International Journal of Computer Engineering in Research Trends
ISSN: 2349-7084/ https://doi.org/10.22362/ijcert/2024/v11/i4/v11i404
Volume 11, Issue 4, April 2024

Research Paper

A Deep Dive into Code Smell and Vulnerability Using Machine Learning and Deep Learning Techniques

Kritika*

** Independent Researcher, New Delhi, India, kritikaa2297@yahoo.com*

**Corresponding Author(s): kritikaa2297@yahoo.com*

Received: 16/01/2024, Revised: 07/03/2024, Accepted:28/04/2024 Published:30/04/2024

Abstract: - *Sustainable software development practices are essential for ensuring code quality, maintainability, and security. However, traditional approaches often overlook the presence of code smells and vulnerabilities, leading to technical debt and security risks. This paper presents a comprehensive analysis of code smells and vulnerabilities in Java applications using machine learning and deep learning techniques. The study curates' datasets from 25 Java applications, utilizing tools like PMD, JDeodorant, IntelliJ Idea, and SciTools Understand to detect code smells and vulnerabilities, and compute software metrics. The experimental approach applies supervised machine learning algorithms and deep learning models, including Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN), on the pre-processed datasets. The results demonstrate that the JRIP algorithm produces the best results for vulnerabilities like Law of Demeter (81.51% accuracy), Too Many Methods (97.09% accuracy), and Local Variable Could Be Final (88.07% accuracy). For the Beam Member Should Serialize vulnerability, the J48 algorithm achieves an accuracy of 96.2%. The PMD tool outperforms IntelliJ Idea in detecting code smells like God Class (>90% accuracy) and Long Method (>90% accuracy) in Java applications. Additionally, the study establishes a relationship between code smells and vulnerabilities, with algorithms like J48 and JRIP effectively identifying patterns across both. Regarding deep learning techniques, CNN achieves higher accuracy than RNN for code smells like God Class (90.08% vs. 86.78%) and Long Method (89.18% vs. 81.08%). However, for vulnerabilities, CNN excels in detecting Law of Demeter (96.77% accuracy) and Cyclomatic Complexity (92.64% accuracy), while RNN demonstrates better performance for Beam Member Should Serialize (88.4% accuracy) and Too Many Methods (94.28% accuracy).*

Keywords- *Code smell, Vulnerability, Deep Learning, Machine Learning, Software Metrics, Java Applications*

---

1. Introduction

The breakneck blossoming of technology, at the same time fabrication of unhackneyed software has paved the way for expeditious menace eventuating due to dearth of software prolongation. The most important phase of software development lifecycle model is the maintenance phase which often gets pretermit due to paucity of time, coercion, contention and lack of blue-collar workers. The hotfoot proliferation of cyber-attacks concerns occupancy of code smells prevalent in the delineation of software being thrived. A Code Smell [1] is a practical overdrawn in code that highlights transgression of vestigial delineation predication and negative smack on delineation oddity.

Code smells can be laconically pigeonholed into bloaters, object-oriented abusers, change preventers, dispensable and couplers. The bloaters are those smells which have increased ginormous expanse that it becomes strenuous to toil with such as long parameter list, long. method, large class, primitive obsession etc. The object-oriented abusers are the types of smells that are pitchy and fallacious in genre. The application of such software unfastened the doors for threats and misappropriation of information such as temporary field, statements using switch condition, classes having alternate interface etc.

The change preventers code smells dictate the exigency of transposing the code at a notable location to be transposed at every location intact with it such as shortgun surgery, parallel inheritance hierarchies, divergent change. The dispensable code smells bestow with uncalled for slice of code if put to an end will make code more legible and methodical like lazy class, data class, duplicate code, comments etc.

The couplers are the class of category that bestows with immoderate conjoining of classes such as feature envy,

message chains, middleman, etc. Deep learning is a rendition of artificial intelligence with the objective to develop programs that can recuperate data and wield its swotting [2]. It wields numerous superimposed neural reticulations to master level of brooding and portrays a perception of data. *Fig 1* represents the hierarchy based on artificial intelligence, machine learning and deep learning. Artificial intelligence is a discipline where competence of a system to broadcast the forte of human mind by wielding swotting from bygone ordeals to pin down, discern, respond etc. Machine learning is a vital component of artificial intelligence that comprehend fundamentally, with the proficiency to comprehend programs without compiling.

1.1 Annals of Deep Learning Architectonic

The surge of the technological furtherance over the decades from evolution of back propagation model to generative contentious neural network has paved way for emergence of technologies like ChatGPT, chatbot devices etc. The neural networks are a multitude of confluence called neurons which broadly comprises of three integrant, namely, neurons, connections or weights and propagation function [2] as indicated in *fig 2*.

Deep learning is a present-day technology on neural network that essay to toil like human brain. The input layer consists of the unrefined particulars. The hidden layer manoeuvres multiple algorithms to undertake the inputs. The output layer thus produces the desired outcomes based on the efforts of input and hidden layers. *Fig 3* represents the deep neural network with embedded hidden layers.

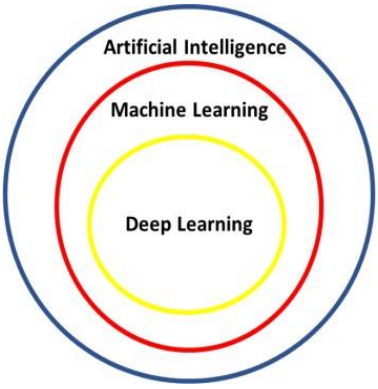

Fig 1: Hierarchy

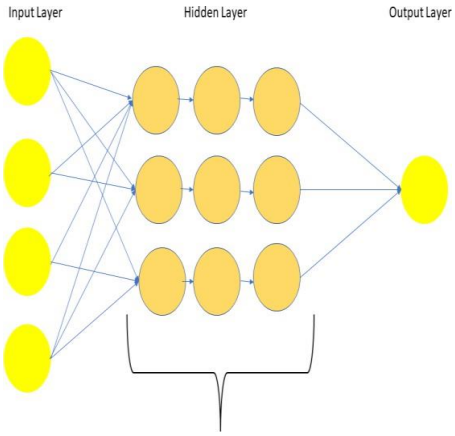

Fig 2: Basic neural network

Deep learning can be further classified into CNN (convolutional neural network) and RNN (recurrent neural network). CNN is a meticulous subset which manoeuvre on kernels, a reticulation of neurons for image processing [3] with the objective of diminishing the aggregate of criterion. A deep feed forward architecture with the capability of having no dimensionally dependent artifacts [4][5]. The pivotal fascination for soliciting CNN is the conceptualization of weight sharing for improvisation of parameters to be drilled with ease and smoothness [4] and feature extraction that goes hand in hand with large datasets. The CNN can be subdivided into 5 distinguished layers as laid down in *fig 4*, namely, input, convolution, pooling, fully connected and output.

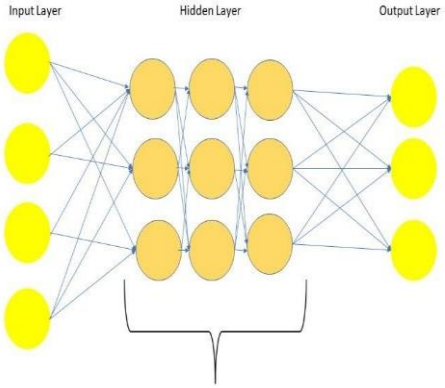

Fig 3: Deep Neural Network

The layer input consists of all the raw data that is initially fed into the network on which further modelling will take place. The convolution layer is cast off with the steadfastness of the facet prevalent in the input layer. The facet is then viewed as a learning parameter, learnt during the training phase. The filter kernels refer to as weights are ushered in and then streamlined by backpropagation using declension.

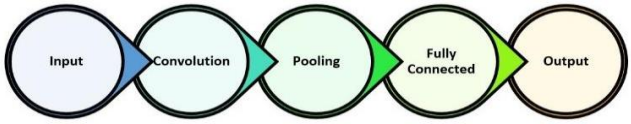

Fig 4 CNN Layers

The pooling layer succors in diminishing the input size while preserving all the vital characteristics of the input. It reduces the sum total of criterion and reckoning in the network. The fully connected layer is the ultimate layer of the architecture which applies an untwisted combination and a trigger. The output layer presents the result derived after bypassing and going hand in hand with the desired parameters to produce a much-deserved result. The layers mentioned of the CNN are depicted in the figure 4 of the paper.

