IJRAR.ORG E-ISSN: 2348-1269, P-ISSN: 2349-5138

INTERNATIONAL JOURNAL OF RESEARCH AND ANALYTICAL REVIEWS (IJRAR) | IJRAR.ORG

An International Open Access, Peer-reviewed, Refereed Journal

SOFTWARE DEFECT PREDICTION USING DEEP LEARNING AND MACHINE LEARNING ALGORITHMS

Mithilesh Janga, Dr. Ch. Satyananda Reddy

Andhra University, Computer Science and Systems Engineering Department, Visakhapatnam District, Andhra Pradesh

Abstract In order to proactively identify potential problems before they disrupt production systems, software defect prediction is essential. This abstract offer a thorough examination of a novel method for improving software defect prediction that combines the advantages of deep learning and machine learning. When they appear in production environments, software flaws can be expensive and disruptive. This project introduces a hybrid methodology that uses the strength of deep learning and machine learning to reduce these risks. To be more precise, we use Stack Sparse Autoencoders to extract useful features from the dataset, which are then used as inputs for classic machine learning algorithms like Gradient Boosting and Random Forest. We conduct numerous experiments to assess the efficiency of our hybrid strategy. We evaluate the predictive accuracy of the encoded features using datasets from a NASA repository that contain actual software defect data. The outcomes unmistakably show that our methodology produces better results than more established methods for defect prediction. Our models consistently achieve higher accuracy and offer useful insights into software defect identification by utilizing the rich feature representation learned by the deep learning component. Organizations can proactively identify software defects, priorities their remediation efforts, and improve overall software reliability by integrating deep learning and machine learning. Additionally, our method offers a useful framework for utilizing artificial intelligence's benefits in software development workflows.

Keywords: - SSAE, SMOTE, Random Forest, Gradient Boosting

1. Introduction

Making sure the software works well and doesn't cost too much to maintain is a major concern in today's world of computer program creation. By concentrating on "software defect prediction," a method intended to find potential problems in software systems before they become significant issues, this study addresses a significant challenge. This study aims to improve software defect prediction by utilizing a novel strategy that fuses the benefits of deep learning and machine learning techniques. This innovative combination of cutting-edge techniques has the potential to revolutionize defect detection and advance efficient software development practices.

The Importance of Software Defect Prediction: Consider software flaws as minor errors or bugs that can cause programs to behave oddly or perform incorrectly. . Later in the software development process, finding and fixing these issues can be time- and money-consuming. Software defect prediction becomes an essential tool in this situation. It aids in the early detection of potential issues, such as the discovery of a small dam crack before it results in a flood. Software can function more effectively, and we can save a lot of money by identifying and resolving these problems early on.

Deep learning and machine learning combined: We require clever tools to assist us in identifying issues as software grows more complex. These intelligent tools are similar to deep learning and machine learning. Deep learning is excellent at analyzing large amounts of data and identifying patterns. Making educated guesses based on patterns it has previously observed is a skill of machine learning. We are developing a super-smart system for foretelling software issues by combining deep learning and machine learning.

Stack Sparse Autoencoder's (SSAE) Function: The Stack Sparse Autoencoder (SSAE) is a vital component of our strategy. Consider searching for the most crucial pieces to a large jigsaw puzzle in order to solve it. SSAE uses software data in this way.

Similar to how you solve a puzzle by concentrating on the most crucial pieces first, it takes the most important components and makes sense of them. It makes a large amount of information easier to understand and is the most crucial component. It's similar to tracking down the crucial hints in a mystery. This assistant is adept at recognizing the complex patterns in the data, which makes it simpler to understand where issues could possibly arise. This makes the process much quicker and smarter and helps us identify potential problem areas.

Making Better Predictions Together: We employ the Random Forest (RF) and Gradient Boosting (GB) classifiers as two unique teamwork techniques to further enhance our software problem predictions. These teamwork techniques resemble the combining of many different ideas to make better educated guesses. The RF method creates numerous decision trees and combines their educated guesses. The GB method builds and enhances each subsequent poor guess until it produces a strong guess. Our prediction is even more precise and dependable thanks to these teamwork techniques.

Preparing by Preprocessing: Real-world data is a little disorganized, but we have tools to tidy it up before we foresee issues. To balance the data and make sure we're not concentrating on just one thing, we use a tool called SMOTE. It's like making sure a recipe has a balanced mixture of ingredients. Additionally, we use Min-Max scaling to ensure that all the data is within the same range, which helps our tools comprehend it better and enable faster prediction.

In this study, a novel and clever method for forecasting software issues is presented. The Stack Sparse Autoencoder, RF, and GB are used for understanding, along with deep learning and machine learning techniques, as well as smart preprocessing for data cleaning. By combining all these techniques, we hope to become adept at identifying issues before they have a significant impact.

2. Literature review

To increase the quality and dependability of software systems, researchers have been working on the challenging task of software defect prediction using machine learning and deep learning algorithms. These algorithms face several difficulties even though they have shown promise in the detection and prediction of defects. Let's look at some of the major issues and recommendations made by researchers:

2.1. Insufficient or Imbalanced Data:

([1] Wang & Yao, 2013) insufficient or unbalanced data as a problem. The crucial problem of data imbalance in software defect prediction is examined in this essay. In software defect prediction, unbalanced datasets are common because there are significantly more instances of non-defective code than defective code. The authors suggest using the Synthetic Minority Over-sampling Technique (SMOTE) to address this problem.

Drawbacks: SMOTE has drawbacks even though it is good at balancing datasets. The introduction of synthetic samples, which might not accurately reflect the underlying data distribution, is the main cause for concern. SMOTE's parameters must be carefully considered because improperly set values can cause over-generalization or the production of noisy samples. SMOTE might also not function at its best when dealing with datasets that have extremely imbalanced data distributions or complex data distributions.

