Journal of the Brazilian Computer Society, 2025, 31:1, doi: 10.5753/jbcs.2025.4495

Prediction of defects in Smart Contracts applying Deep Learning with Solidity metrics

Rogério de J. Oliveira [**Rio de Janeiro State University, IPRJ-UERJ** | ***rjoliveira@iprj.uerj.br***]
Edson M. Lucas [**Rio de Janeiro State University, IPRJ-UERJ** | ***emlucas@iprj.uerj.br***]
Gustavo Barbosa Libotte [**Rio de Janeiro State University, IPRJ-UERJ** | ***gustavolibotte@iprj.uerj.br***]

Polytechnic Institute, Rio de Janeiro State University, Rua Bonfim, 25, Vila Amélia, Nova Friburgo, RJ, 28625-570, Brazil.

Received: 14 May 2024 • **Accepted:** 04 December 2024 • **Published:** 26 February 2025

Abstract Smart Contracts are autonomous, self-executable programs that facilitate agreement execution without the need for intermediaries. These contracts are also susceptible to software defects, leading to vulnerabilities that can be exploited by attackers. The use of models for predicting software defects is a well-studied research area. However, applying these models with Smart Contract metrics is an area that remains underexplored. The aim of this study is to evaluate whether deep learning models used in the prediction of traditional software defects produce equivalent results with specific Smart Contract metrics. Machine learning models were applied to four data sets, and performances were evaluated using Precision, Recall, F-score, Area under the curve (AUC), Precision-recall curve (PRC), and Matthews Correlation Coefficient (MCC). This approach complements traditional formal verification methods, which, although accurate, are often slower and less adaptable to emerging vulnerabilities. By employing deep learning, the model enables faster and more cost-effective analysis of large volumes of Smart Contracts. Unlike conventional techniques that rely on expert-defined rules and require substantial computational resources, this model offers scalable and continuous monitoring. Consequently, the research provides a complementary solution that can significantly enhance the security of the smart contract ecosystem, allowing for the detection of potential defects before exploitation occurs.

Keywords: Smart Contracts, Software Defects Prediction, Code Metrics, Machine Learning, Blockchain

1 Introduction

Softwares are embedded in many areas of human activity, causing a significant impact on our way of living. A software is an intangible product built by people through a process aiming to function as designed without defects. A software defect is an error, vulnerability, or deviation in a software product or in the software development process [Ozakinci and Tarhan, 2016]. Defects can lead to excessive consumption of project resources, including time, cost, and effort [Saifan and Abu-wardih, 2020]. In this scenario, various studies have sought to prevent or identify these defects in the earliest stages of software development, including planning, coding, and preliminary testing, to minimize the impact on project time, cost, and effort. Software defect prediction techniques emerge as a response to this challenge, aiming to classify defective parts of a software system before its release [Chang *et al*., 2011]. Software metrics serve as an indirect indicator of the likelihood of software defects, thus affecting its quality [Wang *et al*., 2012]. Object-oriented (OO) software metrics, for example, are based on elements such as Depth of Inheritance Tree (DIT) and Lines of Code (LOC) [Fenton and Bieman, 2014].

Smart contracts are computer programs that implement and execute transactions and manage business logic in software developed for blockchain, which consists of a decentralized and distributed data structure, characterized by its security and transparency due to the chained blocks that record transactions immutably on multiple computers [Bhargavan *et al*., 2016; Beck *et al*., 2017; Treiblmaier, 2020]. One of the main platforms for deploying these smart contracts is the public Ethereum Blockchain [Osterland and Rose, 2020]. These contracts are commonly programmed using the Solidity language [Pierro and Rocha, 2019]. This language has specific metrics to quantify various aspects of smart contract code, which can be used by developers to improve the quality of the produced code [Pierro and Tonelli, 2020]. They introduce a new dimension for the application of these metrics. In addition to object-oriented software metrics, specific metrics for smart contracts and blockchain can be employed to identify potentially defective software. As pointed out by Liu *et al*. [2023], existing contract security analyses heavily depend on rigid rules defined in advance by experts, a laborious and non-scalable process. Software testing and Machine Learning (ML) are two major research areas whose intersection has attracted researchers' attention [Durelli *et al*., 2019].

Software testing is an essential activity aimed at identifying and addressing vulnerabilities and other types of errors in software. This process helps ensure that the software functions correctly and securely, minimizing the risk of execution deviations from specifications or security breaches. ML, a subarea of Artificial Intelligence, consists of models that enable iterative learning from data, taking into account a specified architecture and reward functions. The learning of these models can be broadly categorized into three main types: supervised, unsupervised, and reinforcement learning [Ghaffarian and Shahriari, 2017; Durelli *et al*., 2019; Mirjalili *et al*., 2020; Jiang *et al*., 2023]. ML models have been increasingly

used in software defect prediction. Considering that software testing costs are substantial, even small improvements in the ability to identify and correct defects can result in significant savings [Bowes *et al*., 2018].

Attention to smart contracts has increased as blockchain technology advances, raising concerns related to their security [Jiang *et al*., 2023]. The Solidity language, widely used for executing smart contracts on the Ethereum Blockchain, presents specific challenges in detecting vulnerabilities, which are particular types of defects. These vulnerabilities can be maliciously exploited and are critical to the security of contracts [Bhargavan *et al*., 2016; Pierro and Tonelli, 2020; Liu *et al*., 2023]. Existing tools depend on rigid logical rules defined in advance by experts, and the time to perform detection significantly increases with the complexity of the smart contract [Zhang *et al*., 2022]. Despite recent advances in automated analysis, testing, and debugging of Ethereum smart contracts, according to Durieux *et al*. [2020], the complexity of implementation and the difficulty of accessing data and qualified professionals to label defects found in smart contracts represent obstacles to the comparison and reproduction of these researches. To the best of our knowledge, there is no Solidity dataset with metrics and labelled vulnerabilities, nor any other work using Solidity software metrics in predicting defects in smart contracts.

This work aims to investigate the effectiveness of machine learning models, specifically applied with traditional object-oriented software metrics (OO) and specific metrics of the Solidity language, in predicting defects in smart contracts. To evaluate the application of smart contract metrics in defect prediction, we implemented a Deep Learning-based model that we named Deep Neural Network — Ethereum Solidity Metrics (DNN-ESM). To build our labelled Solidity software metrics dataset with defects, we combined smart contract datasets from the work of Pierro and Tonelli [2020], containing metrics, with defect labels produced by Yashavant *et al*. [2022]. Thus, we created a new dataset to evaluate smart contracts.

The primary contribution of this research is the proposal of a defect prediction model for smart contracts based on specific metrics of the Solidity language. This approach complements traditional formal verification methods which, although precise, tend to be slower and less adaptable to new vulnerabilities. The proposed model utilizes deep learning to efficiently predict defects, enabling faster and more cost-effective analysis of a substantial number of smart contracts deployed on the blockchain. Unlike conventional techniques that rely on expert-defined rules and require significant computational effort, this Solidity metrics-based approach enables continuous and expansive monitoring that can be applied on a large scale. Consequently, this research offers a complementary solution that can significantly enhance the security of the smart contract ecosystem, allowing for early detection of potential defects before they are exploited.

The remainder of this article is organized as follows. Section 2 presents fundamental concepts about smart contracts and ML models applied in software defect prediction. Section 3 describes the experiment planning. Section 4 presents the execution of the experiment, and Section 5 discusses the results. Section 6 summarizes threats to validity. Section 7 summarizes related work on software defect prediction and vulnerability detection in smart contracts. Finally, Section 8 consolidates the conclusions and outlines future research directions.

2 Background

This section briefly introduces the concepts of blockchain-oriented software and ML models. The discussion is divided into two parts: the first provides the essential foundations of blockchain technology and smart contracts; and the second demonstrates the application of ML models in software defect prediction.

ML models have been gaining ground in software defect prediction. Since software testing costs are very high, even small improvements in our ability to find and fix defects can make a significant difference in overall costs [Bowes *et al*., 2018]. A software defect prediction model typically relies on ML techniques, and supervised learning consists of two essential phases: Training Phase and Validation Phase.