The second technique selected for purpose of research is Recurrent Neural Network (RNN) technique of deep learning. The network is capable of distilling a series of despotic expanse by recurrent application of a transition function to camouflage input function [6]. RNNs as depicted in fig 5 are the most persuasive and rudimentary

neural networks to operate on sequential data coherently because of the presence of hidden internal memory which makes it proficient of recollecting the input information provided to the network. The list of sequential data includes text, image, finance, speech etc.

The recurrent neural network consists of input layer, hidden layers and output layers in which the input layer is responsible for preprocessing data by passing it through filters to the hidden layer which in turn consists of neural networks, heuristics, and actuating. The processed information is then sent to the output layer to produce the desired outcome.

The processing in the RNN ranges notably as such the next input is completely dependent on the information passed by the previous input layers which also makes it back propagation neural network. The objective of back propagation mechanism in the implementation of RNNs is that it can help in associating the fragmentary echoic of fallacy with respect to weights, in turn maneuver the declivity descent to minimize loss function.

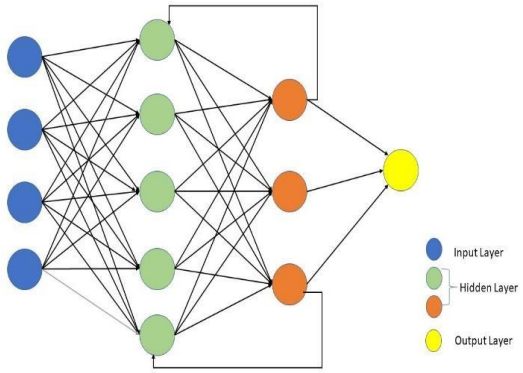

Fig 5 Recurrent neural network

1.2 Annals of Machine Learning Architectonic

Machine learning is an application of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. Machine learning [35] focuses on the development of computer programs that can obtain data and leverage it learn for themselves. The process of learning begins with observations or information, such as examples, direct experience, or instruction, to look for patterns in data and make better decisions in the future based on the examples that we provide. The primary goal is to allow the computers learn automatically without human intervention or assistance and adjust actions accordingly.

Supervised machine learning algorithms [36] can apply what has been learned in the past to new data using labelled examples to predict future events. Starting from the analysis of a known training dataset, the learning algorithm produces an inferred function to make predictions about the output values. The system can provide targets for any new input after sufficient training. The learning algorithm can also compare its output with the correct, intended output and find errors to modify the model accordingly. Unsupervised machine learning algorithms [38] are used when the information used to train is neither classified nor labelled. Unsupervised learning studies how systems can infer a function to describe a hidden structure from unlabelled data. The system doesn't figure out the right output, but it explores the data and can draw inferences from datasets to describe hidden structures from unlabelled data.

Semi-supervised machine learning algorithms [37] fall somewhere in between supervised and unsupervised learning, since they use both labeled and unlabeled data for training – typically a small amount of labeled data and a large amount of unlabeled data. The systems that use this method can considerably improve learning accuracy. Usually, semi-supervised learning is chosen when the acquired labelled data requires skilled and relevant resources in order to train it / learn from it. Otherwise, acquiring unlabeled data generally doesn't require additional resources.

Reinforcement machine learning algorithms [39] is a learning method that interacts with its environment by producing actions and discovers errors or rewards. Trial and error search and delayed reward are the most relevant characteristics of reinforcement learning. This method allows machines and software agents to automatically determine the ideal behavior within a specific context in order to maximize its performance. Simple reward feedback is required for the agent to learn which action is best; this is known as the reinforcement signal.

Machine learning enables analysis of massive quantities of data. While it generally delivers faster, more accurate results in order to identify profitable opportunities or dangerous risks, it may also require additional time and resources to train it properly. Combining machine learning with AI and cognitive technologies can make it even more effective in processing large volumes of information.

1.3 Code Smells

Code smells [7] are the technological obligations in code that highlights dereliction of rudimentary delineation predication and pessimistic thwack on dereliction eccentricity.

God class [8] code smell is a subclassification of code smell which take no notice of the postulate that individual aspect of the programming software should have single culpability and single resolve. Its contrivances the humungous class, manufacturing compact coupling, multiplying the complexity of the maintainability issue of the code.

Long method [8] code smell is a subcategory of code smells that happen when the expanse of the method is colossal and are prone to amalgamate the efficaciousness of the class. It manifests the violation of a single culpability of the postulate of the object-oriented programming as they are compounded, exigent and gruelling. Both long method and god class code smells are destructive and endemic to software idiosyncrasy, the refactoring of which provides an inflated advantage to predilection and overall augmentation of the maintainability of the software.

1.4 Vulnerability

Vulnerability [1] is a security shortcoming, hitch or Achilles heel hinged in software code that can be escapade by an assailant. The factors on which software vulnerability can be categorized are extant i.e. presence of vulnerability in the software system, flare-up i.e. likelihood that attacker gained entrance to the vulnerability and escapade i.e. the aptness of the attacker to enslave the software with the help of certain instrument or methodology.

Law of Demeter aka philosophy of least cognizance is a delineation precept for building up software in line with object-oriented programming. It's a discrete exposition of wobbly integration. The advantage that the law of demeter holds is that it administers to be more accommodating and rectifiable as the objects are least determined by intramural edifice of other objects. A multi-layered edifice is most suitable for implementing such a vulnerability. Despite having positiveness, may write sheathing methods to promulgate the pieces of code. The best way to escapade the violation is by inducting methods on repossessed objects which may not lead to null pointers.

Beam member should serialize is a sub classification of vulnerability that deals with fields either in serializable or transient state of the art. The serializable objects with non-transient, non-serializable objects can lead to crashing of the programme, perforating the entrance for attackers. The keyword transient plays a vital role in meeting collateral constraints in the software and holds no significance with the static fields as it pertains to no compilation error. The best way to escapade the violation is by not approving the untrusted source, by enciphering the hostile objects and usage of web solicitation fire block.

Too many methods is a violation which often tends to have colossal amount of methods with similar or distinguished taxonomy and differing criterion in a class. The methods may be susceptible or insusceptible to other methods prevalent in a particular class. Despite the positives, the negation lies in the fact that more time is required than desired to build a vast code for the software which makes it intricate and enigmatic. The prudent way to escapade it is retitling the method appellative, manoeuvring constant method and bifurcating into smaller segments.

Cyclomatic complexity [9] is an expedient of aggregate of self-sustaining esplanade of every single method within a class. A method having more quantity of conditionality will eventually lead to higher quantity of unrestraint paths. The lofty the quantity of self-sustaining esplanade, the exorbitant the assay of developers are. The advantages of the violation hold in the fact that can be used as caliber yardstick with faster rate of computation, provide direction to testing process and aggregates the least amount of assay and agglomeration of evaluation.

Despite having positives, certain negatives incorporated by the manoeuvring of it is aggregates only the conditionality entanglement and not the data entanglement. The garnered structures are difficult to acknowledge than the non-garnered ones. One of the ways of refactoring such a violation is by cannibalizing the code and reduce the conditionality statements.

1.5 Software metrics

The object-oriented metrics [10][11] is an expedient of disposition which can be unwavering and discernible. The metrics can broadly be broadly designated into product and process metrics. The process metrics is defined as disposition of assorted trademark of the software development. The product metrics is defined as the disposition of assorted trademark of the software product.

The gimmick pre-owned for assessing object-oriented metrics is Scitool Understand [12] which provides a humungous quantity of metrics value categorized into three comprehensive genres viz. complexity metrics, object-oriented metrics and volume metrics. The contrivance is pre-owned for research exploration as it comes in handy with lot of highlights like perusing the code with colossal lines of code, underpins various programming languages like java, c++, ruby etc. and provides bulletin, plotting, and pennon assessing.

The contrivance cast off for discerning code smells and software vulnerabilities in distinguished java applications is a plugin, PMD [13] simpatico with Eclipse. The contrivance keeps in handy the incorporated precept as well as impart an environs custom tailored precept. The mass exuberant characteristics reported by this contrivance is the ineffectual code or the indigent programming conventions, one of the crucial aspects while discerning code smells and vulnerabilities in a source code.