2.2. Feature Selection and Engineering:

([2] Kakkar & Jain, 2016)In the context of predicting software defects, this paper offers a thorough review of feature selection techniques. It divides these methods into filter, wrapper, and embedded categories and discusses each one's advantages and disadvantages. [6](Muthukrishnan & Rohini, 2017)The goal of the study is to help researchers and practitioners choose the best feature selection methods for their individual defect prediction tasks.

Drawbacks: Although the review paper provides insightful analysis of feature selection techniques, it does not offer specific suggestions for picking the approach that is best suited for a given dataset or context. Its utility could be increased by providing clear guidelines for feature selection in software defect prediction.

2.3. Generalization and Overfitting:

([3] Thanapol et al., 2020) The problem of overfitting in software defect prediction models is discussed in this research paper. When a model learns noise from training data and struggles to generalize to new data, it is said to be overfit. The authors present regularization methods, such as L1 and L2 regularization, to address this issue. These methods encourage simpler models and penalize complex ones, increasing the generalizability of the models beyond the training set.

Drawbacks: Although the paper effectively addresses overfitting, it does not go into enough detail about the trade-offs related to determining the degree of regularization. Since it depends on various elements including the dataset's size, quality, and the complexity of the underlying data distribution, choosing the right level of regularization can be difficult.

2.4. Data Imbalance problems will affect the model performance."

([4] Song et al., 2019) the prediction of software bugs while preserving the best model performance. They conducted their research using the Naive Bayes and Random Forest machine learning algorithms. They used information from the publically available PROMISE repository. Their study's findings exposed the negative effects of data imbalance on model performance, highlighting the challenges in creating an effective defect prediction model.

2.5. Metrics: -

([5] Malhotra, 2015) concentrated on defect prediction using several datasets. Their study brought to light the prevalence of software defect metrics that are frequently used to evaluate the efficacy of prediction models. Accuracy, recall, precision, and AUC measures were a few of the frequently used metrics. Malhotra also added a "miscellaneous" category for metrics that were applied less frequently. Additionally, Malhotra's study evaluated several machine learning models for the task of software defect prediction.

3. Problem statement: -

Numerous techniques have been used to address the issue of inconsistent data, which can be broadly categorized as algorithmic or data-level solutions. Data sampling methods to measure class distributions, such as under sampling or oversampling, are included in data-level solutions. This article examines four oversampling experiences and focuses on the oversampling procedure.

3.1. Class Imbalance: -

To solve the problem of inconsistent data, many methods have been adopted, broadly classified as algorithmic or data-level solutions. Data-level solutions include data sampling techniques to measure class distributions, such as under sampling or oversampling. This article focuses on the oversampling process and discusses four experiences of the oversampling process.

3.1.1. Existing Oversampling Techniques (Other than SMOTE):

Borderline-SMOTE:

Summary: An addition to SMOTE called Borderline-SMOTE is intended to concentrate on the minority class's borderline instances. Only those instances that are close to the decision boundary are selected for the generation of synthetic samples.

ADASYN (Adaptive Synthetic Sampling):

An adaptive oversampling method called ADASYN creates more synthetic samples for challenging-to-classify minority class instances. The instances that are further away from the decision boundary are given more weight.

Safe-Level SMOTE:

Safe-Level SMOTE seeks to maintain the "safe" instances—those that are simple to correctly classify—while balancing the dataset. It avoids producing synthetic samples for instances that are safe in favor of concentrating on producing them for instances that are close to the borderline.

3.1.2. Drawbacks of Existing Oversampling Techniques:

Borderline-SMOTE:

Drawback: Borderline-SMOTE, while effective in generating synthetic samples for borderline instances, may not perform optimally when the dataset contains a mix of borderline and safe instances. It may not adequately handle the imbalanced distribution within the minority class.

ADASYN:

Drawback: ADASYN's adaptive approach can lead to an increase in computational complexity. The technique may require significant computational resources, especially when applied to large datasets.

Safe-Level SMOTE:

Drawback: Safe-Level SMOTE focuses on preserving safe instances but may not generate enough synthetic samples for borderline instances. This approach might not fully address the class imbalance issue when the dataset contains a substantial number of borderline cases.

3.1.3. How SMOTE Overcomes the Drawbacks of Existing Oversampling Techniques:

Handling Various Cases: Standard SMOTE provides a balanced approach to oversampling by generating synthetic samples for all minority class instances, regardless of their proximity to the decision boundary. This makes it suitable for datasets with varying levels of imbalance and complexity.

Reduced Complexity: Compared to ADASYN, which adapts to the difficulty of classification, SMOTE typically has lower computational complexity. It generates synthetic samples uniformly, making it more computationally efficient for many practical applications.

Balanced Approach: SMOTE's balanced approach ensures that all minority class instances receive synthetic samples, including borderline instances. This helps in addressing class imbalance comprehensively, even when the dataset contains a mix of safe and borderline instances.

In summary, while Borderline-SMOTE, ADASYN, and Safe-Level SMOTE provide specialized approaches for oversampling, they come with certain drawbacks related to handling complex cases, computational complexity, and the preservation of specific instance types. SMOTE, on the other hand, offers a balanced and versatile oversampling technique that can effectively address class imbalance in a wide range of scenarios.

3.2. Deep Learning Models

3.2.1 Existing Deep Learning Algorithms:

CNNs (Convolutional Neural Networks): CNNs are specialized neural networks created for processing grid-like data, particularly images. To extract hierarchical features, they use convolutional layers, and they are widely used in computer vision tasks.

Long Short-Term Memory networks (LSTMs):

LSTMs are a type of recurrent neural network (RNN) designed for sequential data, such as time series and natural language. They excel in capturing long-term dependencies.

Deep Neural Networks (DNNs):

DNNs are general-purpose neural networks with multiple hidden layers, used for various tasks like classification, regression, and feature extraction.

3.2.2 Drawbacks of Existing Deep learning Models: -

Convolutional Neural Networks (CNNs):

Drawbacks: CNNs require large amounts of labeled data for training, and they might overfit when data is limited. They may not be suitable for non-grid data and sequential data.