In the Training Phase, the model is constructed from a combination of training instances. These instances come from historical software datasets, labelled by a flag representing their classification, such as clean or defective, where the independent variables predict dependent variables, given the values of input attributes. Feature values can be numerical, nominal or categorical, or ordinal [Qiao *et al*., 2020]. In this context, software metrics represent the characteristics or attributes of the generated instances.

In the Validation Phase, after the model has been built, new unknown instances are provided to validate the produced model [Saifan and Abu-wardih, 2020]. Various ML models have been experimented with in software defect prediction using supervised learning, and in diverse strategies, where binary classification is one of the most used [Chicco and Jurman, 2023]. Solidity is the most used language for smart contracts executed on the Ethereum Blockchain [Bhargavan *et al*., 2016; Pierro and Tonelli, 2020; Liu *et al*., 2023]. Existing smart contract vulnerability detection tools depend on rigid logical rules defined in advance by experts, and the time to execute detection significantly increases as the complexity of the smart contract increases [Zhang *et al*., 2022].

2.1 Smart contracts

Smart contracts are computer programs that implement and execute transactions and manage business logic on a blockchain. These contracts can be written in various programming languages and for different blockchains. However, currently, the most used language for smart contracts is Solidity, and the most used platform is the Ethereum Blockchain [Bhargavan *et al*., 2016; Pierro and Tonelli, 2020; Liu *et al*., 2023].

The Ethereum Blockchain is a public ledger that keeps records of all transactions and executes software code implementing smart contracts [Badruddoja *et al*., 2021]. In Ethereum, smart contracts are developed and operationalized on the Ethereum Virtual Machine (EVM), a robust environment that not only facilitates the creation and execution of

these contracts and DApps (decentralized applications) but also allows for the writing and autonomous execution of complex code instructions. Users deploy smart contracts by publishing the bytecode of the smart contract on the blockchain [Zhang *et al*., 2020].

To enhance the development and implementation of smart contracts, development (Dev) and testing (Testnet) environments are crucial. They provide a separate space from the main network (Mainnet) for developers to write, test, and refine their contracts before deploying them on the main network. This is important to mitigate risks, given that any error in a smart contract becomes immutable after its implementation on the blockchain. While read operations are free, it is important to note that write tasks in smart contracts on Ethereum consume gas, a fee charged to execute operations on the network.

Effective smart contract modelling, however, faces significant obstacles. The complexity of the Solidity language, the scarcity of supporting tools, infrastructural limitations, and the lack of clear information on interface patterns and implementation specifications are barriers that developers encounter.

The immutability of the blockchain presents an additional challenge, as correcting errors in already implemented code becomes a daunting task. Unlike traditional software development, which allows for iterations in various phases, smart contract development requires all iterations and testing to be completed before the implementation phase due to the immutable nature of these contracts.

Programming in the context of smart contracts introduces its own constraints, such as size limitations and complexities in interaction between different contracts. These characteristics demand a meticulous approach to ensure that contracts meet the requirements and policies established by stakeholders. In the EVM environment, error detection can be challenging, and even when available, compliance with standards, although crucial, does not guarantee that specifications are fully met, leaving room for vulnerabilities in contract interactions [Velasco *et al*., 2023].

Given the complexity of software development in the blockchain context, a new development paradigm called Blockchain-Oriented Software (BOS) has been created [Ortu *et al*., 2019]. BOS is any software that operates with the implementation of a blockchain. Software developers can apply specific BOS software engineering practices considering the characteristics of a blockchain. Such practices constitute the basis of Blockchain-Oriented Software Engineering (BOSE). This software architecture with its specific design notations, macro architecture patterns, or metamodels can be defined for the development of BOS. Due to the distributed nature of the blockchain, specific metrics are needed to measure the complexity, communication capability, resource consumption, and overall performance of BOS systems. Thus, BOSE can benefit from the introduction of specific metrics.

To enhance system security and reliability in the context of critical security systems, such as BOS, it is crucial to apply adequate testing techniques tailored to the nature of the application. Specifically, test suites need to be implemented for BOS. Furthermore, facilitating the development of smart contracts is an ongoing challenge. The authors highlight the need for software engineers to develop specialized tools and techniques for blockchain-oriented software architecture, development, and testing [Porru *et al*., 2017].

Solidity is the primary programming language for Smart Contracts, initially released in May 2015. Since then, its constant evolution, characterized by various version updates and bug fixes, has responded to emerging needs and specific practices of blockchain-oriented programming, which in turn can impact software metrics. These smart contracts, applied in various industries with high economic impact, reinforce the need for code analysis tools to verify compliance with Solidity coding rules [Pierro and Rocha, 2019]. Additionally, detecting vulnerabilities in smart contracts is a key element for blockchain security [Liu *et al*., 2023].

The vulnerabilities addressed in this work include ARTHM — Arithmetic Errors (Overflow and Underflow), LE — Stuck Ether, RENT — Reentrancy, and TimeO — Transaction Order Dependency. These vulnerabilities are discussed in several studies, including [Chen *et al*., 2020], [Huang *et al*., 2022], [Sui *et al*., 2023], [Yashavant *et al*., 2022], and [Zhang and Liu, 2022]. Before detailing each vulnerability, it is important to understand the role of Ether, a fundamental digital asset within the Ethereum Blockchain. Ether is used to pay transaction fees, execute smart contracts, participate in network consensus, and purchase goods and services on the Ethereum network. This digital asset is crucial in the processes that may be affected by the following vulnerabilities:

ARTHM Related to integer arithmetic errors, which can occur when a smart contract attempts to perform an arithmetic operation and a value is assigned to a variable that is greater or lesser than the representable numerical limit.

LE Refers to the situation where a smart contract has a correct deposit function, but its withdrawal function is defective or absent, resulting in funds being trapped in the contract. This retention may be the result of programming errors or malicious coding.

RENT Allows a function to be called again before the completion of the first invocation. This action can be executed repeatedly and may continue until all funds from the contract are drained.

TimeO Occurs when the contract logic depends on the order in which transactions are executed, introducing vulnerabilities associated with transaction mining and ordering.

2.2 ML for software defect prediction

Machine Learning is a sub-branch of Artificial Intelligence (AI) that focuses on the aspect of computer learning. It includes methods that enable a program to learn from experience. ML algorithms and techniques are often used in the data mining process for preprocessing, pattern recognition, and generating prediction models [Ghaffarian and Shahriari, 2017]. These methods are successfully employed in predicting software defects.

Software defect prediction tasks can be divided into within-project defect prediction (WPDP) and cross-project defect prediction (CPDP). In WPDP, modules from the old version of a program are used to build the defect prediction model, which is then used to predict defective modules in the

new version of the same software project. However, CPDP relies on data from different projects. In CPDP, the prediction model is built based on data from one or more projects and used to predict defects in a different project. This approach can be useful when there is not enough historical data available within the project itself to train a model [Liang *et al*., 2019].

It is important to differentiate between software vulnerability detection and prediction. Vulnerability detection typically requires access to the source code or byte codes of the software, performed using static or dynamic code analysis techniques. On the other hand, software vulnerability prediction is a more proactive approach aimed at anticipating and forecasting potential vulnerabilities in software, allowing preventive measures to be taken to mitigate them before they occur. Vulnerability prediction models are built based on historical data of past vulnerabilities, information about software characteristics, libraries used, and programming practices, among other factors. This involves using data analysis and ML techniques to identify patterns, trends, and risk factors that may lead to the emergence of vulnerabilities.

The IEEE Standard Classification for Software Anomalies provides the main set of attributes for classifying faults and defects in software. In this document, error, failure, defect, issue, and bug are uniformly described as anomalies. This standard aims to define a common vocabulary with which different people and organizations can effectively communicate about software anomalies and establish a common set of attributes that support industry techniques for analyzing software defect and failure data. This standard applies to any software and any phase of the software lifecycle [ISDW, 2010]. It is important to note that this standard has not been revised for over 10 years and currently holds the status 'Inactive — Reserved,', indicating that while it is still available for use, it is not actively maintained or updated.