Following is the exhaustive list of software metrics being manoeuvred for the research analysis of the varied java applications as mentioned in table 1. The vital scripted oddity comprehends writing down and enforcing code, mathematical equations, construct, transmit or have a stake in notebooks and amalgamate numerous libraries like PyTorch, TensorFlow, Keras etc.

The paper delves into the following research questions:

RQ1: *Which machine learning algorithm produces best results for a particular vulnerability?*

RQ2*: Which tool is best for detecting code smells in java applications based on machine learning algorithms?*

RQ3*: Is there exists a similarity between code smell and vulnerability?*

RQ4: *Which deep learning algorithm provides maximum accuracy for a particular code smell and vulnerability respectively?*

The paper is organized into several sections to provide a structured and comprehensive analysis of code smell and vulnerability detection using machine learning and deep learning techniques. The sections are organized as follows:

Section 1 (Introduction) which provides an overview of the research problem, highlighting the importance of detecting code smells and vulnerabilities in software development and presents the research questions addressed in the paper. Section 2 (Literature Review) critically reviews and analyses existing literature related to code smell and vulnerability detection techniques, identifies gaps and limitations in previous research efforts, such as restricted environments,

limited detection techniques, lack of comparative analyses, and limited exploration of deep learning techniques. Section 3 (Various Tools Used) describes various tools employed in the research methodology, including advisors (PMD, IntelliJ Idea, JDeodorant), metrics computation tools (SciTools Understand), and deep learning implementation tools (Google Colab, libraries like Keras, Pandas, and NumPy). Section 4 (Experimental Approach) outlines the comprehensive experimental approach followed in the research which describes eight distinct phases involved, including corpus collection, code smell and vulnerability detection, software metrics computation, dataset formalization, data pre-processing, dataset normalization, testing and training of data, and the application of machine learning and deep learning techniques along with detailed explanation of each phase, including the methodologies, tools, and techniques employed. Section 5 (Result and Discussion) presents the quantitative results obtained from the experimental approach, addressing the research questions posed in the introduction. It also provides a detailed analysis of the performance of different machine learning algorithms and deep learning techniques (CNN and RNN) for specific code smells and vulnerabilities, including accuracy metrics, discusses the relationships and similarities between code smells and vulnerabilities identified through the analysis. Section 6 (Conclusion) provides the analytical analysis of the research performed in the previous section. Section 7 (Threat to Validity) acknowledges potential threats to the validity of the research findings and discusses how these threats were mitigated or addressed in the experimental approach. Section 8 (Future Recommendations) outlines future recommendations and directions for extending the research, such as exploring additional code smells and vulnerabilities, incorporating refactoring techniques, and expanding the scope of the analysis.

Table 1. List of Metrics

Software Metrics	Definition
CountDeclMethod	Number of Class methods
Sum Cyclomatic	Sum of each and all impacted functions and methods
Count Line	Sum total of all lines of code
Sum Cyclomatic Strict	Sum of strict headlong complexities of each and every encapsulated function
Count Paths	Aggreagate of all possible paths
CountDeclClass	Number of classes
Count Line Code	Number of lines of actual source code
MaxCyclomatic	Maximum complexity of all ingrained functions or methods
CountDeclInstance Method	Number of adduce methods
CountDeclInstance Variable	Number of adduce variables
Count Input	Amalgamation of calling subroutines along with universal variables
Count Output	Amalgamation of called subroutines with universal variables read
AltAverageLineCode	Par lines of code consisting of source code with all ingrained functions or methods with inactive regions such as blank lines, comments

2 Literature Review

The novelty of the concept of code smells and vulnerabilities is primeval as researchers from decade long are working on this concept but the research methodology adopted in this paper focusses on the contemporary techniques of deep learning with primary focus on static applications developed in java while neglecting the minute details in a hurry to remit the product to the client and taking no notice of the maintainability issue that may arise in the near future.

Kreimer et. al. in his paper prospected a discernment hinged on decision tree [18] algorithm in which he diagnosed two imperfections, viz., long method and large class using Weka using predefined approaches without highlighting the precision of the data.

Khomh et. al. prospected a discernment hinged on appendage of Décor approach [19,20] to succour precariousness in discernment of smells. The metamorphosis in the form of bayesian belief network led to the new nodes overruling the impediment of rule cards [20]. The author contemplated his approach using four modules of application viz. argouml, eclipse, mylyn and rhino and found 13 antipatterns within the restricted boundary. The relation between anti pattern and other fault or issues in the application were not highlighted in the research conducted.

Hassaine et. al. correlated between human's unsusceptible program and discernment [22]. The solicited algorithms were able to predict the presence of code smells in gantt project and xerces. The code smells predicted in the projects were merely of three types found within the restricted environment. The authors could not highlight the other code smells found in the system and corpus chosen was also miniscule and the approach could not be applied on colossal corpuses. Oliveto et. al. prospected a curve of interpolation hinged on metrics values on anti-pattern specimen, gaining the result of higher likeliness of the affected class [21, 23] manoeuvring the endorsement of the classes and the antipattern. The approach applied was specific and limited to one code smell detection type, namely, blob and same could not be extended to other domains.

Maiga et al. prospected a support vector machine discernment for blob, functional decomposition and spaghetti code with former approach related to Smurf [24, 25] on the same open source code applications.

Palomba et al. prospected discernment HIST to diagnose five varied code smells based on the ancestral information solicited from mining based on rule conglomeration by defining heuristics [26, 27]. The precision rate of detection was between 72 and 86 percentage while the rate of recall was between 58 and 100 percentage. Code smell consists of a huge list and only one type of it was focusses on in the research conducted.

Fu and Shen et al. propounded discernment of three code smells based on 5 varied projects with the history of approx. 5-13 years and displayed the issue of no future versions of

the applications available to be fed into rule mining based on conglomeration [28].

Arcelli Fontana et al. ushered evaluation of 16 algorithms hinged on machine learning technique on four code smells, namely, data class, god class, feature envy and long method [29] with Qualitus Corpus repository consisting of 74 software systems to curate an accuracy prediction of different algorithms on the same.

Mauna Hadj et al. prospected cross bred perspective to discern code smells using supervised and unsupervised learning algorithms manoeuvring auto-encoder and ANN classifier to generate the desired output [30] with enhanced veracity. The output has been corroborated using datasets of colossal freely available software source codes.

Liu H. et al. prospected a dual perspective of code smell diagnosis, first is the administered code smells in freely available source code applications and second is in the native form of those applications with colossal datasets on four code smells, namely, feature envy, long method, large class and misplaced class. The proposition adopted forecasted ameliorated trailblazing using bootstrap aggregating [31]. The observations in the two perspectives were made as reduction in associating proposed approach in relation to the native approach of DÉCOR.

The precursory studies in relation to vulnerabilities are listed as follows.

Cao et al. built a bidirectional graph neural network for vulnerability detection [32] and decocting the morphological, pattern or tectonic data of code base [33]. Wang et al. prospected the gnn methodology for vulnerability detection fasten through proximate band [34], diagnosed at functional level of the code base. Batur et al. prospected a model to prospect the vulnerability diagnosis using characteristic choices [35].

Chakrobarty el al. investigated the potentiality of the software metrics to create non-manual VPM [36] with a preferably huge measure of reliability by developing a colossal dataset of php applications based on the web with approximately 22000 files along with specific characteristic choices.

Zagane et al. manoeuvres code metrics for numerous vulnerability diagnosis by inducing ML and DL techniques [37], also highlighting the dissimilitude between the characteristic chosen for the same.

Shuban et al. [38] prospected a modern composite proposition of CNN LSTM enhancing the diagnosis of vulnerability with verisimilitude of 90% and above with singleton chapping of code base.

Rebecca L Russel et al. exhibited the potency of the vulnerability detection based on C/C++ code blocks and curated it with SATE IV dataset with convolutional neural network approach[39]. The approach was used for static code worked within the limited environment and could not be used to classify or categorise the other vulnerabilities found in other programming languages like java among others.

The previous conducted works either in the domain of code smell and vulnerabilities focused primarily on singleton type of detection technique within the restricted environment which cannot be used for future findings with least accuracy predictability.

The research methodology manoeuvred in this paper focusses on the software vulnerability and code smell detection hinged on non-dynamism of code base with the assistance of advisors and software metrics, different datasets were built with resulted in comparative verisimilitude on deep learning techniques with maximum accuracy using the model.