Long Short-Term Memory networks (LSTMs):

Drawbacks: Training LSTMs can be computationally expensive and slow. They suffer from vanishing and exploding gradient problems and require substantial data for effective training.

Deep Neural Networks (DNNs):

Drawbacks: DNNs need substantial labeled data, and they can overfit, especially in high-dimensional settings. They may not handle sequential or grid data as effectively as specialized models.

3.2.3 How SSAE Overcomes the Drawbacks of Existing Deep learning models:

Sparse Representation: SSAE encourages sparse feature representations, capturing essential data characteristics while discarding noise.

Unsupervised Learning: SSAE can learn from data without requiring labeled examples, making it suitable for tasks with limited labeled data.

Data Compression: SSAE naturally compresses data, reducing storage requirements and enhancing processing speed.

Versatility: SSAE can be applied to various data types, including sequential, unstructured, and structured data.

In summary, while CNNs, LSTMs, and DNNs excel in specific domains, SSAE stands out for feature extraction and data compression, especially in scenarios with limited labeled data, where sparsity and unsupervised learning are essential. SSAE's ability to create compact, informative data representations makes it valuable for reducing dimensionality and improving data analysis efficiency. The choice of algorithm should consider the specific task and data characteristics.

3.3. Machine learning classifiers

Machine learning classifiers are algorithms that learn patterns and relationships in data to make predictions or decisions. Here are three common classifiers and their advantages:

3.3.1 Existing Machine Learning Classifiers and Their Advantages

Logistic Regression:

Simplicity and Interpretability: Logistic regression is easy to understand and interpret, making it a great choice for beginners and when model interpretability is essential.

Efficiency: It works well with large datasets and training times are relatively quick.

Linear Separability: Effective when the decision boundary is approximately linear.

Support Vector Machines (SVM): -

Effective in High-Dimensional Spaces: SVM can handle datasets with a high number of features and find complex decision boundaries.

Robust to Outliers: SVM is less sensitive to outliers compared to some other classifiers.

Naive Bayes: -

Simple and computationally efficient.

Works well with high-dimensional data.

3.3.2 Drawbacks of Existing Machine Learning Classifiers

While existing machine learning classifiers offer advantages, they also present challenges when applied to software defect prediction:

Limited Feature Representation:

Some classifiers may struggle to effectively represent complex software features, leading to reduced prediction accuracy.

Imbalanced Data:

Software defect datasets often have imbalanced classes, making it difficult for classifiers to learn the minority class adequately.

High Dimensionality:

In software defect prediction, the feature space can be large and noisy, making it challenging for some classifiers to discern relevant patterns.

Sensitive to Hyperparameters:

Many classifiers are sensitive to hyperparameters, requiring fine-tuning for optimal performance, which can be time-consuming and resource-intensive.

3.3.3 Overcoming Challenges Using Random Forest and Gradient Boosting Classifiers

To address the challenges posed by existing classifiers in software defect prediction, leveraging Random Forest and Gradient Boosting classifiers can be beneficial:

Random Forest:

Random Forest handles high dimensionality well, effectively dealing with noisy and irrelevant features in software defect prediction.

Its ensemble nature helps in mitigating overfitting and improving prediction accuracy, especially with imbalanced datasets.

Parameter tuning in Random Forest is relatively straightforward, making it easier to optimize performance.

Gradient Boosting:

Gradient Boosting is resilient to overfitting and performs well even with complex feature spaces common in software defect prediction.

It excels in handling imbalanced data by giving more weight to misclassified samples, thus enhancing prediction of the minority class.

Through ensemble learning and weak learner integration, Gradient Boosting can effectively model software defect patterns and achieve high prediction accuracy.

By leveraging these advanced classifiers, we can enhance software defect prediction and address the challenges posed by traditional classifiers.

4. Proposed Model: -

Our proposed model for software defect prediction combines the power of deep learning and machine learning algorithms. The primary algorithm we employ is the Stack Sparse Autoencoder (SSAE) for effective feature extraction and dimensionality reduction. SSAE helps capture the underlying structure and patterns in software code, enabling the identification of potential defects.

We applied two well-known machine learning classifiers, Random Forest (RF) and Gradient Boosting (GB), to predict software defects. While GB iteratively develops weak learners into powerful predictors, RF is an ensemble learning approach that uses decision trees to make predictions. Our model can successfully handle complex data and make precise predictions by incorporating RF and GB.

A.) Insufficient or Imbalanced Data:

To tackle the challenge of data imbalance, we utilize the Synthetic Minority Over-sampling Technique (SMOTE).

SMOTE generates synthetic instances for the minority class by interpolating between existing minority class samples. It selects a minority sample and finds its k-nearest neighbors (usually k=5) in feature space. Then, it creates new synthetic samples by taking linear combinations of the selected sample and its neighbors.

SMOTE generates synthetic instances of the minority class, balancing the dataset and preventing bias towards the majority class. This technique enhances the model's ability to accurately detect defects in software systems.

B.) Feature Selection and Engineering:

Furthermore, we employ data preprocessing techniques to optimize the model's performance. Specifically, we use min-max scaling to normalize the input features. Min-max scaling transforms the feature values to a common range, mitigating the impact of varying scales and improving convergence during training.

C.) Overfitting: Overfitting occurs when a model learns to perform very well on the training data, but it does so at the cost of performing poorly on unseen data. In other words, an overfit model learns the noise and random fluctuations in the training data, rather than the true underlying patterns. This often leads to a model that is overly complex and highly tailored to the training data, and as a result, it fails to generalize to new data. Overfitting is a common pitfall in

machine learning, and it can be detrimental to a model's performance.