One of the most promising technologies used in software defect detection projects is neural networks. A neural network is a type of machine-learning system inspired by the human brain, where a collection of interconnected nodes can perform complex tasks by learning from data. These nodes are simple computational units that can process information. The nodes are interconnected, and the connections are weighted. The weight of a connection determines the strength of the connection between two nodes [O'Shea and Nash, 2015].

Deep Learning is a sub-branch of ML, which consists of a family of algorithms, including, among others, Recurrent Neural Networks (RNN), Deep Neural Networks (DNN), Convolutional Neural Networks (CNN), and Long Short-Term Memory (LSTM). This field acts as a subcategory of Machine Learning, using neural networks to model and decipher complex data.

DNN is an improved version of the traditional artificial neural network with multiple dense layers. DNN models are recently becoming very popular due to their excellent performance in learning, not only the non-linear mapping from input to output, but also the underlying structure of input data vectors [Alghanim *et al*., 2022]. However, according to Jiang *et al*. [2023], comprehensive reviews on the application of ML in smart contract security detection are relatively rare.

According to Zeng *et al*. [2020], a significant challenge in vulnerability detection with Deep Learning is the lack of datasets. Many available datasets cannot be directly applied in training Deep Learning models due to the need for data preprocessing. Especially for Deep Learning-based methods, a lot of training data is required to achieve excellent performance. Furthermore, there is no publicly available labelled dataset with samples of various vulnerabilities, and the data collection process often requires experts to laboriously label the code. And this manual process of collecting software vulnerabilities can be expensive.

The learning of the ML models can be broadly categorized into three main types: supervised, unsupervised, and reinforcement [Ghaffarian and Shahriari, 2017; Durelli *et al*., 2019; Mirjalili *et al*., 2020; Jiang *et al*., 2023]. In the following paragraphs, these three types of learning are briefly presented.

- ***Supervised Learning*** is used when, for each set of input variables, there is a corresponding output variable. In this scenario, an algorithm is used to learn the input-output mapping function, aiming to predict outputs for future inputs or better understand the relationship between input and output. Supervised algorithms learn by generalizing from known examples. These algorithms find ways to produce the desired output based on the input-output pairs provided by the user [Durelli *et al*., 2019]. There is a supervisor to provide the learning algorithm with insights into how good or bad an action or decision is. In supervised learning methods, the dataset is fully populated, and the learning method can check if a particular action is correct or incorrect [Mirjalili *et al*., 2020].
- ***Unsupervised Learning*** is used in situations where labelled training data is not available (input-output mapping). The goal of the learning system is to identify patterns and structures in the provided dataset [Ghaffarian and Shahriari, 2017]. The algorithm itself is responsible for finding the labels and defining them. These learning algorithms need to learn the structure of the dataset and the relationship between the features [Mirjalili *et al*., 2020]. Unsupervised learning is typically associated with clustering problems, where the objective is to determine if inputs fall into distinct groups [James *et al*., 2013]. When only a subset of input data has associated output data, the problem becomes a blend of supervised and unsupervised learning, often referred to as a semi-supervised problem [Durelli *et al*., 2019]. In this setting, algorithms must incorporate into the analysis the input data for which associated output data is available, as well as the input data for which there are no corresponding output data [James *et al*., 2013]. According to Durelli *et al*. [2019], the vast majority of software testing problems have been formulated and addressed as supervised learning problems. Semi-supervised algorithms are more frequently used than unsupervised algorithms.
- ***Reinforcement Learning*** is trained to achieve a specific

goal. If the action taken is correct, the algorithm receives a reward; if it is incorrect, it receives a penalty. This type of learning mimics the learning process of living beings, based on a system of rewards and punishments [Ghaffarian and Shahriari, 2017; Mirjalili *et al*., 2020].

This work focuses on supervised learning methods. In the next section, we will present the experiment planning with a DNN applied to defect prediction in smart contracts.

3 Design of Experiment

We followed the methodology proposed by Tong *et al*. [2018] to predict defects in smart contracts. In general, we first prepared the data; then we chose the machine learning models, and finally, we measured the performance. We focused on the following research questions (RQs):

RQ1: Are machine learning models applied with traditional OO software metrics also effective for defect prediction in smart contracts using only Solidity metrics?

RQ2: Does the defect prediction method DNN-ESM applied with Solidity metrics perform similarly to current methods for detecting defects in smart contracts?

Next, we detail the elements associated with the methodology proposed in this experiment planning.

3.1 Data preparation

The data preparation stage can be divided into two subtasks: data collection and preprocessing, which are detailed below.

A good dataset is essential to ensure effectiveness in predicting software defects. As highlighted by Pachouly *et al*. [2022], the quality of the dataset significantly impacts the performance of defect prediction models, with well-curated and validated datasets leading to more reliable and accurate predictions. In this study, we used datasets containing only traditional OO metrics and datasets exclusively with Solidity metrics. The dataset with traditional OO metrics was obtained from the PROMISE package, while the Solidity datasets were constructed based on the works of Pierro and Tonelli [2020] and Yashavant *et al*. [2022]. To evaluate the performance of our DNN-ESM model, we initially relied on the PROMISE KC3 dataset and later on specific datasets from the Ethereum Blockchain: ARTHM, LE, RENT, and TimeO. These latter datasets were composed of Solidity smart contract metrics with and without known defects (see Table 1).

The data underwent a preprocessing phase, including checking for missing data, duplicate instances, and converting categorical data into numerical. The data were then normalized using the Standard Scaler from Scikit-learn, while the Label Encoder was used for binary labels. The Standard Scaler was applied to standardize the numerical features, ensuring a mean of 0 and a standard deviation of 1 for each, while the Label Encoder transformed categories into numerical values, allowing ML algorithms to efficiently interpret both numerical and categorical attributes.

Table 1. Solidity Metrics.

Name	Description
Payable	Number of Payable Functions.
Mappings	Number of Mapping Types.
Modifiers	Number of Function Modifiers.
Addresses	Number of Addresses.
Events	Number of Events.
Contracts	Number of Contracts.
ABI	Size of the ABI (Application Binary Interface).
Bytecode	Size of the Bytecode.

3.2 Model selection criteria

Figure 1 shows a scheme of the DNN-ESM model architecture. The input layer receives the metrics extracted from the smart contracts. This layer accepts a feature vector of a size equal to the number of attributes in the training set. Next, the network passes through multiple dense hidden layers. The first dense layer contains 128 neurons with L2 regularization, which penalizes excessively large weights (regularization coefficient of 0.01), followed by a LeakyReLU activation (alpha = 0.2) and a dropout of 50% to enhance the model's robustness. Subsequently, the second dense layer, with 256 neurons and L2 regularization (coefficient of 0.01), is applied along with the LeakyReLU activation and batch normalization to stabilize the training. A second dropout of 50% is used to prevent overfitting. The model also includes an autoencoder block, starting with a layer of 128 neurons with ReLU activation to reduce dimensionality and extract relevant features, followed by an encoding layer with 64 neurons, also with ReLU activation. After the encoder, the reconstruction of the inputs is performed by a dense layer of 128 neurons with ReLU activation, followed by a layer that restores the original input dimension. The output layer uses a softmax activation function, with the number of neurons corresponding to the two categories of the output variable (clean and defective). The model's structure was optimized through a grid search for selecting the best hyperparameters [Ng, 2004; Ioffe and Szegedy, 2015; Xu *et al*., 2020; Gao *et al*., 2020].

Our DNN-ESM was compared with popular methods, Logistic Regression, Naive Bayes, Support Vector Machine (SVM), K-Nearest Neighbors (KNN), Decision Tree, Random Forest, and XGBoost, using the KC3 dataset. Subsequently, the DNN-ESM was trained using the Solidity smart contracts datasets. We utilized the Python programming language with libraries such as scikeras, numpy, pandas, sklearn, and matplotlib.

