Linköping University | Department of Computer and Information Science
Master's thesis, 30 ECTS | Computer Science
23 | LIU-IDA/LITH-EX-A--23/022--SE

Predicting Vulnerabilities in Third Party Open-Source Software using Data Mining and Machine Learning Techniques

Förutse sårbarheter i programvara med öppen källkod från tredje part med hjälp av datautvinning och maskininlärningstekniker

by

Frida Andersson
Albin Öberg

Supervisor : Navya Sivaraman
Examiner : Simin Nadjm-Tehrani

External supervisor : Maria Stegberg and Sofia Malmbratt

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Copyright

Abstract

In recent years, the use of third-party open source software (OSS) has increased significantly, making software security a critical concern. Predicting vulnerabilities in OSS can be a daunting task due to its complexity and the large volume of code involved. In this report, the use of data mining techniques and machine learning models to predict vulnerabilities in third party OSS is explored. The data used in this study was collected from GitHub repositories, where each repository consists of different package features fetched through the GitHub API. Repositories with reported vulnerabilities, CVEs, were then mapped to the corresponding vulnerability in the National Vulnerability Database (NVD).

The study used data mining techniques to analyze a large dataset with data from third-party OSS, and identified patterns and relationships between GitHub repository features and reported vulnerabilities. The dataset consisted of over 30 000 instances of OSS packages. Furthermore, the study employed various machine learning models to predict known vulnerabilities. The result showed that the best machine learning mode had an accuracy of 91,7% in predicting vulnerabilities in third-party OSS, and a precision of 0.90 for the positive class, representing repositories with reported CVEs.

Furthermore, the study revealed relationships between GitHub repository features and vulnerabilities. For instance, the analysis uncovered that the number of stars and forks has the highest impact on the predictions performed by the machine learning models. This finding enables more efficient detection of vulnerabilities in third-party OSS, thereby enhancing the overall security of digital systems. Overall, this study provides an approach for vulnerability prediction in third-party OSS and sheds light on important relationships between GitHub features and reported CVEs that were previously unknown.

Acknowledgments

We would like to extend our heartfelt gratitude to all those who have contributed to the successful completion of this master thesis. Firstly, we would like to express our sincere appreciation to our examiner Simin Nadjm-Tehrani and our supervisor Navya Sivaraman at Linköping University for their guidance and support throughout the entire research process. Their insights, and valuable feedback have been instrumental in shaping our research and helping us achieve our academic goals.

Additionally, we would like to extend our deepest appreciation to our supervisors, Sofia Malmbratt and Maria Stegberg at Saab, for their exceptional support, encouragement, and guidance throughout the research process. Their knowledge and guidance have been crucial in making this thesis a success.

Finally, we want to express our sincere appreciation to Oscar Olsson, who has generously devoted his time and expertise to guide us through the intricate process of identifying the best approach for this study. His invaluable insights have been essential in choosing the right approach for this study. We are truly grateful for his support and guidance.

Linköping, May 2023
Frida Andersson and Albin Öberg

Acronyms

ANFIS	Adaptive Neuro Fuzzy Inference System 2, 3, 8–10, 13, 16, 26–29, 37–40, 43, 45, 48
CPE	Common Platform Enumeration 3, 24, 29
CVE	Common Vulnerabilities and Exposures 2–4, 13, 14, 20–24, 29–31, 33–37, 40–47
CVSS	Common Vulnerability Scoring System viii, 3, 4, 15, 22, 23, 30, 41, 43, *Glossary:*
CWE	Common Weakness Enumeration 3, 4, 13, 15, 22, 24, 29
NVD	National Vulnerability Database 2–4, 13, 15, 16, 21, 24, 26, 31, *Glossary:* NVD
OSS	Open Source Software 1, 2, 26, 47, 48, *Glossary:* OSS
PCA	Principal Component Analysis 3, 5, 22, 24, 30
RFC	Random Forest Classifier 12, 27, 37–40, 43, 47
SVC	Support Vector Classifier 27, 37–40, 43, 45
SVM	Support Vector Machine 3, 7, 8, 12–15, 26, 29, 43
VDM	Vulnerability Discovery Model 16

Glossary

NVD The National Vulnerability Database is the U.S. government repository of vulnerability data. The database includes information of security checklists, security related software flwas, product names, and impact metrics. *Glossary:* NVD

OSS Computer software that is released under a license in which a copyright holder grants users the rights to use, study, change, and distribute the software and its source code to anyone and for any purpose. *Glossary:* OSS

Contents

List of Tables

1 | Introduction

This master thesis project aims to investigate how vulnerabilities in third-party open source software can be predicted using data mining and machine learning techniques. The project was executed at Saab and this chapter consists of an introduction to the problem, involving motivation, research questions, delimitation, and contributions.

1.1 Motivation

The reliance on technology in various industries, including defense and security, has resulted in an increased dependence on software. This has made software security a critical issue, and Saab, a high-security organization, is no exception [55]. Saab uses third-party Open Source Software (OSS) to improve the functionality, performance, and security of its products and services. However, the use of OSS also increases the potential for vulnerabilities, as acknowledged in Saab's Open Source Software Policy [45]. To mitigate this risk, Saab strives to identify potential vulnerabilities in a timely manner.

The Open Web Application Security Project (OWASP) listed the use of vulnerable and outdated software components as one of the top 10 security risks in 2021 [46]. One example of a consequence of software vulnerabilities is the Zotob worm, which caused widespread disruption in 2005 after a vulnerability in Microsoft Windows was exploited [37].

Moreover, vulnerabilities in OSS libraries can have severe consequences for the security of companies that use them. For example, the Equifax data breach in 2017 resulted from a missed OSS package update and exposed the private data of over 145 million U.S. citizens [38]. To reduce this risk, organizations might need a governance or audit system in place to identify their overall exposure to third-party OSS vulnerabilities [43]. In light of recent incidents like the Equifax data breach, there is increasing recognition of the importance of managing third-party OSS vulnerabilities. As a consequence, this has led to calls for the adoption of Software Bill Of Materials (SBOMs) as a way to improve transparency and accountability in the software supply chain [57]. Thus, identifying and mitigating vulnerabilities in third-party OSS can be seen as a critical component of effective software governance and risk management.

To address the issue of software vulnerabilities, various approaches have been proposed, including machine learning and data mining techniques. Empirical studies have shown that machine learning techniques can outperform common statistical methods [26] and have demonstrated promising results in similar predictive tasks. Additionally, since much of the source code for OSS is publicly available, attackers may be more likely to exploit discovered vulnerabilities, as they can obtain information about the software's weaknesses and how to best exploit them from the web. By predicting vulnerabilities in third-party OSS, organizations like Saab can take a proactive approach to software security and reduce the risk of cyber attacks.

1.2 Aim

The aim of this thesis is to investigate how data mining and machine learning techniques can be utilized to predict vulnerabilities in third-party open-source software, as opposed to identifying or fixing them. Predicting vulnerabilities refers to the ability to anticipate potential security issues before they occur, while identifying vulnerabilities involves finding and reporting on already-existing security flaws. While both tasks are important for software security, this thesis focuses on the predictive aspect and aims to evaluate the performance of different machine learning models in this context.

Additionally, the study aims to analyze the impact of various software characteristics on the likelihood of security vulnerabilities and evaluate the performance of the chosen machine learning models under various vulnerability severities. By achieving this aim, this study will contribute to the development of more effective approaches to identify and mitigate vulnerabilities in third-party OSS, ultimately benefiting software developers, users, and other stakeholders.

1.3 Research Questions

This thesis seeks to answer the following research questions:

1. How can data mining and machine learning techniques be used to predict vulnerabilities in third-party open source software?
2. Which of the chosen machine learning models yields the highest accuracy for predicting vulnerabilities in third party open source software?
3. What is the relative impact of various software characteristics on the likelihood of a security vulnerability?
4. How do the chosen machine learning models perform on different types of vulnerabilities (e.g. high-severity, low-severity)?

1.4 Delimitation

There are several limitations to this study that should be taken into consideration. This study only focuses on third-party OSS repositories available on GitHub and associated with Common Vulnerabiliteis and Exposures (CVE) s in the National Vulnerability Database (NVD) . Therefore, the findings may not be relevant to other types of software or repositories that do not have CVE associations. Furthermore, the study only employs a selected number of data mining techniques and machine learning models. Other techniques and models may yield different results, and their exclusion from this study may limit the overall accuracy of the findings. Thirdly, the study is restricted to a specific set of features provided by the GitHub API, which may overlook other important features or metadata associated with the repositories, such as commit history. Moreover, external factors such as user behavior or changes in the software development environment that could impact the accuracy of vulnerability predictions are not considered.

1.5 Contributions

The proposed framework for predicting vulnerabilities in third-party OSS consists of several key components, including the data mining techniques clustering, isolation forest, and association rule mining, and the machine learning models random forest, support vector machine, and Adaptive Neuro Fuzzy Inference System (ANFIS) . The

- Identification of significant features for predicting vulnerabilities in third-party OSS, including repository metadata obtained from the GitHub API.
- Utilization of clustering, isolation forest, and association rule mining to uncover patterns and relationships in the repository metadata related to reported CVEs.
- Evaluating the performance of random forest, support vector machine, and ANFIS on predicting vulnerabilities in third-party OSS.

demonstrate the potential of the proposed framework to enhance the security of third-party OSS by detecting vulnerabilities before they are exploited by attackers. The framework offers a method that can be applied to identify and predict vulnerabilities in third-party OSS repositories on GitHub, with the ultimate goal of improving software security practices.

2 | Theory

This chapter describes software vulnerabilities and CVEs, which are unique identifiers assigned to vulnerabilities. It also explains data mining techniques like Principal Component Analysis (PCA) , clustering, association rule mining, and isolation forest, which are techniques used to analyze large amounts of data. Additionally, the chapter describes three machine learning models: Support Vector Machine (SVM) , ANFIS and Random Forest, and how they can be evaluated. Additionally, the chapter reviews related work on vulnerability detection, data mining on GitHub repository metadata, and vulnerability prediction using machine learning techniques, which provides a foundation for the research presented in this thesis.

2.1 Software Vulnerabilities

This section provides the definition of software vulnerabilities and information about where these can be found and how they have been established. Dowd et al. [15] defines vulnerabilities in software systems as weaknesses that can be exploited by attackers. These vulnerabilities are specific flaws or oversights in a piece of software that allow attackers to perform harmful actions, such as exposing or altering sensitive information, disrupting or destroying a system, or taking control of a computer system or program. The author also note that vulnerabilities can be thought of as a subset of software bugs. A security vulnerability is a type of software bug that has the added potential for malicious use, allowing attackers to launch attacks against the software and supporting systems [15]. Dowd et al. state that most security vulnerabilities are software bugs, but not all software bugs are security vulnerabilities. A bug must have a security-related impact or property to be considered a security issue; in other words, it must allow attackers to do something they would not normally be able to do.

2.1.1 Common Vulnerability and Exposures (CVE)

The CVE is a standard that identifies and names security vulnerabilities in a consistent way [40]. It assigns a unique identifier to each confirmed and public vulnerability and provides information like its description, affected software/system, Common Weakness Enumeration (CWE) ,Common Platform Enumeration (CPE) , and CVSS. Figure 2.1 shows an example of a security vulnerability with an assigned CVE. In addition, the CVE has references to other sources of information like NVD, security advisories, and patches. The standardized system is widely used by organizations and software vendors across industries, making it easier to track and manage security risks and vulnerabilities across different systems and vendors [40].