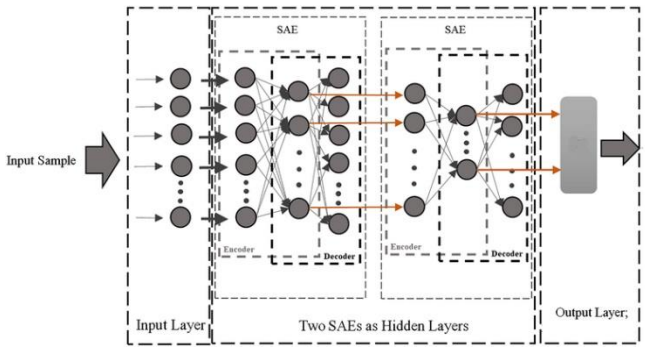
D.) Regularization:

Regularization is a method for avoiding overfitting by introducing a penalty term to the loss function of the model. The model is encouraged to have smaller parameter values by this penalty, which deters it from fitting the training data too closely.

G.) Hyperparameter Tuning: Optimize hyperparameters for each component (SSaE, RF, and GB) to achieve the best performance.

Through the integration of SSAE for feature extraction, RF and GB for classification, SMOTE for data imbalance, and min-max scaling for data preprocessing, our proposed model offers a comprehensive approach to software defect prediction.

4.1 Architecture: -



Stacked sparse auto encoder.

A sparse autoencoder is a type of artificial neural network that learns to encode input data into a compressed representation and then decode it back to its original form. The sparsity constraint encourages the autoencoder to use only a small number of neurons in the hidden layer, resulting in a more efficient representation.

Here's an example architecture for a stack of sparse autoencoders:

Input layer: This layer represents the input data to be encoded. Its size depends on the dimensionality of the input data.

Encoding layers: These layers consist of the hidden layers of the autoencoder. Each layer takes the output of the previous layer and applies a non-linear activation function to create a compressed representation of the input data. The number of neurons in each encoding layer progressively decreases to create a bottleneck effect, forcing the autoencoder to capture the most salient features.

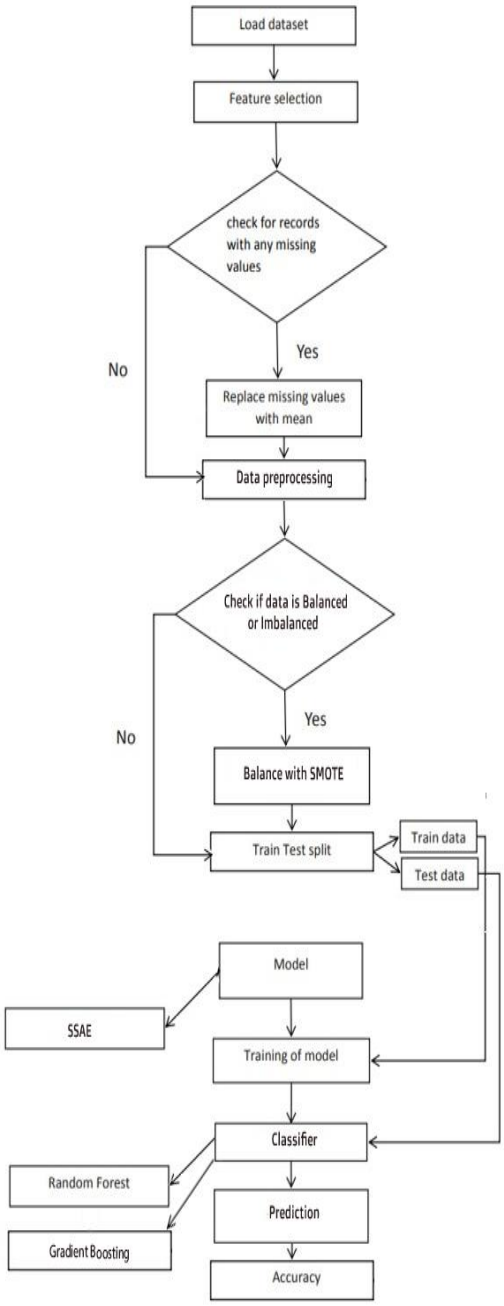
Sparsity constraint: The sparsity constraint is typically enforced by adding a penalty term to the loss function of the autoencoder. This penalty encourages the network to have a small number of active neurons in the hidden layers, promoting sparse representations.

Decoding layers: These layers mirror the encoding layers but in reverse order. They take the compressed representation and reconstruct the original input data by applying another set of non-linear activation functions.

Output layer: This layer produces the reconstructed output, which should ideally match the original input.

The stack of sparse autoencoders can be trained layer by layer in a greedy manner. Each layer is pretrained as an autoencoder independently, with the previous layers frozen. Once all the layers are pretrained, the entire stack can be fine-tuned using backpropagation.

4.2 System Architecture



System Architecture

4.3 Datasets

S.NO	DATASET NAME	TOTAL NO. OF FEATURES/TOTAL NO. OF INSTANCES	NO. OF DEFECTIVE INSTANCES	NO. OF NON - DEFECTIVE INSTANCES	% AGE OF DEFECTIVE INSTANCES
1	KC1	22/ 2109	178	1931	12.72%
2	KC2	37/ 1585	16	1569	1.01%
3	PC1	38/ 759	61	698	8.04%
5	CM1	38/ 344	42	302	12.21%
6	JM1	22/ 9593	1759	7834	18.34%

Dataset details

For our project software defect prediction, we taken our datasets from the NASA repository, The NASA PROMISE Repository is a public dataset repository that provides a collection of software

engineering datasets for research purposes. The repository was established in 2006 as part of the NASA Software Engineering Laboratory's research activities and contains datasets from various domains of software engineering.

The datasets are designed to support the development and evaluation of software engineering techniques, including software defect prediction, software effort estimation, software quality assurance, and software maintenance. The PROMISE repository currently contains over 50 datasets from various software engineering domains. The datasets are collected from publicly available sources, such as open-source software repositories, bug tracking systems, and software development projects. Each dataset includes a set of features, such as lines of code, number of developers, and complexity metrics, and a target variable, such as the number of defects, the effort required, or the quality of the software. In our project we are using KC1, KC2, PC1, JM1, and CM1 and those are all software defect prediction datasets.