The model training was performed using the Adam optimizer with an initial learning rate of 0.001 and a categorical cross-entropy loss function [Kingma and Ba, 2017; Rezaei-Dastjerdehei *et al*., 2020]. To prevent overfitting, we implemented the Early Stopping technique, which monitors the validation loss and stops the training when no improvement is observed for 10 consecutive epochs. Notably, Early Stopping was frequently triggered, terminating the training around the 50th epoch in most cases. Additionally, we employed the ReduceLROnPlateau method to dynamically adjust the learning rate. This method reduces the learning rate by a factor of 0.2 when no improvement in validation loss

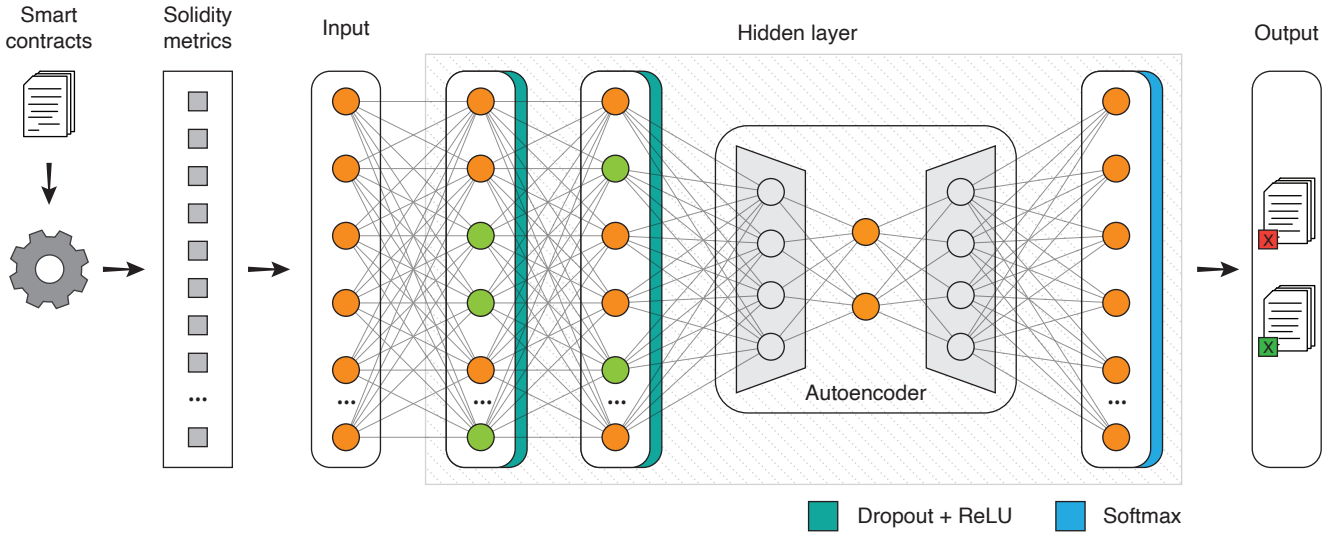

Figure 1. The DNN-ESM model utilizes a deep neural network with Solidity metrics to predict defects in smart contracts. First, the Solidity metrics are extracted from the smart contracts to build the neural network input. These inputs are processed by the network's initial dense layers applying initial transformations. Subsequently, multiple hidden layers of nodes (circles) and connections (lines) perform non-linear transformations using the ReLU activation function and employ Dropout regularization techniques to prevent overfitting. At the center, an autoencoder block is represented by the nodes connected in a "funnel" shape to perform unsupervised learning and dimensionality reduction. Finally, the last layer, represented by the nodes on the right, uses the Softmax activation function to perform binary classification and predicts the presence or absence of defects in the smart contracts.

is seen after 5 epochs, with a lower limit of 0.0001. The training process was configured with a batch size of 32 and a maximum limit of 200 epochs, although this limit was rarely reached due to the effectiveness of Early Stopping [T R *et al.*, 2024]. The dataset was initially divided into two parts: 80% of the data was allocated for model construction, and 20% was reserved for final testing. Within the portion designated for model construction, we performed a further subdivision: 80% of the data was used for training, and 20% for validation. This subdivision allowed for the recurrent use of training and validation data to fine-tune the model until it achieved satisfactory performance. Only after this stage were the 20% of data reserved for final testing used, providing an objective evaluation of the model's performance. When fine-tuning the DNN-ESM model for each dataset, we adjusted different functions and configuration parameters, such as the optimizer, the number of epochs, and the loss rate. Parameters were selected based on the model's accuracy.

During the execution of the experiment, we monitored the loss and accuracy metrics in both the training and validation sets throughout the epochs, and the results obtained are shown in Figure 2.

3.3 Evaluation metrics

Selecting appropriate evaluation metrics is a critical element in the development and validation of machine learning models. Despite its importance, there is no consensus in the literature on a single ideal evaluation metric in all contexts [Chicco and Jurman, 2020]. In the literature, there are multiple evaluation metrics, with the main metrics being Precision, Recall, F-score, and Area Under the Curve (AUC) [Bahaa *et al.*, 2021]. These metrics provide a comprehensive view of the model's performance, allowing for detailed comparisons and the identification of areas that require optimization.

The Accuracy metric assesses the proportion of correct predictions relative to the total predictions. Precision estimates the percentage of correct predictions among all classified as positives. Recall, or Sensitivity, indicates the proportion of true positives correctly identified relative to the total true positive cases. A model with high Recall can recognize most positive cases in the data, although it may misclassify some negatives as positives. The F1-Score, derived from Precision and Recall, allows identifying areas where the model did not perform well, guiding potential adjustments.

The AUC and PRC metrics provide information about the model's performance concerning the Receiver Operating Characteristics (ROC) curve and the Precision-Recall curve, respectively, being essential for evaluating models in scenarios with imbalanced datasets. The MCC is often applied to assess the quality of binary classification models, especially when there is data imbalance [Matthews, 1975]. An MCC close to +1 indicates superior model performance. Analyzing various metrics allows for discerning nuances in the models' performance.

To evaluate the performance of our defect prediction model, in addition to the aforementioned metrics, we also incorporated Accuracy and the Precision-Recall Curve (PRC), metrics widely employed in similar studies [Davis and Goadrich, 2006; Muschelli, 2019; Bowers and Zhou, 2019; Alghanim *et al.*, 2022; Hicks *et al.*, 2022; Chicco and Jurman, 2023; Zain *et al.*, 2023]. Additionally, we included the Matthews Correlation Coefficient (MCC), recognized as one of the most effective metrics for evaluating binary classification models, according to Chicco and Jurman [2020].

4 Experiment Execution

We selected the PROMISE KC3 dataset, which contains object-oriented (OO) software metrics, to serve as a performance benchmark for the implemented prediction models. These models were evaluated based on their accuracy in comparison with previous research. Subsequently, we com-