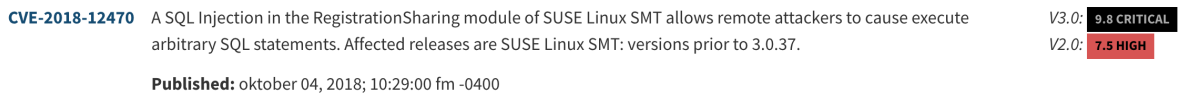

Figure 2.1: Example of a CVE with ID CVE-2018-12470

2.1.2 National Vulnerability Database (NVD)

The NVD and the National Checklist Program (NCP) have been providing information on security risks to users worldwide since 2005 and is maintained by the National Institute of Standards and Technology

(NIST) [5]. NVD was created to serve as a US government database for software vulnerabilities, using open standards to offer reliable and inter-operable information of reported vulnerabilities [5].

The data in the database consists of reported and identified CVEs, the affected software or system, the type of vulnerability, and the severity of the vulnerability. Additionally, the data includes information about the vulnerability's impact such as potential data loss or issues with confidentiality, integrity, and availability of the affected system [5].

2.1.3 Common Vulnerability Scoring System (CVSS)

The CVSS score helps organizations understand the potential danger of a vulnerability in a systematic way [54]. It assigns a numerical value to each vulnerability based on various factors such as the ease of exploiting it, the level of access required, and the potential harm it could cause in terms of confidentiality, integrity, and availability of information [54].

The CVSS score is calculated using CVSS vectors that detail how the score is determined [54]. As shown in Figure 2.2, these vectors include several metrics like Attack Vector, Attack Complexity, Privileges Required, User Interaction, Scope, Confidentiality Impact, Integrity Impact, and Availability Impact for CVSS version 3.0. These metrics takes different elements of the vulnerability into account and are combined to a final CVSS score, which serves as a representation of the overall severity of the vulnerability. For example, a vulnerability that can be easily exploited and has the potential to cause significant harm will receive a higher score, while a vulnerability that requires specific conditions to be exploited and has a limited impact will receive a lower score. However, as Anwar et al. [1] highlights, the CVSS score has gone through different versions over time, with each version having its own characteristics and methods for calculating the score. While both CVSS v2 and CVSS v3 scales between 0 to 10, v3 introduces a critical level of severity as presented in Table 2.1.

Figure 2.2: CVSS Score for CVE-2018-12470

Label	v2	v3
None	-	0.0
Low	0.0–3.9	0.1–3.9
Medium	4.0–6.9	4.0–6.9
High	7.0–10.0	7.0–8.9
Critical	-	9.0–10.0

Table 2.1: Score thresholds for CVSS v2 and CVSS v3

2.1.4 Common Weakness Enumeration (CWE)

The CWE is a standardized list of common software security weaknesses used to identify, describe and categorize them in a consistent manner [35]. Each CVE consist of one or more CWEs, which provides a common language for describing software security weaknesses, making it easier for organizations and software vendors to manage them. The CWE list includes different types of security classifications, each with a specific CWE-id, description, examples, and potential impacts. The list also has a hierarchy of weakness types, categorized to help the user easily understand and identify patterns and trends in software vulnerabilities. A few examples of CWEs are [13]:

- CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-Site Scripting')
- CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
- CWE-200: Information Exposure
- CWE-307: Improper Restriction of Excessive Authentication Attempts
- CWE-352: Cross-Site Request Forgery

2.2 Data Mining

Data mining is a process of discovering patterns, relationships, and insights from large amounts of data [17]. Additionally, it is a crucial step in the data preprocessing phase and is important for identifying useful information from collected data. This section highlights different data mining techniques and describes their approach.

2.2.1 One-Hot Encoding

One-hot encoding for categorical features is a technique commonly used in data preprocessing for machine learning [14]. It involves representing categorical data with binary values, with each category becoming a separate binary feature. This technique is typically applied to categorical features that do not have any intrinsic ordering or ranking, such as colors or types of products.

2.2.2 Principal Component Analysis (PCA)

PCA is a statistical technique that aims to reduce the dimensionality of high-dimensional data while retaining most of the information and variability [34]. PCA transforms the original data into a new set of uncorrelated variables called principal components (PCs), which are linear combinations of the original variables. The first PC captures the largest amount of variance in the data, and each subsequent PC captures as much of the remaining variance as possible. By selecting a subset of PCs that explain most of the variance, PCA can simplify complex data and reveal underlying patterns and structures. PCA has many applications in various fields such as image processing, signal processing, data compression, machine learning, and bioinformatics [28].

PCA is useful in machine learning when the data has high dimensionality, meaning it has many features or variables that may be correlated, redundant, noisy or irrelevant [29]. By applying PCA, machine learning practitioners can reduce the number of features and select the most informative ones that capture most of the variance and structure of the data. This can help improve the performance and interpretability of machine learning models by reducing noise, redundancy, and multicollinearity in the data. In addition, PCA is useful in visualizing high-dimensional data by projecting it onto a lower-dimensional space [29].

2.2.3 Clustering

Clustering is a technique used to group together similar data points based on their features [53]. According to Saxena et al. [53], K-means is one of the most popular clustering algorithms. The algorithm involves choosing k clusters, and randomly selecting k data points to act as centroids for each cluster. The algorithm then uses a distance measure to assign each data point to the cluster with the closest centroid. The process is repeated, with the centroids being recalculated each time, until the centroids no longer move. At this point, the data points are separated into their respective clusters. However, there are limitations to the K-means algorithm, including difficulty in identifying the initial partitions and the number of clusters, as well as sensitivity to outliers and noise [53].

One technique to determine the optimal number of clusters is the Elbow method [11]. The intuition behind this method is that as the number of clusters increases, the Sum of Squared Distances (SSD) decreases because the data points are closer to their centroids. SSD is a measure of the total distance between each data point and its assigned cluster centroid, squared and summed. However, at some point, adding more cluster will not provide much gain in SSD reduction, resulting in a less steep decline

in the curve, forming an elbow shape. When this point is reached the optimal number of clusters is defined [11].

When applying clustering algorithms to data, it is essential to evaluate the quality of the resulting clusters [56]. One common tool for this purpose is the silhouette score [56]. This metric measures the similarity of each object to its own cluster compared to other clusters, and assigns a value between -1 and 1. A higher silhouette score indicates a more distinct and well-defined cluster. Despite its usefulness, the silhouette score has some limitations [56]. Shahapure and Nicholas [56] explains that it can be sensitive to the shape and density of the clusters, and it assumes that the data is normalized.

2.2.4 Association Rule Mining

Association rule mining is a method that helps to find connections between different features in a big dataset [32]. This is done by finding patterns that occur frequently and using those patterns to create rules that describe how those pieces of data are related. An association rule is typically written in the form of "if X then Y" or "X implies Y". The rule indicates that there is a strong correlation or association between two variables X and Y. In mathematical notation, an association rule can be represented as:

$$\mathbf{X}, \mathbf{Y} \rightarrow \mathbf{Z} \tag{2.1}$$

This equation states that if items X and Y appear together, then there is a probability that item Z will appear for the data point with a given *confidence* level. The arrow symbol ($\rightarrow$) denotes the implication or the association between the variables. The items on the left side of the arrow are called *antecedents*, and the item on the right side is called the *consequent* [32].

One popular way to implement association rule mining is by using the Apriori algorithm, which looks for frequently occurring items and generates rules based on those items [32]. To use the algorithm, the dataset needs to be prepared by removing irrelevant or noisy data and turning it into a transactional format. The algorithm then looks for frequently occurring items and uses those to generate candidate itemsets. This process is repeated to generate larger itemsets until the algorithm finds the frequent itemsets that meet the minimum support threshold. Once the frequent itemsets are found, the algorithm generates rules that fulfills the minimum confidence threshold. This means that the rules are based on data that occurs frequently enough to be considered reliable. The results from the Apriori algorithm consists of [3]:

1. Support: refers to the frequency of occurrence of a particular itemset in the dataset. It indicates how frequently the items in the itemset appear together in the dataset.

2. Confidence: measures the strength of the association between two items in an itemset. It measures the proportion of transactions that contain both items in the itemset out of the total number of transactions that contain the first item.

3. Lift: It is used to determine how much more often two items appear together than would be expected if they were independent of each other. A lift value greater than 1 indicates that the two items are positively correlated, and that the presence of one item increases the likelihood of the other item being present. A lift value equal to 1 indicates that the two items are independent, and a lift value less than 1 indicates a negative correlation, meaning the presence of one item decreases the likelihood of the other item being present.

4. Conviction: measures how strongly the presence of the antecedent implies the presence of the consequent in a dataset, taking into account the overall frequency of both items. Conviction values range from 0 to infinity. A value of 1 indicates independence between the two items, a value greater than 1 indicates positive dependence, and a value less than 1 indicates negative dependence.

5. Leverage: measures the difference between the observed frequency of co-occurrence of two items in a dataset and the frequency that would be expected if the two items were independent. A leverage value of 0 indicates independence between the two items, while a positive leverage value indicates that the two items occur together more often than expected. Intuitively, a negative leverage value indicates that they occur together less often than expected.

Once the frequent itemsets and association rules have been generated, their quality and usefulness can be evaluated by measuring their support and confidence [3]. Additionally, the result can be visualized using various graphs and charts to gain insights into the relationships between the variables [32].

2.2.5 Isolation Forest

Outlier detection is a technique used in data mining to find data points or patterns that don't fit with the expected behavior of a system or process [50]. There are many different ways to implement outlier detection. However, according to Hariri et al. [20], isolation forest is a unique outlier detection algorithm that utilizes its model-free nature, computational efficiency, scalability with parallel computing paradigms, and effectiveness in detecting outliers. Unlike other algorithms that build profiles of data to detect nonconforming samples, Isolation Forest leverages the fact that outliers are few and different. It randomly selects features and splits values to construct a tree structure that isolates outliers with shorter branches and normal data with longer branches.

2.3 Machine Learning

Machine learning has become increasingly popular in the field of data analysis and predictive modeling [26]. There are a wide range of algorithms available for use, each with its own strengths and weaknesses. In this background section, three commonly used models are presented: Support Vector Machines (SVMs), Adaptive Neuro Fuzzy Inference System (ANFIS), and Random Forest.

2.3.1 Support Vector Machines (SVMs)

SVMs are a type of supervised machine learning models used for classification and regression analysis. The algorithm identifies a hyperplane that separates the data into classes with the greatest margin, which is defined as the distance between the hyperplane and the closest data points, also known as support vectors [47]. The optimization problem in the linear case is represented as:

$$\min_{w,b} \frac{1}{2}||w||^2 \quad \text{subject to} \quad y_i(w \cdot x_i + b) \geqslant 1, \quad i = 1, \ldots, n \tag{2.2}$$

In this equation, x_i and y_i are the feature vectors and corresponding label for the i^{th} data point, w is the weight vector, b is the bias term and n is the number of data points. Moreover, if the independent variables have a high correlation it can affect the performance of SVM:s by reducing the margin and increase the complexity of the hyperplane. To deal with this problem, the algorithm implements a solution called the kernel trick [22].