Based on the previous studies, several gaps and limitations have been identified related to code smell and vulnerability detection which are addressed in the comprehensive methodology and experimental approach, as outlined below:

1. Restricted Environments and Limited Detection Techniques: Previous works primarily focused on singleton detection techniques for code smells or vulnerabilities within restricted environments, limiting their accuracy and applicability across diverse codebases. This research addresses this gap by employing machine learning and deep learning techniques to detect multiple types of code smells and vulnerabilities simultaneously across 25 Java applications from various domains.

2. Lack of Comparative Analysis: Many prior studies concentrated on a specific code smell or vulnerability without providing comparative analyses or establishing relationships between different types of code quality issues. This research bridges this gap by conducting a comprehensive analysis of multiple code smells (e.g., God Class, Long Method) and vulnerabilities (e.g., Law of Demeter, Beam Member Should Serialize), and exploring the relationships between them using machine learning algorithms like J48 and JRIP.

3. Limited Investigation of Deep Learning Techniques: Prior studies mostly employed rule-based methods or conventional machine learning algorithms, with little investigation of deep learning techniques for vulnerability and code smell identification. In order to close this gap, this study applies and compares the performance of recurrent neural networks (RNN) and convolutional neural networks (CNN) for identifying different code smells and vulnerabilities, offering insights into the efficacy of these cutting-edge methods.

4. Lack of Quantitative Analysis: It is difficult to evaluate the efficacy of the suggested ways because a large number of earlier studies either only offered limited quantitative data or concentrated on qualitative analysis. This study closes this gap by performing a thorough quantitative investigation and providing accuracy numbers for several deep learning and machine learning approaches across a range of code smells and vulnerabilities.

By addressing these gaps, this research contributes to the field of software quality analysis by providing a comprehensive framework for detecting code smells and vulnerabilities using advanced machine learning and deep

learning techniques. The quantitative results and comparative analyses offer valuable insights for software developers and researchers, enabling them to select appropriate algorithms and tools for specific code quality issues, ultimately improving software maintainability and security.

3 Various Tools Used

The varied tools used for convoying the experimental approach is listed in *fig 6.*

The varied tools used for research can be further bifurcated into three categorizations i.e. advisors, metrics and deep learning techniques. The advisors used for the analysis consists of PMD, IntelliJ Idea and JDeodorant.

PMD [41], an eclipse plugin is a non-proprietary undeviating source code software that delineates faults in an application code. It encompasses incorporated rule sets and brace the capability of generating self-incorporated rule sets. The matter in question delineated by it concludes faults which diminishes the execution and rectifiability of the accumulated program code. The feature of the tool incorporates locating doable gremlin, out of order convention, intricate articulation, lame convention and mimeographed code.

IntelliJ Idea[43], prepared in Java programming language is an IDE curating characteristics like intelligent consummation, shackles consummation, undeviating member completion, information flow probing, speech inoculation and predicting mimeographs in code. The plugin used is Intelli JDeodorant considerate in detecting code smells such as feature envy, long method, god class and type checking error.

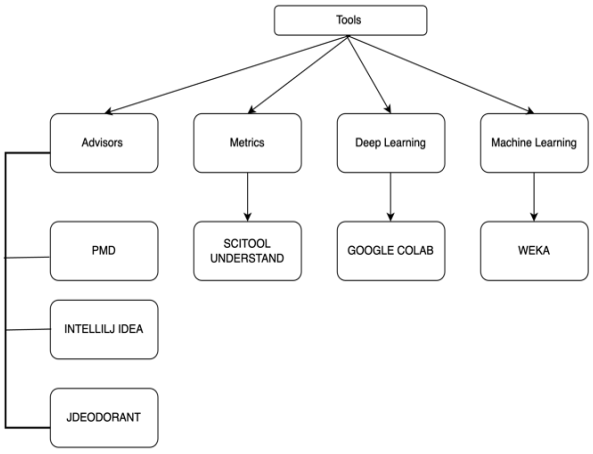

Fig 6. Tools used in research methodology.

JDeodorant[41], code smell detection as well as refactoring tool, is an eclipse plugin employs varied methodology and strategies so as to ascertain code smells and resolve them using refactoring. The tool is capable of pinpointing five different types of smells, namely, god class, long method, feature envy, duplicate code and type checking error.

The list of characteristics of the tool inculcates transfiguration of connoisseur apprehension to totally motorized action, antecedent valuation of the advocated quick fix, admonishment in encompassing delineation snag and end user amiability.

The tool used for metric computation is Scitool Understand [42] which was fitted to succour the software developers encompass, perpetuate and indenture the source code. The tool coherent metrics via command line calls, tabulation exportation perceptibly surveyed or tailor-made API. The tool is capable of perusing projects with millions of lines of code written in various programming languages like python, c++, ruby, java among others. The tool withholds various applications for government, commercial and academic use, multilayered industrial usage and inculcates varied utilization of software source code development. The tool used for deep learning implementation of algorithms is google colab, accelerates using cloud services provided by google, a free jupyter notebook with no premature essentialities to fulfil with multiple adjuvant libraries.

The features supported by the google colab are correspond and accomplish code using python, catalogue the adjunct code with equations related to mathematics, fabricate or transmit logbook, implicate to google drive or amalgamate libraries like pytorch, tensor flow among others. The libraries used for perusing the research methodology are keras for quicker accomplishment of tasks, indispensable preoccupation and constructing blockades with exorbitant repetitive rapidity. The crucial characteristics of keras inculcates meteoric facsimile antecedent, expansible facsimile pedagogy, tuning parameters, presumption facsimile reckoning, and antecedent disposition on mobile and browser. Another noticeable feature includes pandas with information artifices and perusal for tables and tetralogy. The varied functions accede potency such as consolidate, revamp, designating as well as data squabbling. Numpy, one of the basic conglomerations of the programming in python. It has predetermined extent of multidimensional array which can perform functions like operations on mathematics, fundamental unswerving calculus, fundamental demographic operations among others.

Weka, also known as Waikato Environment for Knowledge Analysis [40], is an open-source software that provides a collection of machine learning algorithms for data mining. It includes tools for data pre-processing, classification, regression, clustering, association rules, and visualization. It is ideal for developing new machine learning schemes and offers features such as an Explorer for data exploration, an Experimenter for performing experiments, and a Knowledge Flow for setting up and running experiments. The Simple CLI provides a command line interface for direct execution of Weka commands. The Explorer includes filters for discretization, normalization, resampling, attribute selection, transformation, and association rule mining. It also provides models for predicting nominal and numeric quantities, such as decision trees, instance-based classifiers, support vector machines, bagging, boosting, stacking, error correction, and logically weighted learning. The Cluster tool is used to find groups of similar instances in a dataset, and the Associations algorithm is used to learn association rules. The Attribute Selection tool searches through all possible combinations of attributes in

data and finds the best subset for prediction. Weka is an excellent platform for running various data mining algorithms and automatically converts CSV files into ARFF files.

4 Experimental Approach

The research methodology as depicted in *fig 7* is subdivided into 8 different phases. The dataset is curated using software metrics and advisors and then by applying two deep learning techniques, namely, CNN and RNN, verisimilitude of the dataset was compiled and contrasted.

4.1 Corpus Collection

Section I is the initiation phase. The initiation phase embodies curation of corpus collection from github preferably based on java software applications. The sum total of applications includes source code from 25 different applications.

4.2 Code smell and vulnerability detection

The Section II of the experimental approach embraces code smell and vulnerability detection using code smell and vulnerability confidante respectively. The code smells such as god class, feature envy, long method and duplicate code are detected using JDeodorant[14,15], PMD[13] and IntelliJ Idea[15]. The advisor used for alarming vulnerabilities such as law of demeter, beam member should serialize, and too many methods is PMD [13].

4.3 Software metrics computation

The Section III is the computation of software metrics using a tool called Scitool Understand [12]. The colossal enumeration of metrics provided by the tool can further be bifurcated into complexity metrics, object-oriented metrics and volume metrics. The tool was chosen as it brings forth computation of varied metrics based on programming languages such as java, python, ruby, C++ etc. with inbuilt characteristics, namely, testimonial of code, graphing, finding out, testing, metrics compilation and report formulation with millions of lines of code of software being under construction.