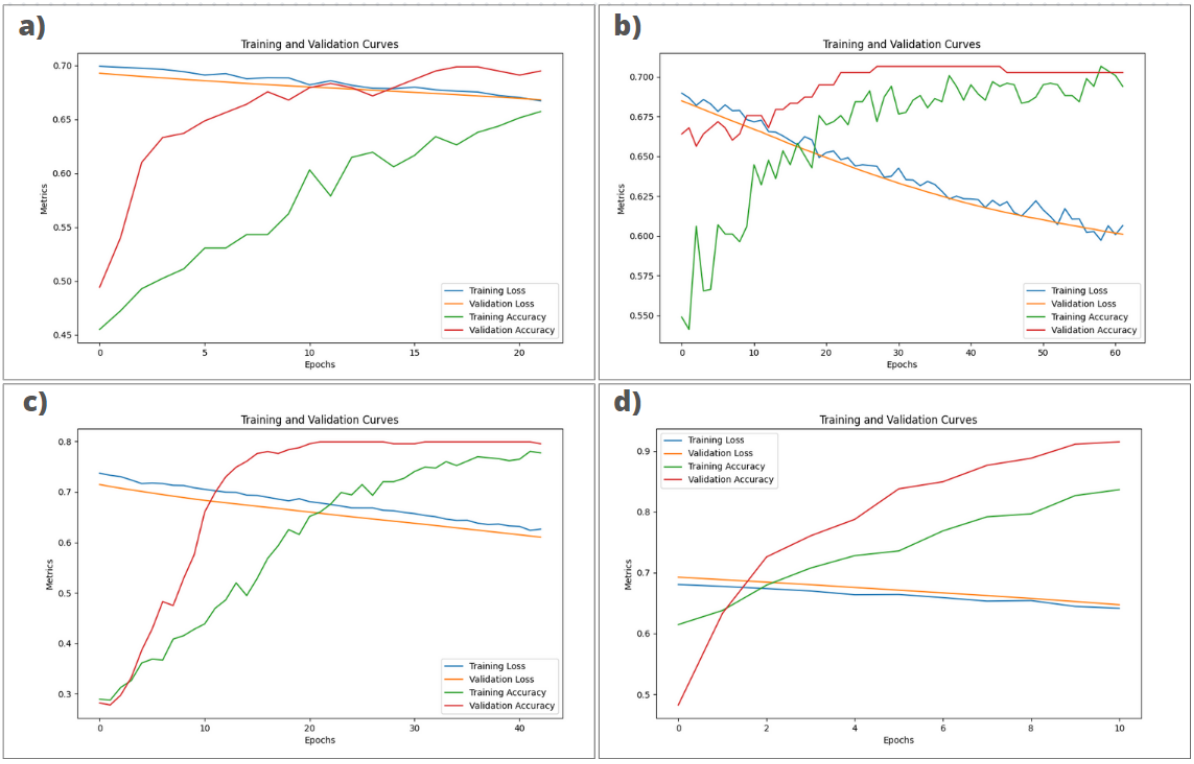

Figure 2. DNN-ESM — Curves of training and validation: (a) ARTHM; (b) LE; (c) RENT e (d) TimeO.

bined datasets containing metrics and defect labels of Solidity smart contracts, resulting in four datasets, each corresponding to a type of defect—ARTHM, LE, RENT, and TimeO, as described in Section 1. The composition of these datasets used in the experiment is presented in Table 2.

We initiated the implementation with the Accuracy, Precision, and Recall metrics.

To assist in selecting the best model, we also used the AUC, PRC, and MCC metrics. The AUC assesses the model's ability to distinguish between positive and negative classes, regardless of the classification cut-off point [Lessmann *et al*., 2008].

5 Results and Discussion

In this section, we present the results of applying different ML models for defect prediction in Ethereum smart contracts. To address research questions RQ1 and RQ2, as presented in Section 3, experiments were conducted using both traditional software metrics and specific Solidity smart contract metrics.

5.1 Results concerning RQ1

Eight Machine Learning models were applied for comparative analysis: Logistic Regression, Naive Bayes, SVM, K Nearest Neighbors (KNN), Decision Tree, Random Forest, XGBoost, and our implementation of Deep Neural Network (DNN-ESM).

Table 3 presents the results of the models for the performance metric Accuracy, which indicates the overall precision of defect prediction. It is observed that the DNN-ESM and Random Forest models applied to the KC3 object-oriented software dataset achieved the highest accuracy rates, with values of 92% and 90%, respectively.

An important aspect when evaluating the performance of classification models, especially in scenarios involving defect detection, is data imbalance. In imbalanced datasets, where one class is significantly more represented than the other, there is a risk that the model achieves high accuracy simply by learning to classify the majority class while failing to identify the minority class (defective contracts). To mitigate this effect, we use metrics such as the MCC (Matthews Correlation Coefficient) and analyze ROC-AUC and Precision-Recall curves, which are recommended in the literature for assessing model performance in imbalanced scenarios [Bahaa *et al*., 2021; Bowes *et al*., 2018]. These metrics provide a more balanced evaluation, considering both the sensitivity and specificity of the model.

Analyzing the results to address research question RQ1, we observed that the Random Forest and DNN-ESM models stood out due to their high accuracy. These results indicate that these models may be effective in defect prediction in a dataset using traditional software metrics. Observing the performance of the DNN-ESM applied to the KC3 dataset, formed by OO software metrics, we have an indication that the DNN-ESM model is adjusted, allowing us to move on to the next research question.

5.2 Results concerning RQ2

In this step, we applied the DNN-ESM model to the datasets with Solidity metrics. The results are presented in Table 4, where each row represents a specific performance metric, and each column represents its result. As explained in Section 2, each calculated metric provides additional informa-

Table 2. Datasets with OO and Solidity software metrics.

Type	Dataset	Instances	Metrics	No Defect	Defective	Defects (%)
O.O	KC3	194	39	158	36	19
Solidity	ARTHM	1292	14	432	860	67
	LE	1292	14	896	396	31
	RENT	1292	14	1145	147	11
	TimeO	1292	14	1017	275	21

Table 3. Model performance by dataset.

Models	Accuracy				
	kC3	ARTHM	LE	RENT	TimeO
Log.Regression	0,87	0,70	0,76	0,92	0,80
Naive Bayes	0,82	0,65	0,68	0,83	0,69
SVM	0,64	0,69	0,75	0,92	0,79
KNN	0,77	0,71	0,82	0,91	0,76
Decision Tree	0,87	0,73	0,88	0,89	0,84
Random Forest	**0,90**	0,80	0,90	0,93	0,89
XGBoost	0,85	0,77	0,87	0,90	0,84
DNN-ESM	**0,92**	0,68	0,70	0,80	0,81

tion about the models' performance in different classification aspects.

Table 4. Performance of DNN-ESM by Solidity dataset.

Metric	ARTHM	LE	RENT	TimeO
Accuracy	0,68	0,70	0,80	0,81
Precision	0,74	1,00	0,23	1,00
Recall	0,82	0,04	0,55	0,06
F-score	0,80	0,05	0,23	0,09
AUC	0,65	0,64	0,72	0,59
PRC	0,77	0,49	0,40	0,33
MCC	0,30	0,49	0,40	0,33

When applying the DNN-ESM model to Solidity metrics seeking to predict ARTHM, LE, RENT and TimeO vulnerabilities, the results demonstrated competitive accuracies in some cases. However, significant limitations were also revealed, such as low Recall, indicating a high rate of false negatives, and varying Precision depending on the smart contract dataset, suggesting the need for optimization, such as adjusting model parameters, modifying the neural network architecture, using data balancing techniques to better handle imbalanced datasets, or applying feature selection methods to improve the relevance of Solidity software metrics used for training.

Table 5. Comparison of accuracy between DNN-ESM and the work of Durieux *et al*. [2020].

Vulnerability	DNN-ESM	Durieux *et al*. [2020]
ARTHM	0,68	0,68 (Mythril)
LE	0,70	-
RENT	0,80	0,88 (Slither)
TimeO	0,81	0,40 (Slither)

The DNN-ESM achieved competitive results when compared to vulnerability detection methods in smart contracts reported by Durieux *et al*. [2020]. As shown in Table 5, it can be observed that for the ARTHM vulnerability, the DNN-ESM achieved an accuracy of 68%, equivalent to the 68% obtained by the Mythril tool. The detection tools did not check the LE vulnerability. Regarding the RENT vulnerability, the Slither tool showed slightly superior, recording an accuracy of 88% compared to the 80% achieved by the DNN-ESM. For the TimeO vulnerability, the DNN-ESM exhibited significantly superior performance, with an accuracy of 81%, compared to the 40% recorded by the Slither tool. These results reinforce the importance of applying ML techniques for software security in blockchain.

6 Threats to Validity

In this experiment, we employed classification algorithms that have been successfully used in software defect prediction and are commonly utilized by other researchers, as described in Section 3. However, we encountered some threats to the validity of our experiments, which can be summarized in the following three aspects.