SVMs have advantages over other machine learning algorithms, such as k-nearest neighbors (kNN), in that they cope well with data that is not linearly separable and are less prone to overfitting [42]. In the case of binary classification, the algorithm seeks the maximum-margin hyperplane that separates the two classes, as seen in Figure 2.3. However, in practice, the algorithm is modified to include a "soft margin" that allows some data points to cross the margin without affecting the final result [44].

The Kernel Trick

In non-linear cases, the data is transformed into a higher dimensional space by using kernel functions, such as polynomial or radial basis functions. By transforming the data, the kernel trick allows SVMs to handle nonlinear and correlated data without increasing the computational cost [22]. The optimization problem in this case becomes:

$$\min_{\alpha} \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j K(x_i, x_j) \quad \text{subject to} \quad \sum_{i=1}^{n} \alpha_i y_i = 0, \quad 0 \leqslant \alpha_i \leqslant C \tag{2.3}$$

In the equation, $K(x_i, x_j)$ is the kernel function used to transform the input data into a higher-dimensional space, α_i and α_j are Lagrange multipliers used to maximize the margin between support vectors, and C is a regularization parameter used to balance the trade-off between maximizing the margin and minimizing the classification error.

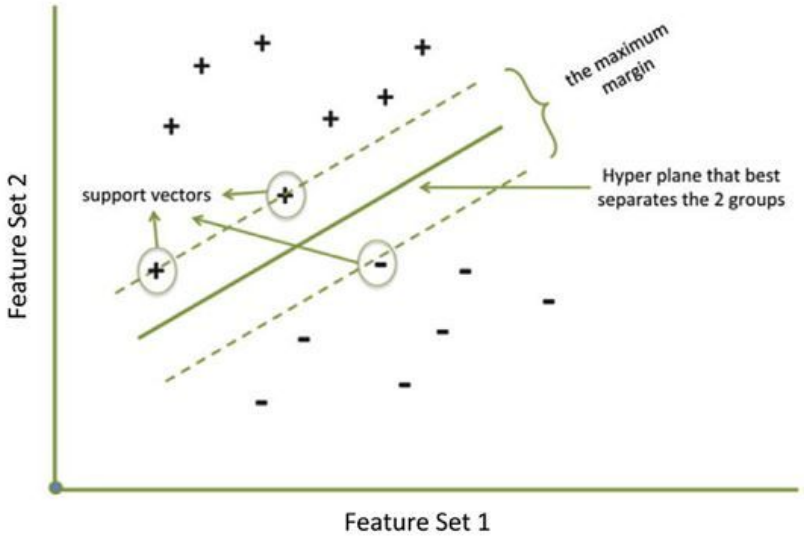

Figure 2.3: Illustration of the hyperplane that maximally separates two classes [49]

The kernel function adds an additional dimension to the data, which allows for better separation in the higher dimensional space [44]. Figure 2.4 shows an illustration of the kernel trick. However, projecting into very high-dimensional spaces can result in the curse of dimensionality, where the number of possible solutions increases exponentially as the number of variables increases, making it harder for the algorithm to select the correct solution. Additionally, the risk of overfitting is significant since the boundary between classes can be specific to the examples in the training data set. The linear SVM does not use the kernel trick, but instead directly operates in the input space, while the non-linear version applies the trick. Finally, the SVM can be generalized to a multi-class version by performing several one-vs-rest classifications and selecting the class with the best separation.

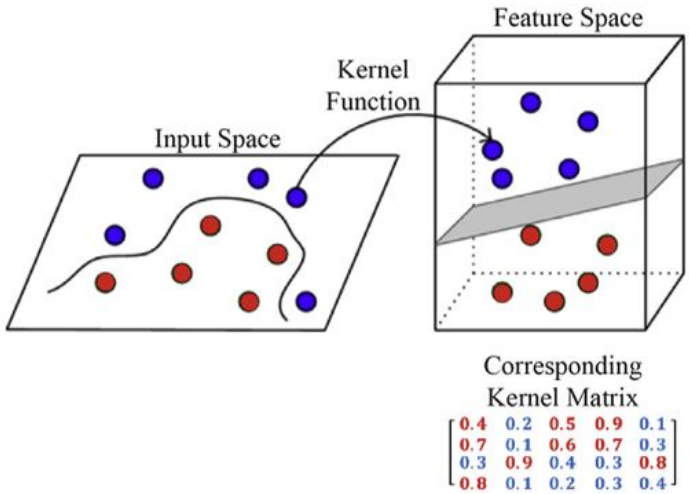

Figure 2.4: Illustration of the kernel trick [49]

2.3.2 Adaptive Neuro Fuzzy Inference System (ANFIS)

The comparative study by Dawar et al. [26] shows that the ANFIS algorithm is proficient in predicting vulnerabilities in third-party software. The authors describe it as a combination of two types of systems: artificial neural networks (ANN) and fuzzy logic systems. Siniša et al. [58] further explains that the ANFIS model performs well for tasks like, guessing how a function behaves, predicting what will happen in the future, and controlling systems. The model consists of a collection of components

that each represent a rule in the fuzzy system, where the final answer consist of a mix of all of these components with weights that are determined by the ANN. The weights are changed during training to make sure the model produces accurate results.

Moreover, the correlation between independent variables can have a significant effect on the model's performance. If variables are highly correlated it causes multicollinearity, which can lead to unstable parameter estimates and reduce the model's predictive power. To deal with this problem, ANFIS uses a technique called correlation analysis, a method which measures the strength and direction of linear relationships between two variables [48]. This technique can help in identifying and eliminating redundant or irrelevant input variables with low or high correlation with the output variable [48].

The ANFIS model consists of two main parts: the premise and the consequence [31]. ANFIS also uses real data from past situations to make connections between the premise and consequence parts using rules. Figure 2.5 shows the structure of the ANFIS model Karaboga and Kaya [31] used in their study which consists of five layers, including four membership functions and four rules.

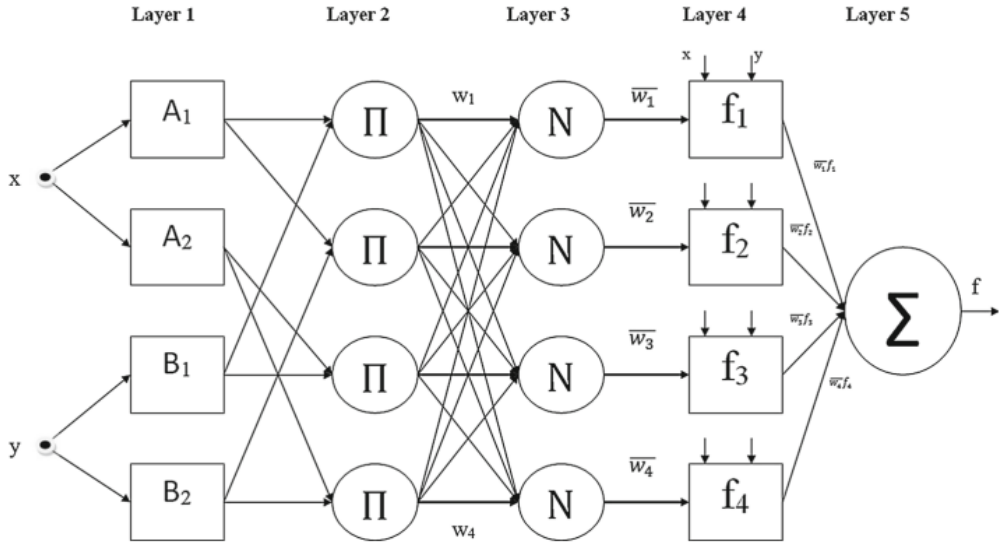

Figure 2.5: Example of the ANFIS structure with two inputs and one output [31]

- **Layer 1:** the "fuzzification layer" which takes the input values (x and y) and turns them into fuzzy groups using the *membership functions*. The shape of these *membership functions* is controlled by the *premise parameters* (a, b, or c). These parameters are used to calculate how much each input belongs to each membership function, see 2.4 and 2.5.

$$\mu_{A_i}(x) = gbellmf(x; a, b, c) = \frac{1}{1 + \left|\frac{x-c}{a}\right|^{2b}} \tag{2.4}$$

$$O_i^1 = \mu_{A_i}(x) \tag{2.5}$$

- **Layer 2:** is the *rule layer*. The nodes of this layer are fixed nodes and labeled with π to indicate that they are multipliers. The layer output W_i represent the fuzzy strength of each rule and can be expressed by equation 2.6.

$$O_i^2 = W_i = \mu_{A_i}(X) * \mu_{B_i}(Y) \quad i = 1, 2 \tag{2.6}$$

- **Layer 3:** is called the *normalization layer*. The purpose of this layer is to calculate normalized fuzzy strengths which belongs to each rule. The normalized value represents the ratio of the fuzzy strength of the i:th rule with regards to the total of all strengths as given in the equation 2.7.

$$O_i^3 = \overline{w_i} = \frac{w_i}{w_1 + w_2 + w_3 + w_4} \quad i = 1, 2, 3, 4 \tag{2.7}$$

- **Layer 4:** is known as the *defuzzification layer* where the weighted values of each rule are calculated in each node. This value is determined by using first order polynomial as given in equation 2.8.

$$O_i^4 = \overline{w_i} f_i = \overline{w_i}(p_i x + q_i y + r_i) \tag{2.8}$$

- **Layer 5:** represents the *summation layer* where the actual output is generated by summing the outputs for each rule in the *defuzzification layer* according equation 2.9.

$$O_i^5 = \sum \overline{w_i} f_i = \frac{\sum_{i=1}^{n} w_i f_i}{\sum_{i=1}^{n} w_i} \tag{2.9}$$

According to a study by Lei et al. [33], the ANFIS model is trained using a combination of two methods: gradient descent and least-square estimates. During the training process, the model performs a forward-pass to determine the parameters for the consequent part using a least-square estimate and then a backward-pass to update the premise parameters using gradient descent. The combination of these two methods allows the ANFIS model to effectively find the optimal parameters that fit the input-output data.

Gradient descent is an optimization algorithm that updates the model parameters in the direction of the negative gradient of the cost function $J(w)$ with a learning rate η, where t is the iteration step:

$$w_{t+1} = w_t - \eta \nabla J(w_t) \tag{2.10}$$

Least-square estimates, on the other hand, is a method for finding the parameters that minimize the sum of squares of the residuals between the observed data and the model predictions:

$$w^* = \arg\min_w J(w) = \arg\min_w \sum_{i=1}^{n} (y_i - f(x_i; w))^2 \tag{2.11}$$

where y_i is the observed data, $f(x_i; w)$ is the model prediction, and n is the number of data points.

2.3.3 Random Forest

Random forest consists of multiple decision trees [6] [39]. Decision trees are models used for classification and regression tasks [39]. The tree is constructed by recursively splitting the data into subsets based on the values of its features until each subset only contains a single class or a single predicted value. The splitting process is determined by selecting the feature that results in the highest reduction of impurity, where impurity is a measure of the homogeneity of the class distribution in the subset [39].

Random vectors are generated independently to control the growth of each tree in a random forest using methods like bagging, random split selection, and randomizing outputs [6] In random forest classification, each decision tree gives a prediction, and the trees in the forest then vote on the most likely accurate result, with the most popular class being the final outcome [6]. An illustration of this can be seen in Figure 2.6.

