Representations (ICLR), Jan. 2021.

[10] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities," IEEE Trans. Dependable and Secure Computing, vol. 19, no. 4, pp. 2244–2258, Jan. 2022.

[11] M. Zagane, "Slice-Based Code Metrics Dataset," Jul. 2025. [Online]. Available: https://github.com/mzagane/slice-based_code_metrics_dataset

[12] Z. S. Harris, "Distributional Structure," WORD, vol. 10, no. 2-3, pp. 146–162, Mar. 1954.

[13] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," Inf. Process. Manag., vol. 24, no. 5, pp. 513–523, Aug. 1988.

[14] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," in Proc. Workshop at ICLR, Jan. 2013.

[15] X. Du, B. Chen, Y. Li, J. Guo, Y. Zhou, Y. Liu, and Y. Jiang, "LEOPARD: Identifying Vulnerable Code for Vulnerability Assessment through Program Metrics," in Proc. 41st Int. Conf. Software Engineering (ICSE), May 2019, pp. 60–71.

[16] T. Viszkok, P. Hegedüs, and R. Ferenc, "Improving Vulnerability Prediction of JavaScript Functions using Process Metrics," in Proc. 16th Int. Conf. Software Technologies (ICSOFT), Jan. 2021, pp. 185–195.

[17] A. Bagheri and P. Hegedűs, "A Comparison of Different Source Code Representation Methods for Vulnerability Prediction in Python," in Proc. Quality of Information and Communications Technology, Aug. 2021, pp. 267–281.

[18] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrer, and L. Grunske, "VUDENC: Vulnerability Detection with Deep Learning on a Natural Codebase for Python," Inf. Softw. Technol., vol. 144, p. 106809, Apr. 2022.

[19] I. Kalouptsoglou, M. Siavvas, D. Kehagias, A. Chatzigeorgiou, and A. Ampatzoglou, "Examining the Capacity of Text Mining and Software Metrics in Vulnerability Prediction," Entropy, vol. 24, p. 651, May 2022.

[20] M. H. Halstead, Elements of Software Science. Amsterdam: Elsevier North-Holland, 1977.

[21] T. J. McCabe, "A Complexity Measure," IEEE Trans. Softw. Eng., vol. SE-2, pp. 308–320, Dec. 1976.

Authorized licensed use limited to: ULAKBIM UASL - MIDDLE EAST TECHNICAL UNIVERSITY. Downloaded on November 07,2026 at 21:38:32 UTC from IEEE Xplore. Restrictions apply.