- ***External validity:***
  The datasets used in this experiment are publicly available, both the KC3 dataset with object-oriented software metrics obtained from the PROMISE repository, and the combined datasets from Pierro and Tonelli [2020] and Yashavant *et al*. [2022] for Solidity smart contracts. However, the characteristics of the datasets differ in terms of the number of instances, attributes, etc. Thus, the results of this experiment may not replicate for other blockchain-oriented software defect datasets.
- ***Internal validity:***
  Another validity concern is the choice of classification algorithms and their configuration parameters. We selected Deep Learning algorithms for defect classification that have been successfully used in related work. However, further research could be conducted using different algorithms and techniques.
- ***Construct validity:***
  In the case of the Solidity datasets, to compose the training and test datasets, we chose the metrics compiled in the work of Pierro and Tonelli [2020] to be used as features and the dataset from Yashavant *et al*. [2022] for identifying labels. However, the proposed metrics may not be optimal, or even the method used in Yashavant *et al*. [2022] to identify data with vulnerabilities for training the Deep Learning model to identify defects.

7 Related Work

In this work, we address Deep Learning methods applied to software metrics for defect prediction. To facilitate understanding, related works are divided into two categories. The first presents Deep Learning methods for defect detection in traditional software using Object-Oriented metrics, and the second presents methods for identifying vulnerabilities in blockchain-oriented software, focusing on smart contracts on the Ethereum platform.

7.1 Software defect prediction

Fault-free software development is a challenging task Since it is executed by several individuals with different roles. Ensuring that the software performs its intended functions correctly is essential. Software defect prediction has become one of the most popular research fields in software engineering. Several studies have used datasets such as KC1, KC2, KC3, CM1, and JM1, and numerous models have been proposed for estimating and predicting software reliability [Yadav and Yadav, 2015].

Deng *et al*. [2020] leveraged a Long Short-Term Memory (LSTM) network to automatically learn semantic and contextual features from source code through Abstract Syntax Trees (ASTs). Specifically, they first extract them and then assess which and how much information they can preserve for various types of nodes. Finally, the LSTM neural network determines whether the verified source code is defective. The experimental project dataset was collected from the PROMISE repository. To evaluate the performance of the proposed model, the authors used only the F-Score measure. According to the authors, the average result indicates that the proposed model, called DP-LSTM, outperforms six out of seven defect prediction projects based on Deep Learning.

Yadav and Yadav [2015] proposed a software defect prediction model based on fuzzy logic using the relevant key reliability metrics of each phase of the Software Development Life Cycle (SDLC). Software metrics are evaluated in linguistic terms, and a fuzzy inference system was employed to develop their model. The predictive accuracy of the proposed model was validated using twenty real dataset software projects. To validate the prediction accuracy of the proposed model, they used evaluation measures of the Mean Magnitude of Relative Error and the Mean Balanced Magnitude of Relative Error. They claim that the predicted defects in the twenty evaluated software projects were very close to the actual defects detected during testing. Analyzing the results, in project 1, the model predicted 155 defects, while the actual number was 148. In project 3, 205 defects were predicted against 209 actual ones. In some cases, the prediction is practically identical to the real value, as in project 19, where both were 91 defects. The furthest predictions occur in projects with a higher number of samples with defects, such as project 11, where 1740 defects were predicted against 1768 actual ones. Even in these cases, the error is low in percentage terms. Observing the percentage differences from project to project, we see that the absolute majority is below 5%.

7.2 Smart contract vulnerability detection

Some studies focus on statistics and bug classification in smart contracts. Mostly, they address the evaluation of the source code or bytecodes of smart contracts. Pinna *et al*. [2019] conducted a study on smart contracts deployed on the Ethereum Blockchain, seeking to understand the software features and metrics of these contracts. They collected a set of over 10,000 source codes of smart contracts and a metadata dataset about their interaction with the blockchain through Etherscan. By examining the data characterizing the usage and purpose of smart contracts, they identified that the number of transactions and balances follows power law distributions, and the software code metrics exhibit, on average, lower values than the corresponding metrics in standard software, but with high variations. They also noticed that of the analyzed smart contracts, 4980 have a unique contract name being deployed only once on the blockchain and at a single address. Therefore, there is no ambiguity, but in another 1225, contract names are used more than once (from 2 to 213 times). Thus, there are very popular names registered with identical names, at different addresses, but with the same Solidity code several times.

Durieux *et al*. [2020] used SmartBugs to execute 9 automated analysis tools on two datasets of smart contracts. The first one was a dataset containing annotated vulnerable smart contracts, and the second one a total of 47,518 contracts collected on Etherscan. In total, 428,337 analyses were performed, taking approximately 564 days and 3 hours. They highlight that only 42% of the vulnerabilities contained in their annotated dataset are detected by all the tools used. A few vulnerabilities (and from only two categories) were simultaneously detected by four or more tools, with the Mythril tool having the highest precision (27%). When considering the larger collected dataset, 97% of smart contracts were marked as vulnerable, suggesting a considerable number of false positives. By observing the 20 smart contracts with the highest number of transactions, they found that most of them represent financial smart contracts.

Gogineni *et al*. [2020] used Deep Learning to classify smart contracts into Suicidal, Prodigal, Greedy, or Normal categories using a variant of LSTM called Average Stochastic Gradient Descent Weight-Dropped LSTM (AWD-LSTM). They trained and validated the AWD-LSTM with input and output vectors of the same length. For multi-class classification, they replaced the "decoder" layer of the AWD-LSTM with some fully connected layers, known as a "custom head." According to the authors, they combined a pre-trained encoder with the "custom head" to obtain better classification. To evaluate the results, they used Accuracy, Precision, Recall, F1 score, and the Confusion Matrix, as well as the ROC curve to display a graph between the true positive rate and the false positive rate of predictions and the AUC metric to check the neural networks' ability to distinguish between various classes.

Fan *et al*. [2021] proposed a model for detecting vulnerabilities in smart contracts on blockchains, called Dual Attention Graph Convolutional Network (DA-GCN). This model was applied to real-world smart contract datasets containing two reentrancy vulnerabilities. The DA-GCN is based on a

graph convolutional network and self-attention mechanisms that focus on the most important nodes and suppress useless information extracted from smart contract bytecodes. Moreover, a multilayer perceptron model is used to identify if the smart contract is vulnerable.

Liu *et al*. [2023] used Graph Neural Networks (GNNs) and expert knowledge of smart contract source codes for vulnerability detection. A total of 16 methods were investigated, where the performance was compared in terms of Accuracy, Recall, Precision, and F1 score. According to the authors, the results show significant improvements in accuracy for three types of vulnerabilities: for reentrancy, timestamp dependency, and infinite loop.

Tonelli *et al*. [2023] studied software metrics of smart contracts extracted from a dataset of over 85,000 contracts deployed on the Ethereum Blockchain. They sought to determine whether the corresponding software metrics present differences in their statistical properties compared to metrics extracted from traditional software systems and widely studied in the literature. The assumptions are that resources are limited on the blockchain, and such limitations may influence how smart contracts are written. The analysis dealt with source code metrics as well as the Application Binary Interface (ABI) and bytecode of smart contracts. According to the authors, the main results show that smart contract metrics tend to suffer from limited blockchain resource constraints, and the exposure of smart contracts to blockchain interaction measured qualitatively in terms of ABI size is quite similar among them without the presence of outliers. The distribution is compatible with the Gaussian statistical distribution (normal), characterized by its bell-shaped form, without the omnipresent presence of fat-tailed distributions, where there are values much distant from the mean, as typical in traditional software. In smart contract software metrics, all values generally fall within a range of a few standard deviations from the mean, and large variations from the mean are substantially unknown. Remarkably, the LOC of smart contracts is the metric that most closely approximates the statistical distribution of the corresponding metric observed in traditional software systems.