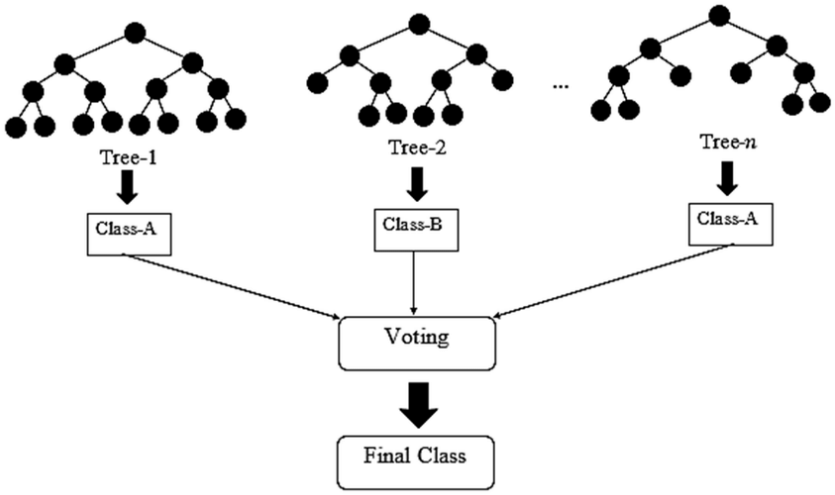

Figure 2.6: Random forest classifier illustration [2]

According to Gregorutti et al. [18], the relationship between independent variables can impact its performance by causing redundancy when implementing the model. This means that multiple variables may provide similar information about the output variable, resulting in overfitting and reduced interpretability. To address this issue, the random forest model uses a technique called variable importance [18]. This method assesses the contribution of each independent variable to the prediction accuracy and diversity of the model. Thus, it helps to identify and remove input variables that are redundant or irrelevant and have low or negative importance scores. This approach can improve the overall performance of the model and make it easier to interpret the results [18].

2.4 Model Evaluation

Model evaluation is a crucial aspect to address when developing machine learning models. This section describes cross-validation techniques and performance metrics that are commonly used to evaluate the effectiveness of machine learning models. Additionally, the section explains the concept of feature importance analysis and how it can be applied to the previously explained machine learning models.

2.4.1 Cross-Validation

Cross-validation is a widely used technique for evaluating the performance of machine learning models and avoiding overfitting [21]. The method helps to estimate the true prediction error of models and fine-tune their parameters [4], thus ensuring the generalization ability of predictive models on independent, unseen data.

K-fold cross-validation is a popular approach that divides the data into subgroups (folds) and trains the model on k-1 folds while testing it on the remaining fold [4]. This process is repeated k times, each time using a different fold as the test set, and the average test performance is used as the final evaluation metric. Stratified K-fold cross-validation is a variation of K-fold that ensures the class frequency of the target variable is preserved in each fold [4]. This approach is suitable for smaller datasets where truly random sampling may lead to skewing of class proportions.

2.4.2 Performance Metrics

Performance metrics play a crucial role in determining the effectiveness of a machine learning model. These metrics provide a way to measure and evaluate the accuracy of different models. The applicability of the models is examined using their predictive capability. Most frequently used performance metrics are accuracy, recall, precision, and F1 score [16] [8], which gives a general idea of how accurate the predictions are. For a binary classification, there are four cases as shown in Table 2.2.

		Predicted	
		Positive	**Negative**
Actual	**Positive**	True Positive (TP)	False Negative (FN)
	Negative	False Positive (FP)	True Negative (TN)

Table 2.2: Confusion matrix for binary classification

Accuracy is the ratio of the number of correct predictions to the total number of predictions made by the model and provides a measure of how often the model makes correct predictions.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{2.12}$$

Precision is the ratio of the number of true positive predictions to the total number of positive predictions made by the model and measures the ability of the model to avoid false positive predictions.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2.13}$$

Recall is the ratio of the number of true positive predictions to the total number of actual positive cases and measures the ability of the model to identify all actual positive cases.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2.14}$$

F1 Score is the harmonic mean of precision and recall, thus balancing the trade-off between precision and recall. A high F1 score indicates a good balance between precision and recall, and a model with high F1 score is preferred when the costs of false positive and false negative predictions are similar.

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.15}$$

2.4.3 Feature Importance Analysis

Feature importance analysis is a technique used in machine learning to determine which features or input variables are most important for a predictive model. It is a key challenge in many practical aspects of machine learning, as it helps measure and evaluate how different features are affecting the model [52]. This analysis involves assigning a score to each feature based on its contribution to the model's performance [62]. The higher the score, the more important the feature is considered. Feature importance analysis can help in identifying the key factors that affect the model's outcome, understanding the relationships between features, and improving the model's performance by removing irrelevant or redundant features [62] [52].

Different methods are used for feature importance analysis, depending on the type of model being used [59]. In Random Forest Classifier (RFC) , feature importance analysis is performed by measuring the decrease in the model's accuracy when a particular feature is removed. This is achieved by permuting the values of a single feature and observing the effect on the model's accuracy. Features with the highest decrease in accuracy are considered the most important.

In the case of SVM with different kernels, the importance of each feature is measured by its contribution to the decision boundary of the SVM. This is determined by the magnitude of the feature's weight in the SVM model. For SVMs using linear kernels, the feature importances are calculated using the coefficients of the linear SVM. Otherwise, for RBF kernels, permutation importance is used to compute the feature importances. Permutation importance works by randomly shuffling the values of each feature in the testing set and measuring the decrease in model performance [52]. The larger the

decrease, the more important the feature is for the model. This process is repeated multiple times to get a more accurate estimate of the feature importances.

For the ANFIS algorithm feature importance is calculated in the same way as for SVM, where each feature is assigned a different weight which resembles the contribution to the decision boundary. According to Honda et al. [23], the consequent part of the fuzzy inference system rules is a linear combination of the input variables, whereas the final output is the weighted average of each rule's output. Thus, by identifying which rules has the largest weights, the most important features can be extracted.

2.5 Related Work

The purpose of this section is to provide an overview and summary of previous research on the topic. It includes an analysis of the use of machine learning models for vulnerability prediction, alternative techniques that have been employed, the data utilized in other studies, and the approaches used to collect that data.

2.5.1 Vulnerability Detection

This section will explain a number of studies which have been conducted to understand and improve the methods used for vulnerability detection. Overall, the studies in this area highlight the importance of considering the quality of data sources and real-world context for accurate vulnerability prediction.

Massacci et al. [36] raised the question in their study that if the data sources used do not completely capture the phenomenon that is interesting to predict, then the predictions may be optimal with respect to the available data but unsatisfactory in practice. The authors conducted a comparison of various vulnerability databases, based on the features of vulnerabilities they contained. The databases were classified into three categories: multi-vendor databases (such as CVE and NVD), vendor-specific databases (Microsoft Security Bulletin, Mozilla Foundation Security Advisories), and other databases (such as OpenBSD Vulnerability Database). They found that most papers on vulnerability prediction use vendor databases as they need references from security bugs to the corresponding code base, which is not supplied in multi-vendor databases. Furthermore, the authors found that the majority of the data sources used in prediction are bug-tracking databases or source version control, but the discovery date and many other important features are often missing, which impacts the prediction power of the models. In conclusion, by focusing on the fundamental question which is the quality of the vulnerability database and the data used to train the models, the study show that by using different data sources, the results may differ.

Svensson [60] investigated the use of different vulnerability sources for vulnerability assessment and compared the NVD with the Node Package Manager (NPM) security advisory. The study found that a significant proportion of the vulnerabilities analyzed were only reported on one source, with 64.4% of the scraped advisories and 49.0% of the fetched CVEs falling into this category. Additionally, the study found that the NPM security advisory was the first source to report the majority of the vulnerabilities, with 81.3% of the total vulnerabilities being initially published on this platform. Furthermore, some information only exists on one source, such as CWE categories in NVD and information about the status of the vulnerability on NPM. Therefore, Svensson argued that using NVD in combination with NPM would provide the most comprehensive information about vulnerabilities, indicating a need to combine NVD with other vulnerability sources that are often used for specific package managers.

Jimenez et al. [27] investigated the impact of real-world labeling on the performance of machine learning models for vulnerability prediction. The study found that models trained on real-world labels, which are labels derived from actual vulnerabilities found in software systems, including SQL injection vulnerability or Cross-Site Scripting (XSS), performed better than models trained on simulated labels, i.e., labels that are artificially generated for the purpose of training. The authors also found that a combination of real-world and simulated labels could further improve the performance of the models. Finally, the authors concluded that real-world labeling is an important consideration when using machine learning models for vulnerability prediction and that the community needs to improve its experimental and empirical methodology to ensure robust and actionable scientific findings. Additionally, the study highlights the importance of considering the real-world context of the data when training and evaluating machine learning models for vulnerability prediction.

Perl et al. [47], presented a solution to reduce the high false-positive rate of source code analyzing tools for identifying software vulnerabilities in open source repositories. To implement this method, code-metric analysis was combined with metadata from code repositories. Then, GitHub commits were mapped to CVEs to create a database of vulnerable commits. Finally, an SVM classifier was trained. The method resulted in a 99 % reduction in false alarms compared to other source code analysis software. The authors focused on four metric categories when extracting information from GitHub, where the first category, *Features Scoped by Project*, used metrics such as programming language, star count, fork count, and number of commits to highlight interesting metrics when searching for vulnerabilities in the software. The other three categories, *Features Scoped by Author*, *Features Scoped by Commit*, and *Features Scoped by File*, were considered less relevant for this thesis as they contained text-based features and features that were more closely tied to specific commits.

2.5.2 GitHub Data Mining for Vulnerabilities Analysis

In recent years, GitHub has become a central hub for open-source software development, offering a variety of collaboration tools for developers to work together on their projects. GitHub provides a vast amount of data on software repositories, such as the number of stars, forks, commits, and issues. Analyzing this data can provide insights into the development and security of these repositories. According to Gahffarian et.al [17], data mining techniques are effective tools that can be used in software vulnerability analysis and discovery. This section highlights several studies that have used data mining techniques to analyze GitHub repository data related to vulnerabilities.

Cosentino et al. [9] conducted a meta-analysis of more than 90 research papers that investigate the use of data mining methods on GitHub data. The authors identify three main dimensions of these papers: the empirical methods employed, the datasets used, and the limitations reported. The meta-analysis reveals several concerns, including issues with dataset collection process and size. This issue have been addressed by several other authors [7][51] which highlights the GitHub REST API restrictions of number of requests per minute as a limitation to build a dataset. However, Cosentino et al. [9] discuss the trade-off between mining data from GitHub using the API or existing datasets from third-party services. They note that these solutions are currently contradictory with respect to their freshness and curation, and require researchers to evaluate this trade-off carefully. The GitHub API allows developers to access a limited amount of fresh and uncensored data, such as the latest snapshot of a project, but there are limitations on API requests. In contrast, existing datasets and services provide curated data, such as GHTorrent and BOA, but they are often out-of-date compared to the original data on GitHub.

Horawalavithana et al. [24] conducted a study to compare user-generated content related to security vulnerabilities on three digital platforms: Reddit, Twitter, and GitHub. The aim was to investigate the correlation between the number of security vulnerabilities mentioned on the platforms, and whether software development activities on GitHub could be predicted from discussions on Reddit and Twitter. The authors also examined two repositories with the highest number of distinct CVEs mentioned during the chosen timeframe to study the relationship between GitHub activity and the number of CVEs. The study found that activities such as forking, watching, and pushing (number of commits) correlated best with the number of CVEs. The pattern of GitHub pushes showed a similar trend with the number of CVE mentions, as software patches are contributed to the repository code through GitHub pushes. However, popularity measures such as fork and watch activities also showed a moderate correlation with the number of CVE mentions.