4.4 Formalizing Dataset

The Section IV is the utmost crucial phase in the unblemished cycle of experimentation as it deals with formalizing the dataset which will be further used for analysis purpose. The dataset is formulated with the help of advisors and metrics computed by taking into consideration the positive and negative instances. The dataset has been curated using stratified sampling approach [16] which is a process of dissecting the projection of the populace into congruent subspecies preceding the sampling procedure, then labelling based on positive or negative instances.

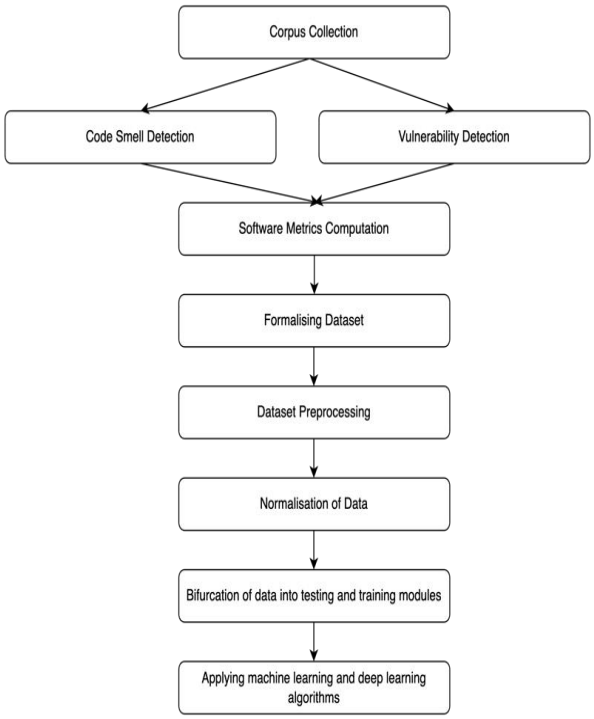

Fig 7 Research Methodology

4.5 Data Pre-processing

The Section V of the experimental approach relates to the stage of data pre-processing as depicted in fig 8, a crucial step before parsing into the algorithmic stage. Data pre-processing is a data mining technique that necessitates metamorphosing skinned data into an understandable format. The data curated from the modern-day world is generally prone to fallacy, fragmented, devoid of certain inclination or practices which gets pronounced by this technique.

Fig 8 Steps in data preprocessing

The *fig 8* mentions the steps taken to prepare dataset for analysis and verisimilitude prediction using google colab and weka by implementing methodologies such as CNN and RNN and many machine learning algorithms like J48, JRip, Naïve Bayes etc. and a comparison has been achieved hinged upon them. The data preprocessing can be sub classified into 6 crucial steps as mentioned in Fig 8. The process initializes with data cleaning, a process of pigeonholing the mislaid data or eradicating rows with mislaid data, flattening the clamorous data or straightening out the data at odds, the chances of getting it either through human fault or doubling of data. Data integration is a way of binding data with varied

delineation along with discord rectification. Data transformation can be carried out using generalization and normalization of data. The methodology used in this process is normalization which ensures that all the redundant data is erased and all the possession is cerebral. Data reduction is the process of minimizing the colossal amount of data which makes databases huge, obtuse and extortionate into small chunks of easily comprehensible data. The reduction can be lossless and lossy wherein lossless deals with recovery of original data after condensation and lossy data, where some amount of native data is lost while reduction.

Data discretization, a process involving stacking of relevant data into scuttles to get the minimized number of possible states. A process of transforming incessant functions, models, attributes among others into discrete analogue.Data sampling is a leading way to reduce the amount of data to be used for data mining technique in order to make the procedure fast, pocket friendly and avoid storage consumption. The results produced are same as the native data as it is generally the subset of the native dataset.

4.6 Normalization of Dataset

The Section VI of the experimental approach consists of normalization of data. The normalization of data is narrowing down of the range of attribute value into a lesser scale to prevent ineffectuality of critical data while data pre-processing and to bring the data on the kindred lamella.

4.7 Testing and Training of Data

The Section VII of the research methodology deals with bifurcation of data into testing and training modules [17] which acts as the key criteria for evaluation of model. The percentage of bifurcation is 80% for training and 20% for testing of data.

The training data depends on the choice of learning whether it is supervised or unsupervised learning. In unsupervised learning, the data is generally untagged, model prognosticate data based on the pattern involved. In case of supervised learning, the data is tagged, and model prognosticate inputs based on it.

The test dataset is an unfettered dataset which a part of representation from native dataset as well as colossal enough to make purposeful prognostication, containing data for each classification present in native data. The testing and training of dataset is an essential component to augment the accomplishment of the model and strengthen its verisimilitude.

4.8 Deep Learning and Machine Learning Technique

The Section VIII of the research methodology deals with machine learning and deep learning technology to be applied on dataset to answer the research questions. It manages running CNN as depicted in *fig 10* and RNN models as depicted in *fig 11* for the same using a cloud platform called Google Colab, a python IDE with varied inbuilt libraries like keras, pandas, numpy among others.The model was validated on the numerical values of training and testing of datasets with parameters such as batch size, epoch and verbose as viewed in figure 9 and 10 related to CNN and RNN modelling respectively.

Batch size is the aggregate number of data sites which can be passed in a singleton computational modelling i.e. it divides the colossal dataset into small batches or bifurcates the large one into small buckets of information for modelling. Epoch refers to the number of times the complete dataset passes the computational process i.e. progressing as well as rearward propagation.

The machine learning algorithms have been applied using a tool called as Weka using 10-fold cross validation as shown in *fig 9*. Cross validation is a resampling procedure used to evaluate machine learning models on a limited dataset. The procedure has a single parameter ‘k’ which refers to the number of groups that a given dataset is split into. The value of ‘k’ chosen is 10 which means that the dataset will be split into 10 parts and the result will be produced the 11^{th} time. This approach involves randomly dividing the set of observations into k groups of folds of approximately equal size. The first fold is treated as the validation set and the method is fit on the remaining k-1 folds. The rules are determined by applying a classifier filter of Weka on different machine learning algorithms.

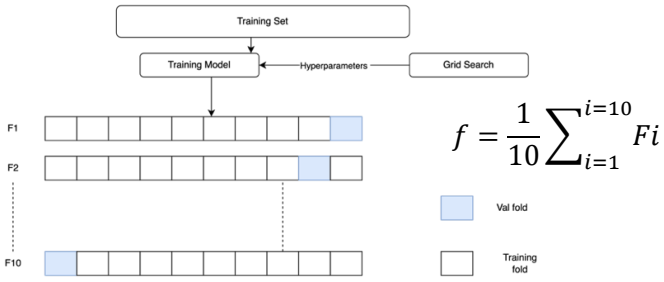

Fig 9: 10-fold cross validation

The result has been calculated based on parameters such as TP rate, FP rate, roc area, precision and kappa statistics obtained by applying machine learning algorithms through Weka were,

TP Rate: rate of true positives (instances correctly classified as a given class)
FP Rate: rate of false positives (instances falsely classified as a given class)
Precision: proportion of instances that are truly of a class divided by the total instances classified as that class
Recall: proportion of instances classified as a given class divided by the actual total in that class (equivalent to TP rate)
F-Measure: A combined measure for precision and recall calculated as 2 * Precision * Recall / (Precision + Recall)

5 Results and Discussion

RQ1: *Which machine learning algorithm produces best results for a particular vulnerability?*

To answer RQ1, a set of tools, namely, PMD, Scitool Understand and Weka was used wherein first 2 set of tools were used to curate datasets for individual vulnerabilities

where PMD acted as an advisor and Scitool understand helped in curating software metrics. Weka is used for applying machine learning algorithms based on supervised learning mechanism, generated the following results for different vulnerabilities.

Law of Demeter:

The algorithm JRIP produced the best results when compared with 81.51% shown in *fig 10*.