Even with some research in the field of smart contracts, there are still a few labelled datasets with defects. To our knowledge, there is no dataset with solidity metrics like Pierro and Tonelli [2020] labelled with vulnerabilities. In general, the literature on defect prediction in smart contracts, especially the availability of datasets with blockchain-oriented software metrics, especially on the Solidity programming language, is still quite limited [Pierro and Tonelli, 2020; Jiang *et al*., 2023], which makes it more challenging to produce works based on defect prediction in smart contracts, and the main works address vulnerabilities in contracts identified from the source code or bytecodes of the Ethereum Virtual Machine (EVM) [Lutz *et al*., 2021; Zhang *et al*., 2022].

8 Conclusion

Smart contracts introduce new metrics and peculiarities beyond those found in traditional software, bringing unique challenges for defect detection. The results presented in this work indicate that the evaluated Machine Learning models can be effective in predicting defects in smart contracts. Furthermore, DNN-ESM applied to specific Solidity metrics showed competitive performance compared to the methods evaluated in the work of Durieux *et al*. [2020], although it presents limitations that require future optimizations. For instance, the model exhibited a low Recall rate in some datasets, indicating a high false negative rate. Additionally, the model's Precision varied depending on the dataset, suggesting the need for adjustments in model parameters, neural network architecture, and the use of data balancing techniques to better handle imbalanced datasets.

In future work, we plan to construct a labelled dataset with compiled metrics based on the studies and add improvements to our DNN-ESM model focusing on moderate computational resource usage. We will also attempt to explore feature selection techniques in a dataset for defect prediction.

Declarations

Funding

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001; GBL is supported by the Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ), grant no. E-26/210.430/2024.

Authors' Contributions

RJO contributed to data curation, software development, visualization, and drafting the original manuscript. EML supervised the project, validated the findings, and reviewed and edited the manuscript. GBL was responsible for funding acquisition, project administration, resource allocation, supervision, validation, and reviewing and editing the manuscript. All authors contributed equally to conceptualization, formal analysis, investigation, and methodology.

Competing interests

We declare no conflicts of interest to disclose.

References

Alghanim, F., Azzeh, M., El-Hassan, A., and Qattous, H. (2022). Software defect density prediction using deep learning. *IEEE Access*, 10:114629–114641. DOI: 10.1109/ACCESS.2022.3217480.

Badruddoja, S., Dantu, R., He, Y., Upadhayay, K., and Thompson, M. (2021). Making smart contracts smarter. In *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–3. DOI: 10.1109/ICBC51069.2021.9461148.

Bahaa, A., Fathy, E. M., Eldin, A. S., and Abd-Elmegid, L. A. (2021). A systematic literature review of software defect prediction using deep learning. *Journal of Computer Science*, 17:490–510. DOI: 10.3844/JCSSP.2021.490.510.

Beck, R., Avital, M., Rossi, M., and Thatcher, J. B. (2017). Blockchain technology in business and information systems research. *Business & information systems engineering*, 59:381–384. DOI: 10.1007/s12599-017-0505-1.

Bhargavan, K., Swamy, N., Zanella-Béguelin, S., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., and Sibut-Pinote, T. (2016). Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, page 91–96. Association for Computing Machinery. DOI: 10.1145/2993600.2993611.

Bowers, A. and Zhou, X. (2019). Receiver operating characteristic (roc) area under the curve (auc): A diagnostic measure for evaluating the accuracy of predictors of education outcomes. *Journal of Education for Students Placed at Risk (JESPAR)*, 24:1–25. DOI: 10.1080/10824669.2018.1523734.

Bowes, D., Hall, T., and Petrić, J. (2018). Software defect prediction: do different classifiers find the same defects? *Software Quality Journal*, 26. DOI: 10.1007/s11219-016-9353-3.

Chang, R., Mu, X., and Zhang, L. (2011). Software defect prediction using non-negative matrix factorization. *JSW*, 6:2114–2120. DOI: 10.4304/jsw.6.11.2114-2120.

Chen, T., Cao, R., Li, T., Luo, X., Gu, G., Zhang, Y., Liao, Z., Zhu, H., Chen, G., He, Z., Tang, Y., Lin, X., and Zhang, X. (2020). Soda: A generic online detection framework for smart contracts. *Proceedings 2020 Network and Distributed System Security Symposium*. DOI: 10.14722/ndss.2020.24449.

Chicco, D. and Jurman, G. (2020). The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *BMC Genomics*, 21. DOI: 10.1186/s12864-019-6413-7.

Chicco, D. and Jurman, G. (2023). The matthews correlation coefficient (mcc) should replace the roc auc as the standard metric for assessing binary classification. *BioData Mining*, 16(1):1–23. DOI: 10.1186/s13040-023-00322-4.

Davis, J. and Goadrich, M. (2006). The relationship between precision-recall and roc curves. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, page 233–240, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/1143844.1143874.

Deng, J., Lu, L., and Qiu, S. (2020). Software defect prediction via lstm. *IET software*, 14(4):443–450. DOI: 10.1049/IET-SEN.2019.0149.

Durelli, V. H. S., Durelli, R. S., Borges, S. S., Endo, A. T., Eler, M. M., Dias, D. R. C., and Guimarães, M. P. (2019). Machine learning applied to software testing: A systematic mapping study. *IEEE Transactions on Reliability*, 68(3):1189–1212. DOI: 10.1109/TR.2019.2892517.

Durieux, T., Ferreira, J. a. F., Abreu, R., and Cruz, P. (2020). Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 530–541, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3377811.3380364.

Fan, Y., Shang, S., and Ding, X. (2021). Smart contract vulnerability detection based on dual attention graph convolutional network. In *Collaborative Computing: Networking, Applications and Worksharing: 17th EAI International Conference, CollaborateCom 2021, Virtual Event, October 16-18, 2021, Proceedings, Part II 17*, pages 335–351. Springer. DOI: 10.1007/978-3-030-92638-0_20.

Fenton, N. and Bieman, J. (2014). *Software Metrics: A Rigorous and Practical Approach, Third Edition*. CRC Press, Inc.. DOI: 10.1201/b17461.

Gao, Y., Liu, W., and Lombardi, F. (2020). Design and implementation of an approximate softmax layer for deep neural networks. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. DOI: 10.1109/ISCAS45731.2020.9180870.

Ghaffarian, S. M. and Shahriari, H. R. (2017). Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Comput. Surv.*, 50(4). DOI: 10.1145/3092566.

Gogineni, A. K., Swayamjyoti, S., Sahoo, D., Sahu, K., and Kishore, R. (2020). Multi-class classification of vulnerabilities in smart contracts using awd-lstm, with pre-trained encoder inspired from natural language processing. *IOP SciNotes*, 1:035002. DOI: 10.1088/2633-1357/abcd29.

Hicks, S., Strumke, I., Thambawita, V., Hammou, M., Halvorsen, P., Riegler, M., and Parasa, S. (2022). On evaluation metrics for medical applications of artificial intelligence. *Scientific Reports*, 12(1):5979. DOI: 10.1038/s41598-022-09954-8.

Huang, J., Zhou, K., Xiong, A., and Li, D. (2022). Smart contract vulnerability detection model based on multi-task learning. *Sensors*, 22(5):1829. DOI: 10.3390/s22051829.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 448–456. JMLR.org. DOI: 10.5555/3045118.3045167.

ISDW (2010). Ieee standard classification for software anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pages 1–23. DOI: 10.1109/IEEESTD.2010.5399061.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013). *An Introduction to Statistical Learning: With Applications in R*, volume 112. Springer. DOI: 10.1007/978-1-4614-7138-7.

Jiang, F., Chao, K., Xiao, J., Liu, Q., Gu, K., Wu, J., and Cao, Y. (2023). Enhancing smart-contract security through machine learning: A survey of approaches and techniques. *Electronics*, 12(9). DOI: 10.3390/electronics12092046.

Kingma, D. P. and Ba, J. (2017). Adam: A method for stochastic optimization. DOI: 10.48550/arXiv.1412.6980.

Lessmann, S., Baesens, B., Mues, C., and Pietsch, S. (2008). Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE transactions on software engineering*, 34(4):485–496. DOI: 10.1109/TSE.2008.35.