Several other studies have explored the relationship between the metadata of GitHub repositories and software vulnerabilities. Naveed [41] examined ten popular C++ source repositories on GitHub and found no significant correlation between the frequency of vulnerabilities and repository metadata such as stars, lines of code, and other attributes. However, Naveed acknowledged the limitations to the study and suggests that more code repositories, including those for other languages, should be analyzed in the future.

In another study, Ibrahim et al. [25] analyzed the top 100 open source PHP projects on GitHub and found that project activities like branching, pulling, and committing had a moderate positive correlation with the number of vulnerabilities. On the other hand, popularity factors like stars and watchers, as well as other factors such as the number of releases, issues, and contributions, had little to no impact on the presence of vulnerabilities [25]. Overall, project activity-related factors were found to be the most influential in the prevalence of vulnerabilities in open source projects.

Chatziasimidis and Stamelos [7] discusses the collection and analysis of data from GitHub repositories and users using data mining techniques. The authors describe the data collection process, which involves using the GitHub API to retrieve data about repositories, such as metadata, commits, issues, and pull requests. However, they encountered some challenges during the data analysis process. They used the Apriori algorithm for extracting association rules in the dataset and K-means algorithm to discretize the feature values as the algorithm only works with discrete values and most GitHub features are given in continuous values. One of the challenges in this step was the Zipf distribution of the number of downloads among the projects in the dataset. This distribution made it difficult to set the support threshold for association rule mining, as it would exclude rare itemsets related to popular projects. To address this, the authors used a confidence threshold of 0.75 and kept rules that detected rare itemsets, while also ensuring that certain itemsets related to popular projects were included in the rules. The authors were able to produce six association rules for successful projects, using the number of downloads as a proxy for project success. They found that successful projects tended to exhibit characteristics such as maturity, high activity, support to other users, active owners, small development teams, and owners with a small number of other projects, and followers.

2.5.3 Vulnerability Prediction using Machine Learning

A number of studies have been conducted in the field of software vulnerability prediction, exploring various techniques and models. These studies aim to use historical vulnerability data to predict future vulnerabilities in software systems and to improve software security.

One of the early works in this area is by Edkrantz [16], who investigates the use of historic vulnerability data from the NVD and the Exploit Database to predict exploit likelihood and time frames for unseen vulnerabilities. In the study, the author evaluates Naive Bayes, three different SVCs, Random Forest, k-Nearest Neighbors, and a Dummy classifier, and found that the linear time SVM algorithm achieved the highest prediction accuracy of 83 %. The study also found that the most important features for vulnerability prediction were common words from vulnerability descriptions, external references, and vendor products. Furthermore, the research demonstrated that when a considerable number of common words were used, the NVD categorical data, CVSS scores, and CWE numbers became redundant, as this information was usually included in the vulnerability description.

Another study by Han et al. [19] proposed a machine learning model to predict the severity of software vulnerabilities using only the vulnerability description. The authors found that their Convolutional Neural Network (CNN) model was able to extract discriminative features of vulnerability descriptions and outperformed other methods in predicting vulnerability severity.

Kamal and Raheja [30] also studied the use of three machine learning algorithms, random forest regressor, K-nearest regressor and decision tree regressor, to predict the severity of software vulnerabilities. The study used data from NVD to predict the CVSS v.3.* scores of un-scored software vulnerabilities in the database and found the random forest regressor to perform best.

Chowdhury and Zulkernine [8] studied the use of complexity, coupling, and cohesion (CCC) metrics to predict vulnerabilities in software. The authors presented a framework to automatically predict vulnerabilities based on CCC metrics and conducted a large empirical study on fifty-two releases of the Mozilla Firefox software. The study used four alternative data mining and statistical techniques, including C4.5 Decision Tree, Random Forests, Logistic Regression, and Naive Bayes, and was able to correctly predict a majority of the vulnerability-prone files. Furthermore, the authors found that decision-tree based techniques such as C4.5 Decision Tree and Random Forests significantly outperform Logistic Regression and Naive Bayes. The result of the report shows an accuracy over 90 % but a low recall. The authors explain that this is a consequent of the imbalanced dataset where the predictor becomes biased towards the overwhelming majority category.

Zhang et al. [61] conducted an empirical study using the NVD to predict *time to next vulnerability* (TTNV) in software and compared the performance of logistic regression, decision tree, and Artificial Neural Network (ANN) models. The authors used time, version (difference between two adjacent instances of the versions), software name, and CVSS as predictive attributes and evaluated the models using accuracy, precision, recall, and F1-score. The results showed that the data in NVD generally had poor prediction capability, except for a few vendors and software applications, e.g., the models built for Firefox. The authors discussed the main reasons for the difficulty of building good prediction models

for time to next vulnerability (TTNV) from NVD and highlighted the low quality of the NVD data as a main factor, such as missing information, vulnerability release time, and data errors.

A more recent study by Jabeen et al. [26] focused on the application of machine learning techniques for predicting software vulnerabilities. The authors recognize the importance of addressing issues related to Vulnerability Discovery Model (VDM) and aimed to determine the effectiveness of machine learning techniques in predicting future trends in vulnerability datasets. Through their empirical study, the authors compare nine machine learning methods and three statistical VDMs, evaluating their performance using both goodness of fit and predictive power criteria. The results of the study show that machine learning models generally outperform the statistical methods, with the ANFIS model yielding the best results for both performance metrics evaluated.

3 | Methodology

This chapter outlines the general methodology of the thesis, and Figure 3.1 shows an overview of the workflow. Before the workflow began, a literature study was conducted which is presented in 2.5. This literature study establishes the foundation for the workflow implementation and the selection of techniques employed. The process started with collecting and preparing the dataset, which served as the basis for building the feature space matrix used to apply data mining and machine learning techniques. Following this, the data mining techniques, association rule mining, isolation forest, and clustering, were applied to the dataset to help identify patterns and relationships in the data, thus partially answering Research Question 1. Simultaneously, the dataset was divided and three machine learning models were trained and evaluated. Combined with the methodology for applying data mining techniques, this provided the results for answering Research Question 1. To answer Research Question 2, each model was evaluated using the performance metric accuracy. Furthermore, the relative impact of the independent variables were evaluated using feature importance analysis on the trained machine learning models, thus addressing Research Question 3. Finally, to provide results for answering Research Question 4, the machine learning models were used to predict the severity level of a repository.

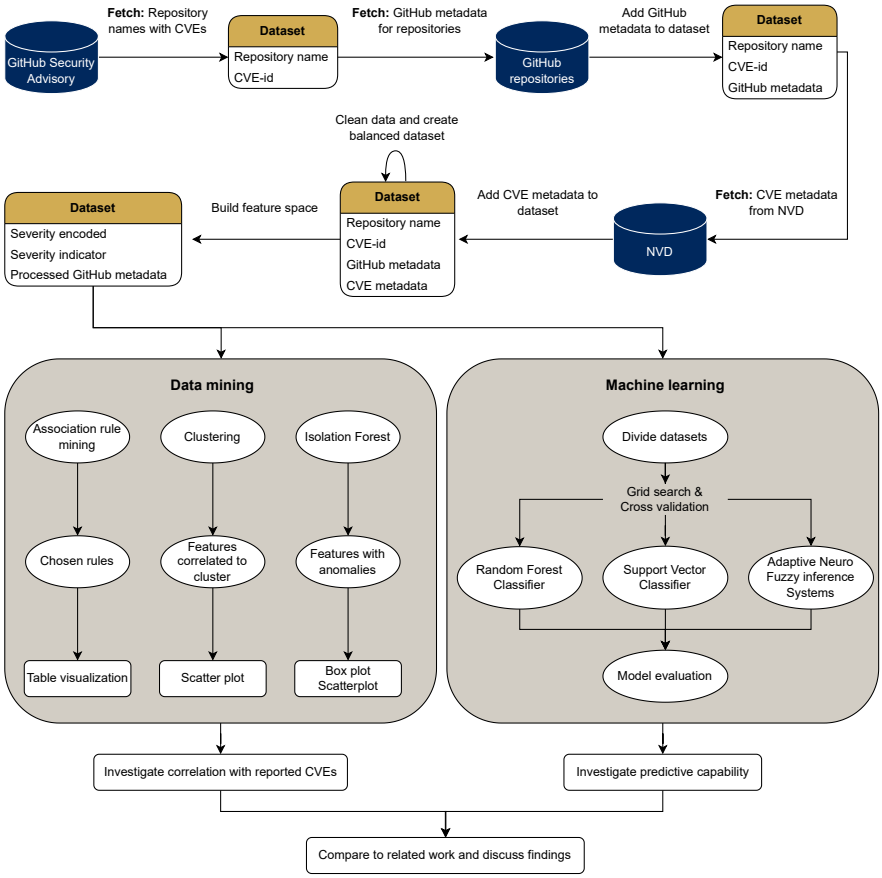

Figure 3.1: Overview of the applied methodology

3.1 Programming Tools

The programming tools used in this thesis were primarily based on Python, with additional libraries such as *numpy* [1] and *pandas* [2], used for their respective functionalities. *Numpy* was especially used for numerical computations and mathematical operations, while *pandas* was utilized for creating dataframes and preparing data. Additionally, *scikit-learn* [3] was used for its implementation of various machine learning algorithms and statistical models.

To collect data from the external data sources, *Postman* [4] was employed for evaluating the APIs capability. This allowed for easy testing and exploration of the available endpoints and parameters, as well as the ability to quickly iterate and modify API calls as needed. Overall, the selection of these libraries was based on their well-established reputation, extensive community support, and robust functionality in the field of data science and machine learning.

3.2 Data Collection and Preparation

In this thesis, a large part of the work consisted of mining data to build a feature space matrix. More information about the chosen databases is presented in Section 2.1. This section describes how the data collection process was carried out to collect the data from GitHub and NVD.

3.2.1 Data Collection

To ensure high-quality data and a real-world context for accurately predicting vulnerability severity [27], a combination of these two sources of vulnerability data was chosen. Additionally, metadata from code repositories was utilized to achieve a lower false-positive rate [47]. GitHub was used in addition to NVD for collecting data, as studies have shown that certain metadata from GitHub is significant in predictive tasks and has correlations to the number of found vulnerabilities and reported CVEs [25] [24]. By integrating data from both sources, a more comprehensive understanding of the real-world context was aimed for and an increase in the accuracy of vulnerability prediction was expected.

GitHub Data Collection

GitHub is considered the main data source used in this study. GitHub provides several open APIs that enables users to extract information on security advisories and public repositories. The data collection from GitHub was divided into two steps:

1. Gathering data from the GitHub Security Advisory Database
2. Gathering information about the repositories corresponding to a reported security vulnerability in the GitHub Security Advisory Database

Initially, the plan was to use GitHub's GraphQL API to collect all security advisories and use keywords to search for the assigned repository. However, this approach was not feasible as the search API had a rate limit of 30 searches per minute. To overcome this issue, a repository called *security-advisories* was identified on GitHub [5]. This repository contained all listed security advisories with reported repositories. By using the GitHub REST API, it was possible to extract the URLs in the security advisories repository for each separate folder in the repository. Figure 3.2 shows an example API call where the marked URL represents the API URL for the folder *advisories*. By looping through the content URLs, the download URL could be extracted and used to download the security advisory information in JSON format. Figure 3.3 shows an example of the response given from the download URL which represents the content for a reported security advisory. This could be found in a similar format in the GitHub repository.