- **KC1: -** The KC1 dataset contains data on software modules from a large telecommunications system developed in C++. The dataset includes 2109 module descriptions, and each module has 22 features that capture the size, complexity, and object-oriented design properties of the module. The target variable is whether the module contains faults or not.

- **KC2:** - The KC2 dataset contains data on software modules from a NASA software project. The dataset includes 522 module descriptions, and each module has 22 features that capture the size, complexity, and objectoriented design properties of the module. The target variable is whether the module contains faults or not.

- **PC1:** - The PC1 dataset contains data on software modules from a commercial software project developed in C#. The dataset includes 1109 module descriptions, and each module has 23 features that capture the size, complexity, and object-oriented design properties of the module. The target variable is whether the module contains faults or not.

- **JM1:** - The JM1 dataset contains data on software modules from a software project developed in Java. The dataset includes 10885 module descriptions, and each module has 21 features that capture the size, complexity, and object-oriented design properties of the module. The target variable is whether the module contains faults or not.

- **CM1: -** The CM1 dataset contains data on software modules from a NASA software project. The dataset includes 498 module descriptions, and each module has 21 features that capture the size, complexity, and objectoriented design properties of the module. The target variable is whether the module contains faults or not.

4.4 Data cleaning preprocessing

In order to prepare data for analysis or machine learning tasks, preprocessing is a crucial step. To make raw data more reliable, consistent, and compatible with the chosen algorithm, it must be transformed and cleaned. Two frequently used statistical measures in data preprocessing are mean and median. Here is a quick explanation of how to use them:

Mean:

The mean is the average of a set of numbers and is calculated by summing up all the values and dividing the sum by the total number of values. The mean is used to measure the central tendency of a dataset.

In data preprocessing, the mean can be used for various purposes, such as:

Handling missing values: You can replace missing values with the mean value of the feature. This ensures that missing data doesn't significantly impact the overall statistics of the dataset.

Feature scaling: A common method to normalize features, giving them zero mean and unit variance, is to subtract the mean from each data point and divide by the standard deviation.

Median:

The middle number in a sorted list of numbers is known as the median. The median is the average of the two middle values when the number of values on the list is even. When there are outliers or skewed distributions in a dataset, the median is used to describe the typical value.

In data preprocessing, the median can be used for tasks such as:

Handling outliers: Outliers can significantly impact the mean, making it less representative of the data. In such cases, using the median can provide a more robust measure of central tendency.

Imputing missing values: Instead of using the mean to fill in missing values, the median can be used as an alternative central tendency index. This method is helpful when the distribution of the data is skewed or when the mean may be significantly affected by outliers.

The use of mean and median depends on the characteristics of the dataset and the particular specifications of the analysis or model training. Both have their applications in various scenarios.

4.5 Data transformation

Data transformation is essential for predicting software defects. It entails transforming unprocessed data into a form that can be used for modeling and analysis. Here are a few typical data transformation methods for predicting software defects:

Feature Extraction: This technique involves selecting or deriving relevant features from the raw data that can capture the characteristics of software defects. These features can include code complexity metrics, code churn (changes made to the code over time), historical defect data, and developer expertise metrics.

Normalization: Scaling numerical features to a standardized range, usually between 0 and 1, is the process of normalization. It makes sure that the analysis is not dominated by different features with different scales. Techniques like min-max scaling or z-score normalization can be used for normalization.

Min-Max scaling: -

Data transformation methods like normalization and min-max scaling are frequently used to scale numerical features to a standardized range. It makes sure that various features' values fall within a predetermined range, usually between 0 and 1, enabling fair comparisons between variables with various scales.

The following is the min-max scaling formula:

In this formula, X_scaled is equal to (X - X_min) / (X_max - X_min), where X_scaled denotes the feature's scaled value,

X represents the feature's initial value,

X_min is the feature's lowest value across the dataset,

The feature's maximum value for the dataset is represented by X_max.

You must determine the minimum and maximum values for each feature in the dataset before applying min-max scaling. The feature's original values can then be converted to their corresponding scaled values using the formula.

The scaled values that are produced will be between 0 and 1. The scaled value will be 0 if a value is equal to the minimum value of the feature. The scaled value will be 1 if a value is equal to the maximum value of the feature. Within this range, values between the minimum and maximum will be scaled proportionally.

Missing values are frequently present in datasets taken from the real world. Using methods like mean imputation (replacing missing values with the feature's mean), median imputation, or regression imputation (forecasting missing values based on other variables), missing values can be imputed.

Dimensionality Reduction: When a dataset has a lot of features, it is possible to use dimensionality reduction techniques to cut down on the number of variables while still keeping the most crucial data. This can be achieved by using methods like Principal Component Analysis (PCA) or feature selection algorithms (such as Recursive Feature Elimination).

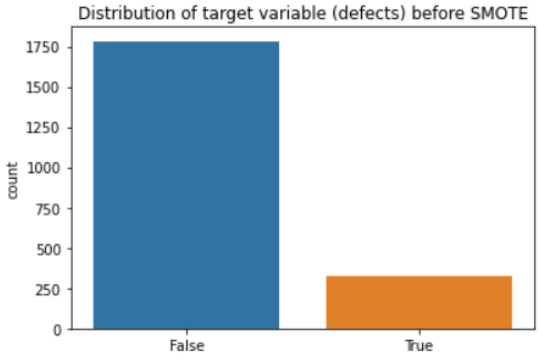
Managing Unbalanced Data: Software defect prediction datasets frequently experience class imbalance, where the proportion of defective instances compared to non-defective instances is disproportionately low. To balance the dataset and avoid biased predictions, strategies like oversampling the minority class (defective instances) or undersampling the majority class (non-defective instances) can be used.