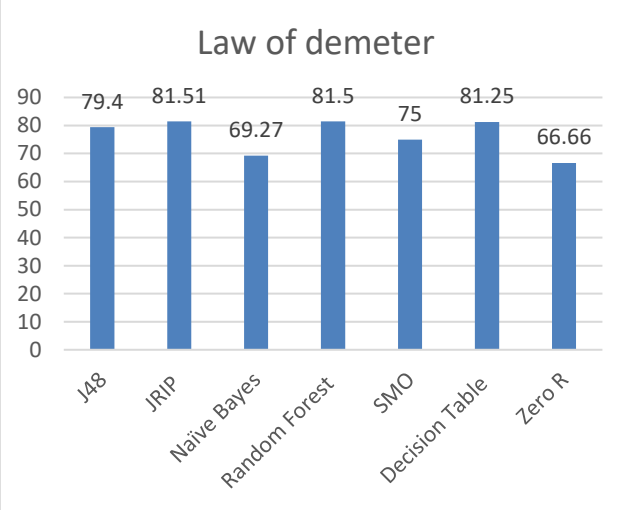

Fig 10: Algorithm comparison for law of demeter

Beam member should serialise.

The algorithm J48 produced the best results when compared with 96.2% as shown in in *fig 11*.

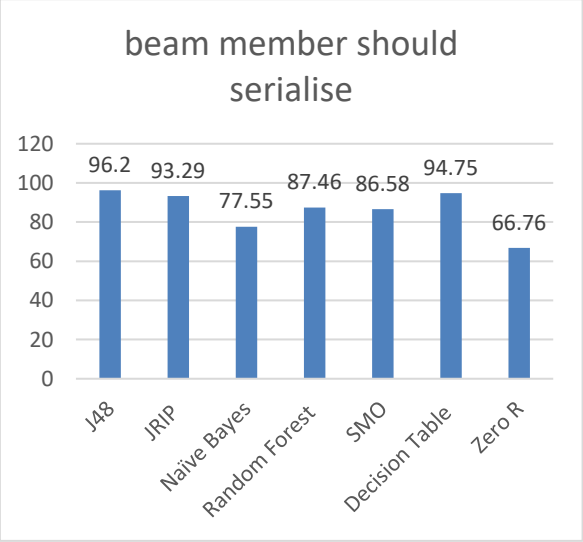

Fig 11: Algorithm comparison for beam member should serialize.

Npath complexity:

The algorithm JRIP produced the best results when compared with 99.07% shown in *fig 12*.

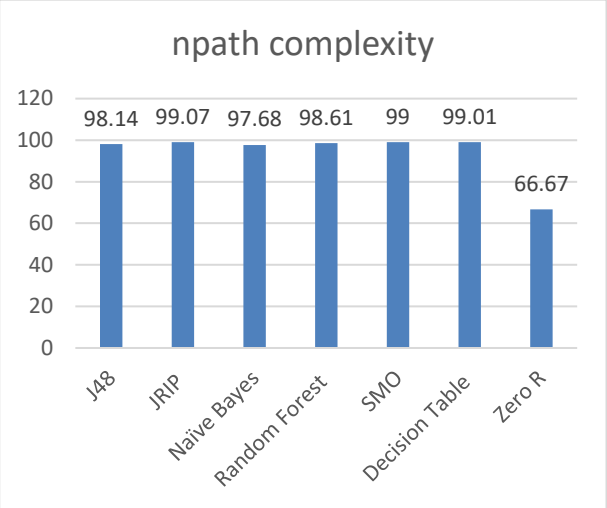

Fig 12: Algorithm comparison for Npath Complexity

Too many methods:

The algorithm JRIP produced the best results when compared with 97.093% shown in *fig 13*.

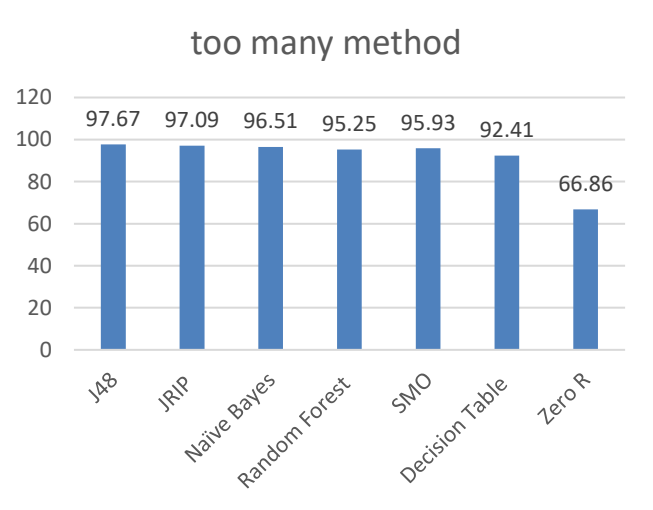

Fig 13: Algorithm comparison for too many method

Short Variable:

The algorithm Random Forest produced the best results when compared with 71.68% shown in *fig 14*.

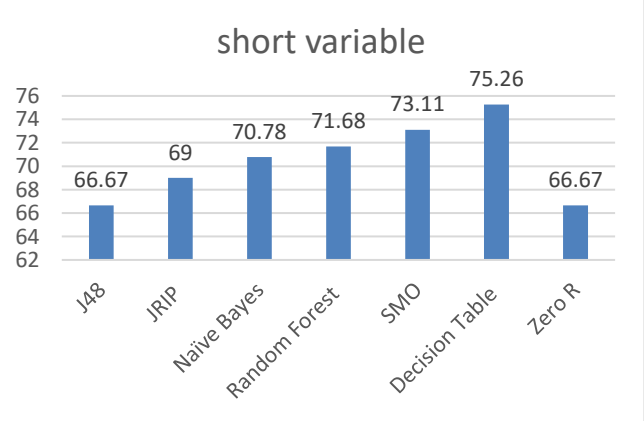

Fig 14: Algorithm comparison for short variable

Method argument could be final:

The algorithm JRIP produced the best results when compared with 75.86% shown in *fig 15.*

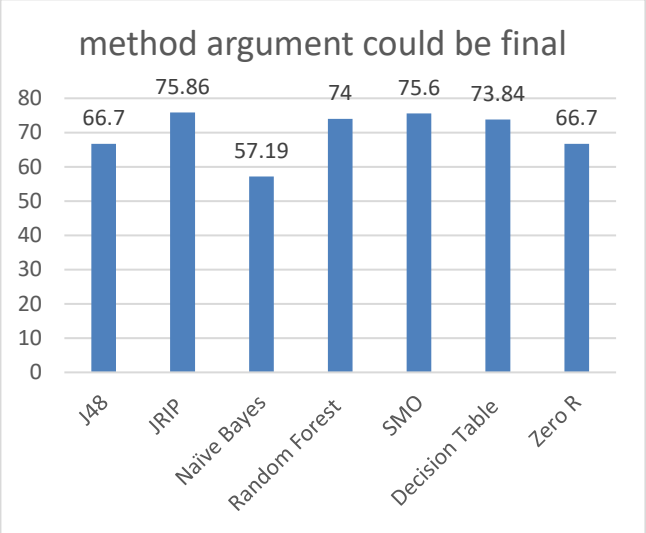

Fig 15: Algorithm comparison for method argument could be final

Local variable could be final:

The algorithm JRIP produced the best results when compared with 88.07% shown in *fig 16.*

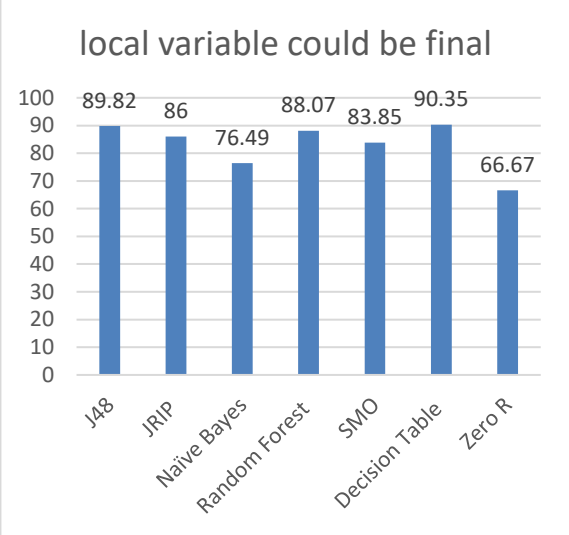

Fig 16: Algorithm comparison for local variable could be final

***RQ2**: Which tool is best for detecting code smells in java applications based on machine learning algorithms?*

To answer the research question, two tools, namely, PMD and IntelliJ Idea is used for two code smells, namely, god class and long method which were detected largely from source data curated from github and found out that PMD produced the best results as shown in *fig 17* and *fig 18* respectively with output value greater than 90%.