[1] NumPy: `https://numpy.org/`
[2] Pandas: `https://pandas.pydata.org/`
[3] Scikit-learn: `https://scikit-learn.org/stable/`
[4] Postman: `https://www.postman.com/`
[5] GitHub Security Advisory Database: `https://github.com/github/advisory-database`

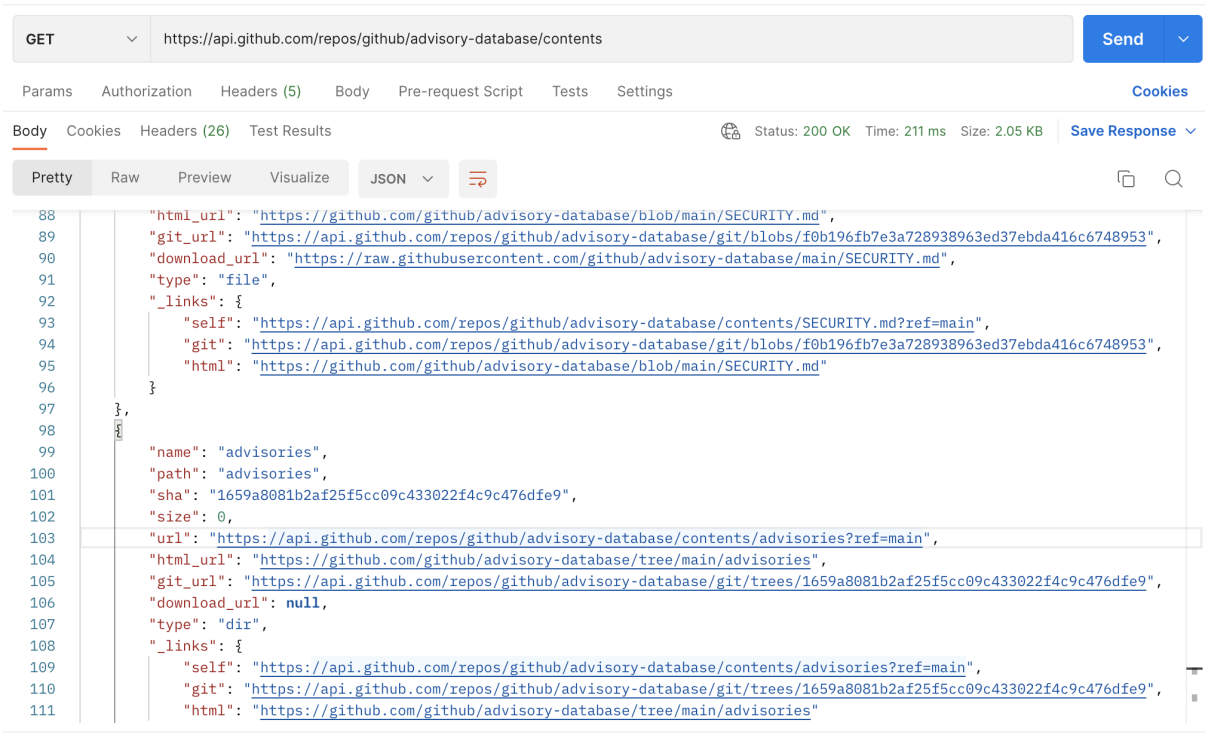

Figure 3.2: API call for GitHub security advisory content


```
{
  "schema_version": "1.4.0",
  "id": "GHSA-229r-pqp6-8w6g",
  "modified": "2023-01-25T22:54:47Z",
  "published": "2017-10-24T18:33:36Z",
  "aliases": [
    "CVE-2013-6421"
  ],
  "summary": "sprout Code Injection vulnerability",
  "details": "The `unpack_zip` function in `archive_unpacker.rb` in the sprout gem 0.7.246 for Ruby allows context-dependent attackers to
  "severity": [

  ],
  "affected": [
    {
      "package": {
        "ecosystem": "RubyGems",
        "name": "sprout"
      },
      "ranges": [
        {
          "type": "ECOSYSTEM",
          "events": [
            {
              "introduced": "0"
            },
            {
              "last_affected": "0.7.246"
            }
          ]
        }
      ]
    }
  ],
  "references": [
    {
      "type": "ADVISORY",
      "url": "https://nvd.nist.gov/vuln/detail/CVE-2013-6421"
    },
    {
      "type": "WEB",
      "url": "http://vapid.dhs.org/advisories/sprout-0.7.246-command-inj.html"
    },
    {
      "type": "WEB",
      "url": "http://www.openwall.com/lists/oss-security/2013/12/03/1"
    },
    {
      "type": "WEB",
      "url": "http://www.openwall.com/lists/oss-security/2013/12/03/6"
    }
  ],
  "database_specific": {
    "cwe_ids": [
      "CWE-94"
    ],
    "severity": "HIGH",
    "github_reviewed": true,
    "github_reviewed_at": "2020-06-16T20:50:49Z",
    "nvd_published_at": null
  }
}
```


Figure 3.3: Example result from the download URL

This approach increased the rate limit from 1800 requests per hour to 5000 requests per hour for a specific repository. However, the GitHub REST APIs limitations have been discussed in several studies [9] [7] [51], and with over 11,000 security advisories, it was crucial to address this issue. Although the rate limit was increased, the amount of requests that needed to be handled exceeded 5000 per hour, as each request represented one folder in the GitHub Security Advisory Database which contained multiple folders to go through. To handle the limitation, multiple tokens were generated. The tokens were stored in an array, and when the rate limit was reached, the token used could be replaced and added to another array called *usedTokens*, allowing for a significant increase in the amount of fetches possible. If the token list was empty, the *usedTokens* list was used where the same token was employed until the rate limit was reset.

After retrieving all available advisories from the GitHub Security Advisory Database in the previous step, the repositories connected to a CVE were identified. Each security advisory had an attribute called *urls* which contained different URLs depending on the advisory. In most cases, the URL "Package" provided a direct link to the GitHub repository, which was extracted and added to a new column named *package* in the dataframe. However, for reported advisories with missing package URLs, an alternative solution was implemented. The process was divided into three steps:

1. A check was made to see if any of the URLs contained the keyword "github.com". If this was the case, the URL referred to a specific commit that could be used to locate the repository.
2. The packages that contained URLs were matched by name, owner, and language with rows that were missing a repository link. These rows referred to the same packages, so the link could simply be added to these repositories as well.
3. If none of the URLs contained the keyword "github.com", and no other rows contained the link to the repository, the GitHub REST API was used to search for the package given the name and ecosystem. The name value referred to the repository's name, and the ecosystem referred to the vendor. Additionally, the token solution was implemented to handle the rate limitation of the API. Figure 3.4 shows an example API call to retrieve the package name for the repository *openssl* with vendor *RubyGems*.

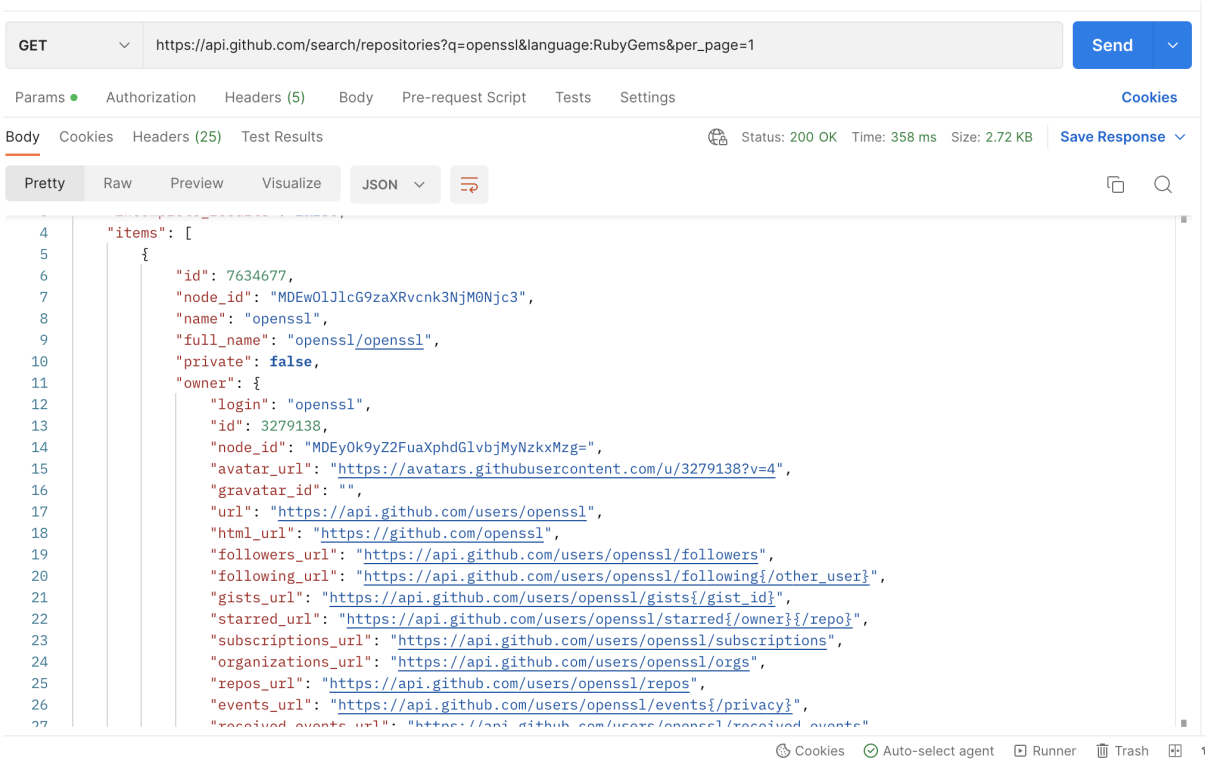

Figure 3.4: Example of the result when searching for repository

After extracting the repository identifications, the GitHub REST API was reused to extract information on the repositories with listed advisories. The API yielded complete matches throughout the dataset since the name and owner were already extracted in the format *owner/package-name*. To create a balanced dataset, as explained in section 3.2.3, this process was repeated for repositories without listed CVEs.

In summary, by utilizing the GitHub REST API, it was possible to retrieve a significant amount of informative data for each repository efficiently. Of the information returned by the API, the relevant data was added to the dataset, with the exception of dependencies and the number of commits, which were difficult to gather without having ownership of the repository.

NVD Data Collection

After creating the dataset with information from GitHub, the next step in the data collection phase involved collecting additional information on the identified CVEs using the NVD API. To ensure that the information collected from NVD pertained only to the CVEs that had corresponding repository information in the dataset, the NVD API was used to retrieve data only for those specific CVEs. Figure 3.5 shows an example API call to retrieve the NVD metadata for *CVE-2020-14422*.

The collected NVD metadata, including the severity of the vulnerability, the *CWE* category, and a textual description of the vulnerability was added to the existing dataset. Inconsistencies between the GitHub Security Advisory Database and the NVD were identified, and resolved by cross-checking the data. For instance, in cases where there was a disparity in the publication date of a CVE between the two data sources, the information from NVD was considered more reliable. This is because the NVD database is continuously updated and fully synchronized with the official CVE list. Consequently, any updates or changes to a CVE are promptly reflected in NVD, ensuring its timeliness and accuracy [12]. This enriched information allowed for a more detailed analysis of each CVE, thus providing a better understanding of the potential risks and impacts of the vulnerabilities.