4.6 Data visualization: -

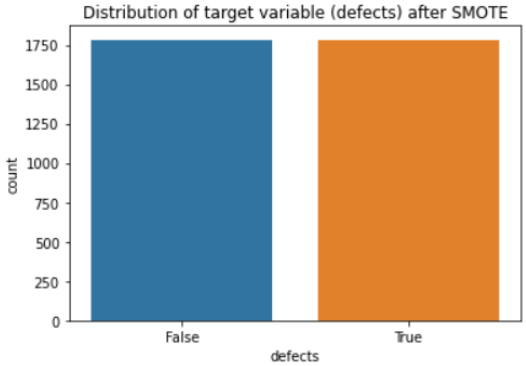
The graphic representation of information and data is known as data visualization. In order to help people comprehend and make sense of massive amounts of data, data visualization is a technique that makes use of a variety of static and interactive visuals within a specific context. In order to visualize patterns, trends, and correlations that might otherwise go unnoticed, the data is frequently presented in a story format. Data visualization is frequently employed as a means of commercializing data. To analyze vast amounts of data and make data-driven decisions, data visualization tools and technologies are crucial in the world of big data. Colors and patterns are appealing to human eyes.

Red and blue are easily distinguishable, as are square and circle. Another form of visual art that captures viewers' attention and keeps them focused on the message is data visualization. A person can quickly identify trends and outliers when viewing a chart. A person can internalize something quickly if they can see it. It is narrative with a goal. Numerous methods exist for displaying data, including histograms, density plots, correlation matrix plots, pie charts, etc.

4.6.1 Minority and majority class before SMOTE.

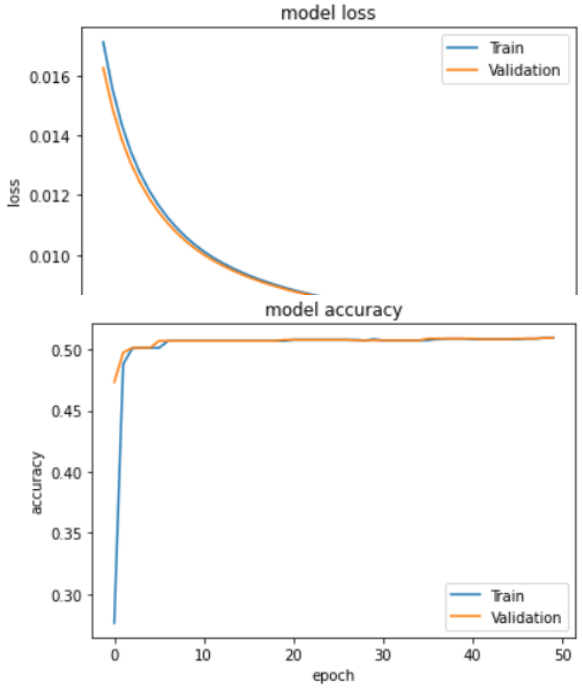


4.6.2 Minority and majority class after SMOTE.



4.6.3 Model accuracy

4.6.4 Model Loss



4.7 Algorithms

4.7.1 SSAE(Stack sparse auto-encoders)

Stacked Sparse Autoencoder Algorithm:

Input:

X: Features of the input data are represented by a matrix with size m x n, where m denotes the sample count and n the number of features.

layer_sizes: The total number of neurons in each layer is represented by a list of integers. activation process Utilizing an activation function for each layer

sparsity_constraint: Sparsity regularization parameter.

learning_rate: Learning rate for optimization.

num_epochs: Number of training epochs for each layer.

batch_size: Batch size used during training.

Output:

Encoded_features: Extracted and learned features from the dataset.

For each layer in layer_sizes:

Initialize encoder weights W^l and biases b^l with appropriate dimensions.

Initialize decoder weights W'^l and biases b'^l to match the encoder dimensions.

Training Loop (num_epochs times):

For epoch in 1 to num_epochs:

Initialize the average sparsity term avg_sparsity[l] as 0.

For each batch of size batch_size in the training data:

Forward Pass (Encoding):

Compute the encoded representations z^l using the encoder:

z^l = activation_function(X_batch @ W^l + b^l)

Sparsity Constraint:

Calculate the average activation for each neuron over the batch:

avg_activation = (1 / batch_size) * sum(z^l)

Update avg_sparsity[l] as the exponential moving average of avg_activation.

Reconstruction Loss:

Compute the reconstruction recon_X by decoding z^l using the decoder:

recon_X = activation_function(z^l @ W'^l + b'^l)

Calculate the mean squared error (MSE) reconstruction loss:

loss = (1 / batch_size) * sum((X_batch - recon_X)^2)

Backpropagation and Update:

Calculate the loss gradients with respect to the biases and weights of the encoder and decoder.

By utilizing the gradients and learning rate, update the encoder and decoder weights and biases:

W^l += learning_rate * gradient_W^l

b^l += learning_rate * gradient_b^l

W'^l += learning_rate * gradient_W'^l

b'^l += learning_rate * gradient_b'^l

Encoding Data:

Pass the entire dataset X through the trained encoder of the current layer to obtain the encoded features encoded_features.

Output:

- Encoded_features: The final encoded features obtained after passing through all the layers.

4.7.2 SMOTE Algorithm (Synthetic Minority Oversampling Technique):Input:

X: Features of the dataset (matrix of size m x n).

y: Labels of the dataset (vector of size m).

k_neighbors: Number of nearest neighbors for synthetic sample generation (k_neighbors >= 3).

sampling_strategy: Desired ratio of the number of synthetic samples to the number of original samples.

Output:

X_resampled: Oversampled features (matrix of size m' x n).

y_resampled: Oversampled labels (vector of size m').

Steps:

1. Count the number of instances in each class.

2. Identify the minority class samples and majority class samples.

3. Calculate the number of synthetic samples to generate per minority instance:

For each minority instance, generate (sampling_strategy - 1) synthetic samples.