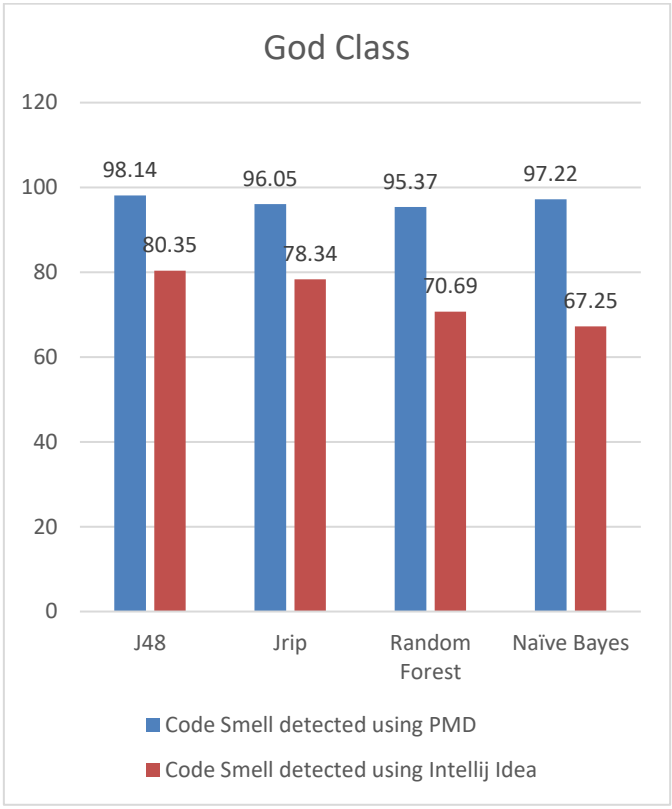

Fig 17: God Class result for two different software

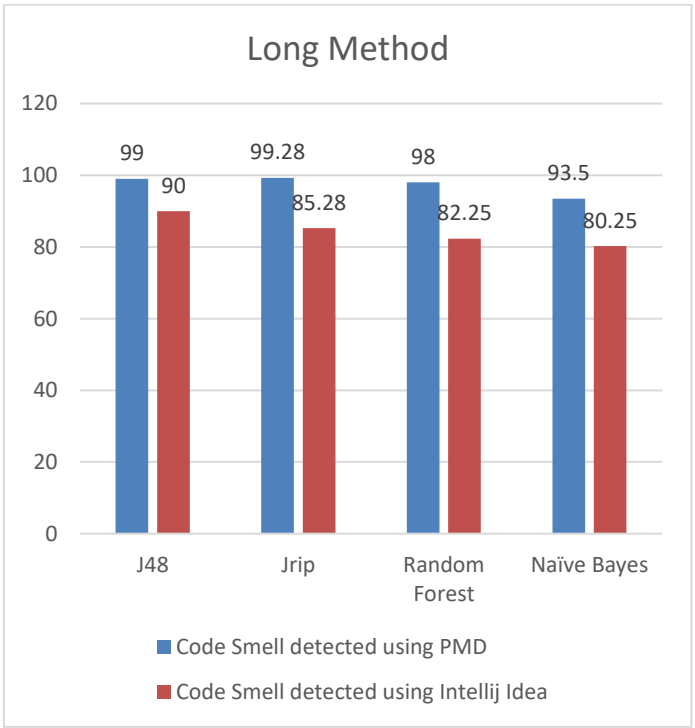

Fig 18: Long Method result for two different software

***RQ3**: Is there exists a similarity between code smell and vulnerability?*

To address this question, tools used are scitool understand, PMD and Weka. There exists a relationship between code smell and vulnerability. The violation pattern shown by both corresponds with one another. Not only in definition but, practically also they both are similar to each

other being two different terms with one meaning theoretically as well as practically. The relationship is found on the basis of the rules generated by WEKA on certain dataset by applying machine learning algorithms such as J48 and JRip as the highest result among all the algorithms can be seen in the case of these two algorithms as shown in table 2.

Table 2: Relationship between code smell and vulnerability

Code smell	**Vulnerability**	**Algorithm**	**Rule matched**
God class	Too many methods	JRIP	CountDeclMethod>=17
Cyclomatic complexity	Npath complexity	J48	SumCyclomaticStrict>8
Long method	Excessive method length	JRIP	CountLine>=80, SumCyclomatic >=11

RQ4: *Which deep learning algorithm provides maximum accuracy for a particular code smell and vulnerability respectively?*

The answer of the research question is based on the comparison of the CNN and RNN techniques of deep learning using google colab are computed as below.

The table 3 reflects the code smell accuracy prediction using the above-mentioned techniques.

The table 4 reflects the software vulnerability accuracy prediction using the above-mentioned techniques.

Table 3: Comparison of CNN and RNN techniques for code smells

Code Smell	Accuracy prediction using CNN	Accuracy prediction using RNN
God Class	90.08%	86.78%
Long Method	89.18%	81.08%

Table 4: Comparison of CNN and RNN techniques for vulnerabilities

Vulnerability	Accuracy prediction using CNN	Accuracy prediction using RNN
Law of Demeter	96.77%	91.39%
Beam member should serialize	85.50%	88.40%
Too many method	71.42%	94.28%
Cyclomatic Complexity	92.64%	80.82%

Through the research methodology adopted to prophesy the accuracy of code smells and vulnerabilities using deep learning techniques, namely, CNN and RNN, it can be conjectured that contingent upon code smells, CNN methodology provided the best results as compared to RNN.

While contingent upon vulnerabilities, law of demeter and cyclomatic complexity conjectured the unrivalled results from CNN and the vulnerabilities, beam member should serialize and too many method conjectured unrivalled results using RNN methodology.

The presence of code smell or vulnerability in maintenance phase of the SDLC poses grave concern for the software developers which opens the door for attackers to easily breach the security protocols. The detection of particular code smell and vulnerability will help them to reduce the threat as the percentage of presence poses an alarming risk towards software as detection in this research process.

6 Conclusion

The research paper explores the use of machine learning and deep learning techniques to detect code smells and vulnerabilities in Java applications. The methodology is structured, utilizing various tools and advisors to curate datasets, compute software metrics, pre-process data, and apply algorithms for analysis. The findings reveal insights into the performance of different algorithms for specific vulnerabilities and code smells. Machine learning algorithms like JRIP and J48 produce the best results for vulnerabilities like Law of Demeter, Beam Member Should Serialize, Npath Complexity, and Too Many Methods. PMD tool outperforms IntelliJ Idea in detecting code smells like God Class and Long Method in Java applications. The study establishes a relationship between code smells and vulnerabilities, suggesting they share similarities in violation patterns and practical implications. This aligns with the theoretical understanding that both code smells and vulnerabilities can negatively impact software quality and maintainability. The study compares the accuracy of Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN) for specific code smells and vulnerabilities. CNN outperforms RNN for certain code smells, while RNN provides better accuracy for some vulnerabilities. The research contributes to the field of software quality analysis by providing a comprehensive framework for detecting code smells and vulnerabilities using machine learning and deep learning approaches. Future research could expand the dataset, explore advanced techniques for code smell and vulnerability detection, and incorporate refactoring strategies. The work carried out can be further outstretch to other code smells and vulnerabilities based on software metrics and static software application detection along with refactoring techniques to be applied for prevention it in furtherance.

7 Threat to validity

The threats to validity as pointed out[16] has been overcome in this approach. The first threat was few applications with limited usage was selected which was solved by selecting applications from varied domains of software application code blocks.

The second threat of automatic generation or detection of code smells and vulnerabilities were handled by selecting classes and methods on random selection rather than automatic selection.

The third threat to validity [44] is that during the evaluation we re-implement the baseline approach first with Jdeodorant(eclipse) and second with Jdeodorant(IntelliJ

Idea) which could significantly influence the evaluation results.

The fourth threat to validity [44] is that code smells involved in the evaluation are generated automatically, so to reduce the threat we randomly select methods/classes etc.

The fifth threat to validity [44] is that the evaluation is based on the assumption that the involved software entities in the subject are well designed. However, the assumption may not hold, sometimes the calculation may be inaccurate.

Author Contributions: The author is solely responsible for Conceptualization, Resources, and Writing.