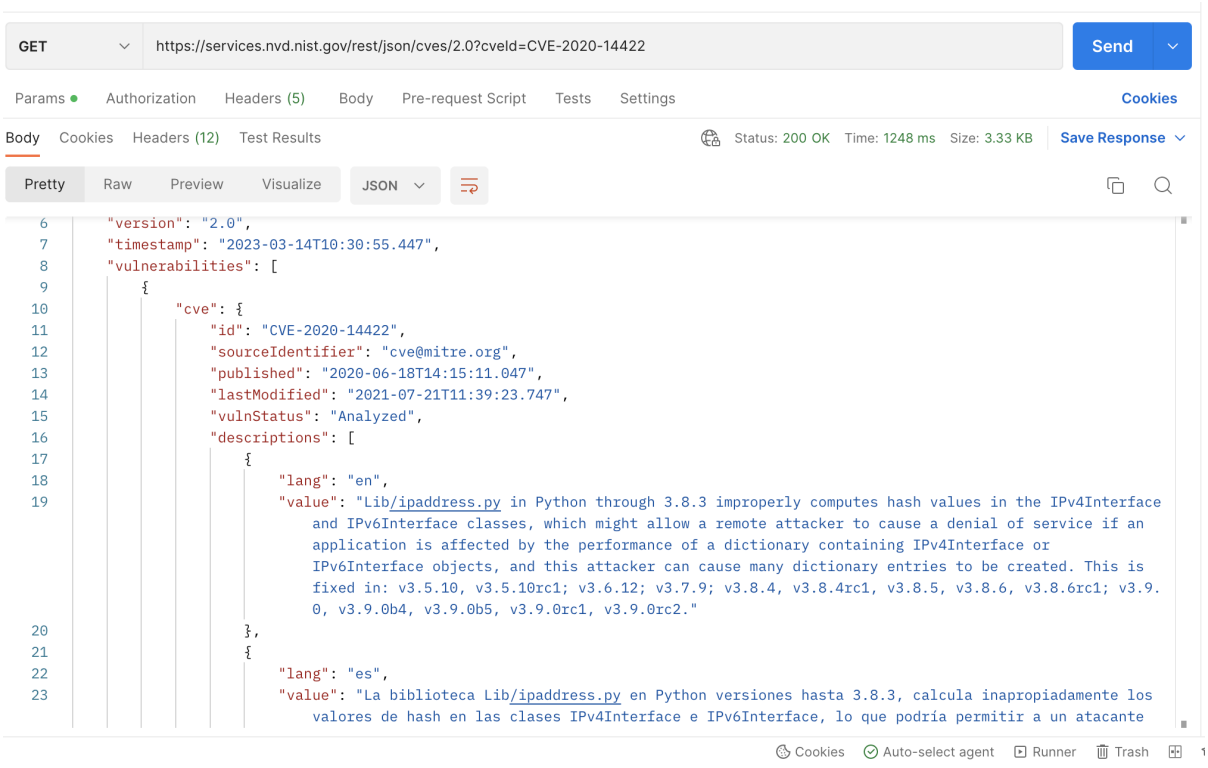

Figure 3.5: An example API call to collect information about a specific CVE from NVD

3.2.2 Data Cleaning

To clean and preprocess the raw data, duplicates were identified using the unique identifier for each vulnerability (CVE). Additionally, data points with missing values for GitHub features, such as repository size and number of contributors, were dropped. After this step, missing values were still present in other features, such as language, repository topics, organization name, CVE information, and CWE. To address missing values for language and organization name, null values were treated as a new categorical feature called "NA". Furthermore, rows lacking the CVSS score were excluded due to the absence of reported severity details. Without such information, it is impossible to predict the severity of vulnerabilities with a ground truth value. Finally, the features that had a majority of missing values were dropped from the data set, for example the *vendor* with 89% missing values.

As the CVE had been used to collect repository metadata, there were many duplicate rows based on the GitHub repository, i.e., based on the column *package_name*. As seen in Table 3.1, the repository *tensorflow/tensorflow* contained almost 400 reported CVEs. To avoid bias in the models' training, a choice was made to merge these into one row, although they represented different CVEs with different severity levels. To handle these duplicates, the column values for numerical information such as repository size and the number of contributors were added, while for the non-numerical values such as language, name, vendor, and severity level, the most frequent value was used.

Package name	**Unique CVEs**
tensorflow/tensorflow	399
k0xx11/bug_report	114
JacksonGL/NPM-Vuln-PoC	112
jenkinsci/jenkins	97
rails/rails	97

Table 3.1: Five packages with most reported CVEs

3.2.3 Creating a Balanced Dataset

Studies have highlighted that there is a significant imbalance between vulnerable and non-vulnerable observations in datasets [8], and as a result, there is a need to balance the dataset to accurately represent the population of interest. However, other studies have emphasized the importance of maintaining the real-world context of the data [27]. Therefore, it is crucial to ensure that the created dataset accurately represents the population of interest while also taking into account the need for real-world context.

As previously described, the dataset used in this study was collected from GitHub and NVD representing repositories with reported CVEs. These repositories provide the ground truth and represent the true positives in the prediction task, which are labeled as the positive class. However, it is also necessary to have data for repositories that do not have reported vulnerabilities, which represent the negative examples and negative classes in the prediction.

To create this balanced dataset, all unique organization names were collected from the original dataset containing repositories with listed CVEs. Using the GitHub REST API again, all repositories created by each unique organization were fetched, excluding repositories with reported CVEs. This implementation ensures that, at least for the repositories with a listed organization name, the ratio between vulnerable and non-vulnerable repositories represents the real world. It should be noted that this approach creates a dataset with more negative examples (repositories without reported CVEs) than positive examples (repositories with reported CVEs). This is consistent with the real-world setting, where there are typically more non-vulnerable software observations than vulnerable ones. As a result, it is appropriate to have more true negatives than true positives in the dataset.

3.3 Building the Feature Space

To build the feature space, feature extraction was performed to obtain a high-dimensional feature vector of binary and integer-valued features for each observation. However, not all of these features may be relevant, so a feature selection step was conducted to identify and select the most important variables for analysis. The dimensionality reduction technique PCA was also applied to improve efficiency and

handle multicollinearity between features. The feature space that was constructed in this step was utilized for both the data mining techniques and the predictive machine learning tasks in the next steps.

3.3.1 Feature Extraction

Feature extraction is a essential step in building the feature space. It involves transforming raw data into more meaningful and informative features that can improve the performance of the analysis. To create a high-dimensional feature vectors of binary and integer-valued features for each observation, the following steps were performed on the dataset:

- Extraction of base severity and base score from the column *metrics*.
- Creation of the new column *has_CVE*, indicating if the repository has a reported CVE.
- Standardization of date formats for all columns containing dates, such as repository creation or last update date. The time was removed to only keep the date in a standardized format.
- Creation of new columns containing the age of repositories (in days) and days since last updated.

Additionally, three bigger feature extraction steps were performed which are described in the subsections below.

Transforming the CVSS Score

As described in subsection 2.1.3, the CVSS score has evolved through various iterations over time, with each version having its own characteristics and methods for calculating the score. Moreover, the CVSS v2 score is generally lower than the newer versions, CVSS v3.*. In this dataset, most vulnerabilities included both metrics, CVSS v2, and v3. In these cases, the CVSS score for v3 was used as it is deemed the more accurate representation of the vulnerability. However, as some vulnerabilities only included the v2 score, they had to be transformed.

If the old scores were used in the same context as the updated score, it could lead to an inaccurate representation of the data. Therefore, the old CVSS score had to be updated to a v3 score. To accomplish this, the v2 vector was converted to v3 vector. This allowed for a more accurate comparison of severity across different vulnerabilities. Vector 3.1 and Vector 3.2 below shows how the same severity is represented by the two different vectors.

$$CVSS : 2.0/AV : N/AC : M/Au : N/C : N/I : P/A : N \tag{3.1}$$

$$CVSS : 3.*/AV : N/AC : L/PR : N/UI : R/S : C/C : L/I : L/A : N \tag{3.2}$$

To understand this pattern, various vectors were analyzed and the conversions were tested on observations that contained both scoring versions. This process resulted in a conversion that was accurate in predicting the severity level. Given that the thesis primarily concerns the severity category of a score, it was not necessary for the prediction score to be precise, as long as it yielded the same severity level. Once the new vector was generated, the *cvsslib* package could be utilized to convert the vector to a score, which could then be used to place the CVE in a severity category.

Encoding the Features

To ensure that the models can make predictions on the categorical features, the features need to be encoded [14]. The one-hot-encoding method was used which converts the categorical features into a set of binary features. For instance, if the categorical feature language had three values, Python, Java, and JavaScript, it was converted into three binary features, one for each category. If a sample belonged to the category Python, the binary feature for this category would be 1, and the binary features for categories Java and JavaScript would be 0. However, the encoding process was limited to features with a low number of unique categorical values. This was done to prevent an excessive increase in the dimensionality of the feature space, which could have negatively impacted the model's performance and interpretability.

Scaling the Data

Normalization or scaling of data is an essential step when dealing with features that vary in scales or units. As the dataset consists of features in different scales, such as the repository size in *byte* which have a high variance, compared to the number of open issues with a lower value. Standardization, a widely recognized normalization technique, was chosen for its effectiveness in numerous machine learning applications. For this process the *StandardScaler*, which utilizes Z-score normalization, from *sklearn* was used to scale the continuous features *repo_size*, *stars*, *forks_no*, *open_issues*, *contributors_count*, *pull_request*, *repo_days_since_update*, and *repo_age*.

Dimensionality Reduction

Dimensionality reduction is an important step in data analysis as it helps to reduce the number of variables while preserving important patterns and relationships. One common technique used for this purpose is PCA which is a statistical data mining technique that transforms high-dimensional data into a new set of uncorrelated variables called principal components (PCs). To determine whether to use PCA or not, a correlation matrix was created to identify variables with significant correlations. For variables with a high correlation, the Python library *decomposition* was used to perform a PCA to decompose the variables using principal components.

3.3.2 Feature Selection

Feature selection is a critical step in the data analysis process as it helps identify the most important variables for the analysis. By selecting a subset of variables, it can potentially improve the performance of the model and reduce overfitting.

As part of the feature selection methodology, relevant features were considered from both the GitHub repository metadata and the NVD. However, during the evaluation, it became apparent that the features extracted from the NVD were not suitable for online prediction due to the unavailability of offline sources or real-time additional information. Therefore, these features were removed from the feature selection to ensure the practical applicability of the machine learning model. Examples of features extracted from the NVD data include the CWE, when the CVE was published and modified, and the CPE. Despite their relevance in data analysis for vulnerable repositories, these features cannot be used in vulnerability prediction due to the unavailability of offline sources or additional information in real-time. Therefore, during feature selection, most features from the NVD were removed to ensure the practical applicability of the machine learning model for predicting the severity of open source repositories. The kept features from NVD represents the two target variables - severity level and an indicator to show if a data point contains a reported severity. No other features could be used to train the model, since all the features in the NVD, directly implies that a repository has a CVE.

3.4 Finding Patterns in the Dataset

Data mining techniques such as association rule mining, isolation forest, and clustering are useful in identifying patterns and relationships within the data [17]. Although they don't directly help in the predictive task, they are used to provide insights into the data and assist in identifying important features that can be used in predicting vulnerabilities.