4. Initialize empty lists for the synthetic samples and their corresponding labels.

5. For each minority instance at index i:

a. Find the k_neighbors nearest neighbors of instance X[i].

b. For each synthetic sample to generate:

i. Choose a random neighbor index j from the k_neighbors.

ii. Calculate the difference vector between X[i] and X[j]: diff_vector = X[j] - X[i].

iii. Generate a random number alpha between 0 and 1.

iv. Create the synthetic sample: synth_sample = X[i] + alpha * diff_vector.

v. Add synth_sample to the list of synthetic samples.

vi. Add the corresponding label y[i] to the list of labels for synthetic samples.

6. Stack the original minority samples with the generated synthetic samples to create X_resampled.

7. Stack the corresponding minority labels with the labels for synthetic samples to create y_resampled.

8. Return X_resampled and y_resampled.

4.7.3 RF (Random Forest)

Random Forest Algorithm:

Input: Training dataset D, num_trees, max_depth, num_features_per_tree

Output: Random Forest ensemble of decision trees

Procedure:

- Initialize an empty ensemble forest.

- For each tree_i in num_trees:

- Create a bootstrap sample bootstrap_sample from D with replacement.

- Randomly select num_features_per_tree features from the available features.

- Train a decision tree tree on bootstrap_sample with selected features and max_depth.

- Add tree to the forest.

- Return the forest.

Prediction using Random Forest:

Input: Random Forest ensemble forest, sample to predict

Output: Predicted class label or value

Procedure:

For each tree in the forest:

Traverse the tree by comparing features of the sample with internal node features.

Once a leaf node is reached, return its predicted class label (classification) or value (regression).

Aggregate predictions from all trees (e.g., majority voting for classification, averaging for regression) for the final prediction.

Key Equations (Gini Impurity):

- Gini impurity at node N:

Gini(N) = 1 - Σ(p_i^2), where i is the class index, p_i is the probability of class i.

- Gini impurity after a split using feature f and threshold t:

Gini_split(f, t) = (N_left / N_total) * Gini(left) + (N_right / N_total) * Gini(right),

where N_left and N_right are the numbers of examples in the left and right subsets after the split,

and N_total is the total number of examples in the node.

4.7.4 Gradient Boosting

Gradient Boosting Algorithm:

Input: Training dataset D = {(x_i, y_i)}, num_iterations, base learner h(x; θ), loss function L(y, F(x)), learning rate η

Output: Ensemble of weak learners { F_m(x) }

Procedure:

1. Initialize predictions F_0(x) for all x to a constant value (e.g., the mean of target values).

2. For m in range num_iterations:

a. Compute pseudo-residuals r_i^(m) for each example (x_i, y_i):

$r_i^{(m)} = -[\partial L(y_i, F_{m-1}(x_i)) / \partial F_{m-1}(x_i)]$

b. Fit a base learner h(x; θ) to predict r_i^(m) by minimizing the loss function:

$\theta_m = \text{argmin}_\theta \Sigma_i L(y_i, F_{m-1}(x_i) + h(x_i; \theta))$

c. Calculate the step size for the update:

$\gamma_m = \text{argmin}_\gamma \Sigma_i L(y_i, F_{m-1}(x_i) + \gamma * h(x_i; \theta_m))$

d. Update the ensemble prediction:

$F_m(x) = F_{m-1}(x) + \eta * \gamma_m * h(x; \theta_m)$

3. Return the ensemble of weak learners { F_m(x) }.

Prediction using Gradient Boosting:

Input: Ensemble of weak learners { F_m }, sample to predict x

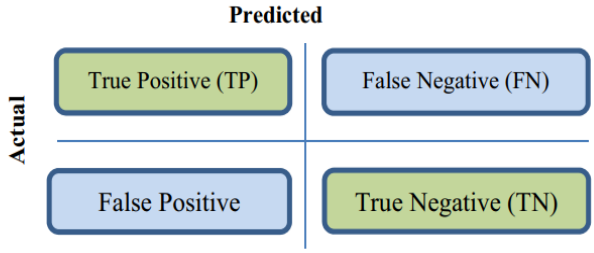
Output: Predicted class label or value

Procedure:

1. Initialize the final prediction F(x) to 0.
2. For each F_m in the ensemble:
   a. Compute the contribution of F_m(x) to the final prediction:
   F(x) += F_m(x)
3. Return F(x).

5. Performance evaluation metrics

Checking how well a model works is important when we're making it. This helps us find the best model that shows our data correctly and predicts how it will do in the future. In this research, we used something called a "confusion matrix" (you can see it in Figure) to see how well our techniques workedWe looked at different ways to see how good the models were, like accuracy, precision, recall, F-Measure, and AUC.



Values	Meaning
True positive (TP)	Output predicted by model and output in test data is true
True negative (TN)	Output predicted by model and output in test data is false
False positive (FP)	Output predicted by model is true while output in test data is false
False negative (FN)	Output predicted by model is false while output in test data is true

Table 1 list of values of evaluation metrics

Evaluation metrics	Values
Accuracy	$\frac{TN+TP}{TP+FP+TN+FN}$
Recall	$\frac{TP}{FN+TP}$
Precision	$\frac{TP}{FP+TP}$
F-measure	$\frac{2 * Precision*Recall}{Precision+Recall}$

Table 2 List of evaluation metrics

5.1. Accuracy

The percentage of correctly predicted samples compared to the entire sample is the accuracy of the model. It shows how frequently the developed model can be accurate forecast the result.

5.2. Precision

Precision The ratio of correctly identified true positives is what is measured.

5.3. Recall

Other names for recall include sensitivity or true-positive rate. It is useful to calculate the proportion of true positives to all true positives.

5.4. F- measure

F-measure Measuring the harmonic mean of evaluation metrics for recall and precision is useful. Its value is between 0 and 1.

5.5. Area under the curve (AUC)

In a classification problem, the "Area Under the Curve" (AUC) is a metric used to assess how well a model can distinguish between various classes. It is frequently employed when dealing with binary classification problems, which have two possible outcomes or classes.