Data availability: Data available upon request.

Conflict of Interest: There is no conflict of Interest.

Funding: The research received no external funding.

Similarity checked: Yes.

References

[1] Kritika (2023). Correlating Propensity Between Code Smells and Vulnerabilities in Java Applications. International Journal of Scientific Research in Computer Science and Engineering, 11(1), 23-28.

[2] Sunita, S. (2021). An Overview of Deep Learning. International Journal of Engineering Research and Technology, 9(5).

[3] Shetty, D. H., Varma, M. J., Navi, S., & Ahmed, M. R. (2020). Diving Deep into Deep Learning: History, Evolution, Types and Application. International Journal of Innovative Technology and Exploring Engineering, 9(3).

[4] Indolia, S., & Goswami, A. K. (2018). Conceptual Understanding of Convolutional Neural Network -A Deep Learning Approach. International Conference on Computational Intelligence and Data Science, 679-688.

[5] Albawi, S., & Zawi, S. A. (2017). Understanding of a Convolutional Neural Network. International Conference on Engineering and Technology, 1-6.

[6] Liu, P., Qiu, X., & Huang, X. (2016). Recurrent neural network for text classification with multi-task learning. 25th International Joint Conference on Artificial Intelligence, arXiv-1605.

[7] Fontana, F. A., Zanoni, M., Marino, A., & Mäntylä, M. V. (2013). Code smell detection: Towards a machine learning-based approach. IEEE international conference on software maintenance, 396-399.

[8] Kovacevic, A., Slivka, J., & Vidakovic, D. (2022). Automatic Detection of Long Method and God Class code smells through neural source code embeddings. Expert Systems with Applications, 117607.

[9] Tahir, A., Counsell, S., & MacDonell, S. G. (2016). An empirical study into the relationship between class features and test smells. 23rd Asia-Pacific Software Engineering Conference (APSEC), 137-144.

[10] KS, V. K. (2019). A method for predicting software reliability using object-oriented design metrics. International Conference on Intelligent Computing and Control Systems (ICCS), 679-682.

[11] Elia, I. A., Antunes, N., Laranjeiro, N., & Vieira, M. (2017). An analysis of openstack vulnerabilities. 13th European Dependable Computing Conference (EDCC), 129-134.

[12] Kim, D. K. (2017). Finding bad code smells with neural network models. International Journal of Electrical and Computer Engineering, 7(6), 3613.

[13] Pessoa, T., Monteiro, M. P., & Bryton, S. (2012). An eclipse plugin to support code smells detection. arXiv preprint arXiv:1204.6492.

[14] Fokaefs, M., Tsantalis, N., Stroulia, E., & Chatzigeorgiou, A. (2011). Jdeodorant: identification and application of extract class refactorings. Proceedings of the 33rd International Conference on Software Engineering, 1037-1039.

[15] Felix, S. J., & Vinod, V. (2018). A study on different tools for code smell detection. International Journal of Computer Science and Engineering, 6(7), 762-764.

[16] Liu, H., Jin, J., Xu, Z., Bu, Y., Zou, Y., & Zhang, L. (2019). Deep Learning Based Code Smell Detection. IEEE Transactions on Software Engineering.

[17] Medar, R., Rajpurohit, V. S., & Rashmi, B. (2017). Impact of training and testing data splits on accuracy of time series forecasting in machine learning. International Conference on Computing, Communication, Control and Automation (ICCUBEA), 1-6.

[18] Kreimer, J. (2005). Adaptive detection of design flaws. Electronic Notes in Theoretical Computer Science, 141(4), 117-136.

[19] Khomh, F., Penta, M. D., Guéhéneuc, Y.-G., & Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change- and fault-proneness. Empirical Software Engineering, 17(3), 243–275.

[20] Khomh, F., Vaucher, S., Guhneuc, Y. G., & Sahraoui, H. (2009). A Bayesian Approach for the Detection of Code and Design Smells. Ninth International Conference on Quality Software, 305–314.

[21] Khomh, F., Vaucher, S., Guhneuc, Y.-G., & Sahraoui, H. (2011). Bdtex: A gqm-based bayesian approach for the detection of antipatterns. Journal of Systems and Software, 84(4), 559-572.

[22] Hassaine, S., Khomh, F., Gueheneuc, Y. G., & Hamel, S. (2010). IDS: An Immune-Inspired Approach for the Detection of Software Design Smells. Seventh International Conference on the Quality of Information and Communications Technology, 343–348.

[23] Oliveto, R., Khomh, F., Antoniol, G., & Gueheneuc, Y. G. (2010). Numerical Signatures of Antipatterns: An Approach Based on B-Splines. 14th European Conference on Software Maintenance and Reengineering, 248–251.

[24] Maiga, A., Ali, N., Bhattacharya, N., Saban, A., Guhneuc, Y. G., & Aimeur, E. (2012). SMURF: A SVMbased Incremental Anti-pattern Detection Approach. 19th Working Conference on Reverse Engineering, 466–475.

[25] Maiga, A., Ali, N., Bhattacharya, N., Saban, A., Guhneuc, Y. G., Antoniol, G., & Ameur, E. (2012). Support vector machines for anti-pattern detection. Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, 278–281.

[26] Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., Lucia, A. D., & Poshyvanyk, D. (2013). Detecting bad smells in source code using change history information. 28th IEEE/ACM International Conference on Automated Software Engineering, 268–278.

[27] Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., Poshyvanyk, D., & Lucia, A. D. (2015). Mining Version Histories for Detecting Code Smells. IEEE Transactions on Software Engineering, 41(5), 462–489.

[28] Fu, S., & Shen, B. (2015). Code Bad Smell Detection through Evolutionary Data Mining. ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 1–9.

[29] Arcelli Fontana, F., Mäntylä, M. V., Zanoni, M., & Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. Empirical Software Engineering, 21(3), 1143–1191.

[30] Hadj-Kacem, M., & Bouassida, N. (2018). A hybrid approach to detect code smells using deep learning. 13th International Conference on Evaluation of Novel Approaches to software engineering, 137-146.

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

[32] Cao, S., et al. (2021). BGNN4VD: constructing bidirectional graph neural network for vulnerability detection. Information and Software Technology, 136, 106576.

[33] Subhan, F., Wu, X., Bo, L., Sun, X., & Rahman, M. (2022). A deep learning-based approach for software vulnerability detection using code metrics. IET Software, 16(5), 516-526.

[34] Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., & McConley, M. (2018). Automated vulnerability detection in source code using deep representation learning. 17th IEEE international conference on machine learning and applications (ICMLA), 757-762.

[35] Mahesh, B. (2020). Machine learning algorithms-a review. International Journal of Science and Research (IJSR), 9(1), 381-386.

[36] Jiang, T., Gradus, J. L., & Rosellini, A. J. (2020). Supervised machine learning: a brief primer. Behavior Therapy, 51(5), 675-687.

[37] Van Engelen, J. E., & Hoos, H. H. (2020). A survey on semi-supervised learning. Machine Learning, 109(2), 373-440.

[38] Alloghani, M., Al-Jumeily, D., Mustafina, J., Hussain, A., & Aljaaf, A. J. (2020). A systematic review on supervised and unsupervised machine learning algorithms for data science. In Supervised and unsupervised learning for data science (pp. 3-21). Springer, Cham.

[39] Moerland, T. M., Broekens, J., Plaat, A., & Jonker, C. M. (2023). Model-based reinforcement learning: A survey. Foundations and Trends® in Machine Learning, 16(1), 1-118.

[40] Kirkby, R. (2002). WEKA Explorer User Guide for version 3-3-4. University of Waikato.

[41] Paiva, T., Damasceno, A., Padilha, J., Figueiredo, E., & Sant'Anna, C. (2015). Experimental evaluation of code smell detection tools.

[42] Kim, D. K. (2017). Finding bad code smells with neural network models. International Journal of Electrical and Computer Engineering, 7(6), 3613.

[43] Kurbatova, Z., Golubev, Y., Kovalenko, V., & Bryksin, T. (2021, November). The IntelliJ platform: a framework for building plugins and mining software data. 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), 14-17.

[44] Liu, H., Jin, J., Xu, Z., Bu, Y., Zou, Y., & Zhang, L. (2019). Deep Learning Based Code Smell Detection. IEEE Transactions on Software Engineering.