Liang, H., Yu, Y., Jiang, L., and Xie, Z. (2019). Seml: A semantic lstm model for software defect predic-

tion. *IEEE Access*, 7:83812–83824. DOI: 10.1109/ACCESS.2019.2925313.

Liu, Z., Qian, P., Wang, X., Zhuang, Y., Qiu, L., and Wang, X. (2023). Combining graph neural networks with expert knowledge for smart contract vulnerability detection. *IEEE Transactions on Knowledge and Data Engineering*, 35(2):1296–1310. DOI: 10.1109/TKDE.2021.3095196.

Lutz, O., Chen, H., Fereidooni, H., Sendner, C., Dmitrienko, A., Sadeghi, A. R., and Koushanfar, F. (2021). Escort: ethereum smart contracts vulnerability detection using deep neural network and transfer learning. *ArXiv*. DOI: 10.14722/ndss.2023.23263.

Matthews, B. (1975). Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405(2):442–451. DOI: 10.1016/0005-2795(75)90109-9.

Mirjalili, S., Faris, H., and Aljarah, I. (2020). *Introduction to Evolutionary Machine Learning Techniques*, pages 1–7. Springer Singapore. DOI: 10.1007/978-981-32-9990-0_1.

Muschelli, J. (2019). Roc and auc with a binary predictor: a potentially misleading metric. *Journal of Classification*, 37. DOI: 10.1007/s00357-019-09345-1.

Ng, A. (2004). Feature selection, l1 vs. l2 regularization, and rotational invariance. *Proceedings of the twenty-first international conference on Machine learning*. DOI: 10.1145/1015330.1015435.

Ortu, M., Orrú, M., and Destefanis, G. (2019). On comparing software quality metrics of traditional vs blockchain-oriented software: An empirical study. In *2019 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 32–37. DOI: 10.1109/IWBOSE.2019.8666575.

O'Shea, K. and Nash, R. (2015). An introduction to convolutional neural networks. *ArXiv e-prints*. DOI: 10.48550/arXiv.1511.08458.

Osterland, T. and Rose, T. (2020). Model checking smart contracts for ethereum. *Pervasive and Mobile Computing*, 63:101129. DOI: 10.1016/j.pmcj.2020.101129.

Ozakinci, R. and Tarhan, A. (2016). The role of process in early software defect prediction: Methods, attributes and metrics. In *International Conference on Software Process Improvement and Capability Determination*. DOI: 10.1007/978-3-319-38980-6_21.

Pachouly, J., Ahirrao, S., Kotecha, K., Selvachandran, G., and Abraham, A. (2022). A systematic literature review on software defect prediction using artificial intelligence: Datasets, data validation methods, approaches, and tools. *Engineering Applications of Artificial Intelligence*, 111:104773. DOI: 10.1016/j.engappai.2022.104773.

Pierro, G. A. and Rocha, H. (2019). The influence factors on ethereum transaction fees. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 24–31. DOI: 10.1109/WETSEB.2019.00010.

Pierro, G. A. and Tonelli, R. (2020). Paso: A web-based parser for solidity language analysis. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 16–21. DOI: 10.1109/IWBOSE50093.2020.9050263.

Pinna, A., Ibba, S., Baralla, G., Tonelli, R., and Marchesi, M. (2019). A massive analysis of ethereum smart contracts empirical study and code metrics. *IEEE Access*. DOI: 10.1109/ACCESS.2019.2921936.

Porru, S., Pinna, A., Marchesi, M., and Tonelli, R. (2017). Blockchain-oriented software engineering: Challenges and new directions. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 169–171. DOI: 10.1109/ICSE-C.2017.142.

Qiao, L., Li, X., Umer, Q., and Guo, P. (2020). Deep learning based software defect prediction. *Neurocomputing*, 385:100–110. DOI: 10.1016/j.neucom.2019.11.067.

Rezaei-Dastjerdehei, M. R., Mijani, A., and Fatemizadeh, E. (2020). Addressing imbalance in multi-label classification using weighted cross entropy loss function. In *2020 27th National and 5th International Iranian Conference on Biomedical Engineering (ICBME)*, pages 333–338. DOI: 10.1109/ICBME51989.2020.9319440.

Saifan, A. and Abu-wardih, L. (2020). Software defect prediction based on feature subset selection and ensemble classification. *ECTI Transactions on Computer and Information Technology (ECTI-CIT)*, 14:213–228. DOI: 10.37936/ecti-cit.2020142.224489.

Sui, J., Chu, L., and Bao, H. (2023). An opcode-based vulnerability detection of smart contracts. *Applied Sciences*, 13(13). DOI: 10.3390/app13137721.

T R, M., Thakur, A., Sinha, D., Mishra, K., Kumar, V. V., and Guluwadi, S. (2024). Transformative breast cancer diagnosis using cnns with optimized reducelronplateau and early stopping enhancements. *International Journal of Computational Intelligence Systems*, 17. DOI: 10.1007/s44196-023-00397-1.

Tonelli, R., Pierro, G. A., Ortu, M., and Destefanis, G. (2023). Smart contracts software metrics: A first study. *PLOS ONE*, 18(4):1–31. DOI: 10.1371/journal.pone.0281043.

Tong, H., Liu, B., and Wang, S. (2018). Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning. *Information and Software Technology*, 96:94–111. DOI: 10.1016/j.infsof.2017.11.008.

Treiblmaier, H. (2020). *Toward More Rigorous Blockchain Research: Recommendations for Writing Blockchain Case Studies*, pages 1–31. Springer International Publishing. DOI: 10.1007/978-3-030-44337-5_1.

Velasco, G., Vaz, N., and Carvalho, S. (2023). Challenges and opportunities in smart contract development on the ethereum virtual machine: A systematic literature review. In *Anais do VI Workshop em Blockchain: Teoria, Tecnologias e Aplicações*, pages 15–28, Porto Alegre, RS, Brasil. SBC. DOI: 10.5753/wblockchain.2023.756.

Wang, H., Khoshgoftaar, T., and Napolitano, A. (2012). Software measurement data reduction using ensemble techniques. *Neurocomputing*, 92:124–132. DOI: 10.1016/j.neucom.2011.08.040.

Xu, J., Li, Z., Du, B., Zhang, M., and Liu, J. (2020). Reluplex made more practical: Leaky relu. In *2020 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–7. DOI: 10.1109/ISCC50000.2020.9219587.

Yadav, H. B. and Yadav, D. K. (2015). A fuzzy logic based

approach for phase-wise software defects prediction using software metrics. *Information and Software Technology*, 63:44–57. DOI: 10.1016/j.infsof.2015.03.001.

Yashavant, C., Kumar, S., and Karkare, A. (2022). Scrawld: A dataset of real world ethereum smart contracts labelled with vulnerabilities. *arXiv preprint arXiv:2202.11409*. DOI: 10.48550/arXiv.2202.11409.

Zain, Z. M., Sakri, S., and Ismail, N. H. A. (2023). Application of deep learning in software defect prediction: Systematic literature review and meta-analysis. *Inf. Softw. Technol.*, 158(C). DOI: 10.1016/j.infsof.2023.107175.

Zeng, P., Lin, G., Pan, L., Tai, Y., and Zhang, J. (2020). Software vulnerability analysis and discovery using deep learning techniques: A survey. *IEEE Access*, 8:197158–197172. DOI: 10.1109/ACCESS.2020.3034766.

Zhang, L., Chen, W., Wang, W., Jin, Z., Zhao, C., Cai, Z., and Chen, H. (2022). Cbgru: A detection method of smart contract vulnerability based on a hybrid model. *Sensors*, 22(9). DOI: 10.3390/s22093577.

Zhang, P., Xiao, F., and Luo, X. (2020). A framework and dataset for bugs in ethereum smart contracts. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 139–150. DOI: 10.1109/ICSME46990.2020.00023.

Zhang, Y. and Liu, D. (2022). Toward vulnerability detection for ethereum smart contracts using graph-matching network. *Future Internet*, 14(11). DOI: 10.3390/fi14110326.