In simpler terms, the AUC is like a summary of how well your model can tell things apart. If the AUC is closer to 1, it means your model is good at distinguishing between the classes. If it's closer to 0.5, it suggests that your model isn't doing much better than random guessing.

6. Discussion on Results

The primary goal of this study is to evaluate the effectiveness of the deep learning model that produced the encoded features. when combined with different methods for feature selection and feature extraction, as well as methods for addressing data imbalance issues, before being tested with machine learning algorithms like random forest and gradient boosting classifiers to determine their accuracy.

AUC: 0.6494482656582791

	precision	recall	f1-score	support
False	0.89	0.99	0.94	1783
True	0.88	0.31	0.45	326
accuracy			0.89	2109
macro avg	0.88	0.65	0.70	2109
weighted avg	0.89	0.89	0.86	2109

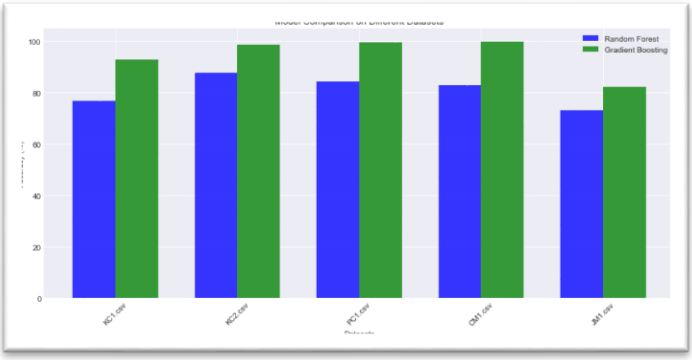
Results of Random Forest

AUC: 0.9610568800773494

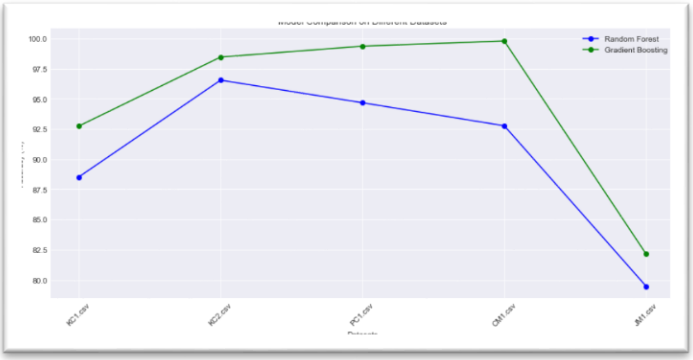
	precision	recall	f1-score	support
False	0.96	0.99	0.97	1783
True	0.93	0.76	0.84	326
accuracy			0.95	2109
macro avg	0.95	0.88	0.91	2109
weighted avg	0.95	0.95	0.95	2109

Results of Gradient boosting classifiers

Bar_chart Model comparison on different datasets



Summary of results



Line chart Model comparison on different datasets

Data set	Model	Accuracy	Precision	Recall	F-measure	AUC - average
KC1	RF	87	0.81	0.58	0.67	61
	GB	95	0.78	0.26	0.39	94
KC2	RF	90	0.91	0.59	0.72	80
	GB	98	0.98	0.76	0.85	98
PC1	RF	95	0.98	0.34	0.5	64
	GB	98	0.88	0.65	0.75	77
CM1	RF	93	93	99	96	64
	GB	98	98	99	99	97
JM1	RF	82	0.75	0.07	0.13	56
	GB	85	0.69	0.21	0.33	79

Difference in % for both classifiers (RF & GB)	
KC1	4.597701149
JM1	1.219512195
KC2	5.555555556
PC1	2.105263158
CM1	4.301075269

Difference in classifiers for each dataset.

7. Conclusion and future work: -

Researchers are always interested in finding better ways to predict defects in software systems accurately and quickly. This can help save time and money during software projects. The data about software defects is usually complex and unbalanced. Even the NASA dataset has this issue, which makes it hard to accurately predict defects. To make predictions better, a new approach is suggested. This approach combines different methods: one for picking out important information, another for selecting the right parts of that information, and two more for dealing with unbalanced data.

This improved dataset makes different computer programs that predict defects work much better. Among the methods tried in this study, using deep learning for extracting the features and machine learning classifiers are used for predict the defects from the output of the deep learning model gave the most accurate results. There's potential for more research to create even better methods that predict software defects more accurately and with minimum errors.

8. References

[1] *Wang, S., & Yao, X. (2013). Using class imbalance learning for software defect prediction. IEEE Transactions on Reliability, 62(2). https://doi.org/10.1109/TR.2013.2259203*

[2] *Kakkar, M., & Jain, S. (2016). Feature selection in software defect prediction: A comparative study. Proceedings of the 2016 6th International Conference - Cloud System and Big Data Engineering, Confluence 2016. https://doi.org/10.1109/CONFLUENCE.2016.7508200*

[3] *Thanapol, P., Lavangnananda, K., Bouvry, P., Pinel, F., & Leprevost, F. (2020). Reducing Overfitting and Improving Generalization in Training Convolutional Neural Network (CNN) under Limited Sample Sizes in Image Recognition. InCIT 2020 - 5th International https://doi.org/10.1109/InCIT50588.2020.9310787*

[4] *Song, Q., Guo, Y., & Shepperd, M. (2019). A Comprehensive Investigation of the Role of Imbalanced Learning for Software Defect Prediction. IEEE Transactions on Software Engineering, 45(12). https://doi.org/10.1109/TSE.2018.2836442*

[5] *Malhotra, R. (2015). A systematic review of machine learning techniques for software fault prediction. Applied Soft Computing Journal, 27. https://doi.org/10.1016/j.asoc.2014.11.023*

[6] *Muthukrishnan, R., & Rohini, R. (2017). LASSO: A feature selection technique in predictive modeling for machine learning. 2016 IEEE International Conference on Advances in Computer Applications, ICACA 2016. https://doi.org/10.1109/ICACA.2016.7887916*