3.4.1 Detecting Outliers with Isolation Forest

Isolation Forest is an unsupervised decision-tree-based algorithm used for outlier detection. This algorithm was implemented using the Scikit-learn library and trained on the independent variables from the feature matrix built in the previous step. All the default algorithm parameters were used, except for the random state, which was set to ensure reproducibility. The trained Isolation Forest model was used to predict whether an observation in the dataset was considered an outlier or not. The indicator was added to the dataset as the new column *outlier* which stored the predicted labels of potential outliers based on the given feature set.

To further investigate the identified outliers, the most significant features were extracted using the outlier detection algorithm object, which was an instance of the trained Isolation Forest model. Each

feature was independently shuffled and the outlier score was calculated using the shuffled feature values. The difference between the original outlier score and the shuffled outlier score was then computed and normalized with the standard deviation. This difference represented the feature importance score, which was stored in an array and sorted from top to bottom.

Furthermore, to visualize the result and study the outliers, a figure with the boxplots for the respective feature was created. These outliers could then be analyzed further to look into if a higher outlier score was correlated with severity or the severity level. Additionally, the data points classified as outliers with a high outlier score were investigated in order to identify their correctness and reliability. The new column *outlier*, was also included in the other data mining techniques, to further try and find correlations and patterns, based on the data points being classified as an outlier or not.

3.4.2 Grouping Data with Clustering

Clustering is a method used to group similar data points together to identify subpopulations in the data. For instance, if vulnerabilities tend to be concentrated in certain repository sizes or packages, clustering can identify these groups and highlight their significance to predict vulnerabilities. In this case, unsupervised clustering is used to explore the characteristics of repositories and identify clusters based on their similarity without the guidance of a target variable or label.

All features from the built feature space, including the *outlier* indicator was used in the analysis. This allows examination of the clusters to find the characteristics that are strongly associated with vulnerabilities, without using the indicator for an vulnerability. This method can complement the later stage prediction task and provide additional insights into the data that may not be captured by a supervised approach.

The first step was to select a clustering algorithm, and after reviewing the literature in 2.2.3, the K-means method was chosen as it is a widely used and reliable clustering algorithm. The next step was to determine the best number of clusters.

After considering the background research, the elbow method was selected. The elbow point indicated that six clusters were optimal. Once the optimal number of clusters was identified, the clustering algorithm was run to assign each item in the dataset to a cluster based on its similarity to other items in the cluster.

Additionally, a correlation matrix was created during the clustering process to find the variables that had the most impact on the clusters. These variables were visualized in a 3-dimensional scatter plot to further investigate if specific clusters correspond to the target variables. To measure the similarity of each data points to its own cluster compared to other clusters, the silhouette score was calculated using *silhouette_score* from *sklearn*. This information was useful as it allowed to understand the relationship between GitHub features and their impact on clustering, aiding in the interpretation and analysis of the resulting clusters.

3.4.3 Discovering Relationships with Association Rule Mining

Association rule mining can identify frequent itemsets and the relationships between them, which can be useful in identifying patterns in the data. For example, if a particular GitHub repository feature is frequently associated with vulnerabilities, association rule mining can identify this pattern and highlight the importance of this feature for predicting vulnerabilities. Furthermore, this technique can help identify interesting associations or co-occurrences between different features in the GitHub repository metadata.

As association rule mining only works on discrete values, the continuous values in the dataset had to be categorized. This was done by dividing the features with continuous values in five different bins using the K-means clustering algorithm, which has been highlighted as a suitable approach in related work [7]. These bins resembled the significance of each value, where *bin_0* indicates a value in the lowest quantile and vice versa for *bin_4*. After all features had been converted to discrete values, the library *mlxtend* was used to implement the Apriori algorithm.

The Apriori algorithm was used to identify association rules between the target variables, *has_CVE* and *severity_encoded*, and the independent features. The initial values for the *min_support* and threshold parameters were set to 0.1 and 0.7, respectively, to capture frequently occurring rules that represent at least 10% of the dataset and ensure the representability of the distribution of the target variable [7]. The

resulting rules were ranked by confidence, indicating the proportion of transactions that contain the first item in a rule that also contains the second item, and used to explore the underlying relationships among the variables. Subsequently, the *min_support* parameter was reduced to uncover less common but significant rules, and the process was repeated to gain further insights from the obtained results.

3.5 Predicting Vulnerabilities and Severity Level in Repositories

This chapter explains the methodology used to implement three different machine learning models to predict vulnerabilities in third-party open source software: SVM, Random Forest, and ANFIS. The selection of these models was based on the findings from 2.5.3, since these models had shown the best predictive performance in similar tasks. In the comparison study by Jabeen et al. [26], the ANFIS model displayed the highest predictive result when predicting the number of vulnerabilities reported for a specific software. In another study by Edkrantz et al. [16], the SVM model was used to predict the exploit likelihood and time frame for unseen vulnerabilities and yielded the highest prediction accuracy of the evaluated algorithms. Although the prediction target in these studies differs from the one investigated in this thesis, the success of the SVM and ANFIS model in similar tasks indicates its potential for evaluation in this study. Furthermore, prior research studies conducted by Han et al. [19] and Kamal and Raheja [30] have also reported promising predictive results for the target variable using neural network algorithms and random forest models, respectively, along with other independent features. These findings further support the suitability of these models for the current scenario, as they have previously demonstrated promising outcomes in predicting the same target variable. It is worth noting that Han et al. [19] focused solely on utilizing vulnerability descriptions to predict severity, while Kamal and Raheja [30] utilized data from the NVD specifically for un-scored vulnerabilities.

To begin with, the dataset was divided into training and testing sets using the stratified K-fold cross-validation method to ensure a balanced representation of the dependent variables in each fold [4]. In many similar prediction tasks, temporal intermixing needs to be considered by using training data from earlier time periods than the validation and testing sets so that information about the future don't "leak" into the model. However, as the independent variables only consists of GitHub repository features, temporal intermixing is not a concern as the repository features are not likely to change over time. Therefore, a common practice in developing machine learning models was planned to be applied by dividing the dataset into training, validation, and testing sets. However, due to the relatively small size of the dataset, a choice was made to only use training, and testing sets, but keep this in mind during the hyperparameter tuning process.

For all three models, the dataset was first divided into independent features (X), and the target variable (y). To ensure unbiased evaluation of the model, a stratified K-fold cross-validation was used with k=5. This means that for each fold, 80% of the dataset is used to train the model, while the rest 20% was used to evaluate the model. This was repeated using both *has_CVE* and *severity_encoded* as the target variable (y). After dividing the dataset according to the cross-validation, each model was implemented using different Python libraries or by developing the model from scratch.

3.5.1 Adaptive Neuro Fuzzy Inference System (ANFIS)

Initially, the plan was to use the anfis library [6] in Python to implement the ANFIS model. However, this library had some issues with the membership functions, thus making it unusable for this task. Consequently, a decision was made to implement the model using the Python library as guideline together with the *scikit-fuzzy* library and an OSS library on GitHub [7], that had shown promising results. To ensure the correct implementation of the ANFIS model, a validation process was performed using a public dataset. This involved comparing the obtained results with the published outcomes from the OSS library. By successfully replicating the published results, it was concluded that the model had been implemented accurately.

To start with, the data needed to be prepared which involved converting the dataset to an array of values. After this, input and output variables were defined and the fuzzy inference system was implemented using the *scikit-fuzzy* library. The input variables represented the independent features in

[6] https://github.com/twmeggs/anfis
[7] https://github.com/gabrielegilardi/ANFIS

the dataset, while the output was the target variable *has_CVE* or *severity_encoded*. The ANFIS model was trained by optimizing both the premise and consequent parameters using forward propagation and minimizing a cost function. The premise parameters capture the fuzzification of the input variables, determining their membership values. The consequent parameters, on the other hand, represent the transformation functions that map the fuzzified inputs to the output variable. Additionally, Particle Swarm Optimization (PSO) was used to optimize the hyperparameters *nPop*, and *epochs*

Additionally, as the model usually predicts continuous variables, the model had to be modified to handle discrete values. This step was done by developing a sigmoid function that converts the prediction to values between 0 and 1. Moreover, as the output represents a likelihood, a threshold of 0.5 was used to facilitate the interpretation of the results. Finally, the implemented ANFIS model was used together with the optimal hyperparameters and stratified K-fold cross-validation to predict the target variables.

3.5.2 Random Forest Classifier (RFC)

The RFC was implemented using the *scikit-learn* library in Python. Hyperparameter tuning was performed using a grid search approach with the following hyperparameters: *n_estimators* (number of trees in the forest), *max_depth* (maximum depth of the tree), *min_samples_split* (minimum number of samples required to split an internal node), and *min_samples_leaf* (minimum number of samples required to be at a leaf node). To avoid overfitting, regularization was achieved by choosing low grid search values for max_depth and high values for min_samples_split, and min_samples_leaf. The grid search was performed using the stratified K-fold cross-validation with 5 folds, and the optimal hyperparameters were chosen based on the highest cross-validation score. To evaluate the performance of the RFC, the model was trained on each fold in the cross-validation using the optimal hyperparameters obtained from grid search.

3.5.3 Support Vector Classifier (SVC)

Support Vector Classifier (SVC) was implemented using Scikit-learn library in Python. To evaluate the performance of the SVC model, hyperparameter tuning was performed using a grid search with cross-validation. The dataset was split into 5 folds using *StratifiedKFold* from Scikit-learn. The SVC model was trained and tested on each fold of the data, with the hyperparameters C of the error term, the kernel function to be used, and the kernel coefficient for the 'rbf' kernel function. The regularization parameter C controlled the tradeoff between maximizing the margin and minimizing the classification error on the training data. By choosing small grid search values for C, the risk of overfitting was reduced. To evaluate the performance of the best SVC, the model was trained on each fold in the stratified K-fold cross-validation using the optimal hyperparameters obtained from grid search.

3.6 Model Evaluation

This section highlights the methods used to evaluate the models and investigate the feature importance for each model. During the training process, the models were trained using an 80% portion of the data from the cross-validation set. Subsequently, the models' performance was assessed on the remaining 20% of the data for each class, which served as the testing set for each fold in the cross-validation.

3.6.1 Finding Important Features

Feature importance analysis is a technique used to determine which features are most important for a predictive model [52] [62]. Therefore, to address Research Question 3 and measure the relative impact of each independent variable on the target variable, feature importance analysis was performed on each trained model. The approach was only applied to the models used to predict the target variable *has_CVE* since this is the only aspect related to the research question. Furthermore, different methods were used depending on the machine learning model implemented [59].

For the RFC model, the feature importances were recorded for each fold in the cross-validation using the attribute *feature_importances_* for the built RFC model. The average feature importances across all folds was calculated and printed in descending order. A similar approach was applied for the SVC,

however, if the kernel function was *linear*, it used the *coef_* attribute for the built SVC model, otherwise it used *permutation_importance()* from the *sklearn inspection* package. For the ANFIS model, the weights of the different membership functions were extracted and sorted in descending order. These weights represent the degree to which the membership functions affect the output. As each membership function corresponds to an independent variable, the weights could be used to analyze which feature is of the highest importance for the model.

3.6.2 Performance Metrics

Research has demonstrated that using imbalanced datasets can lead to biased predictions towards the majority category, resulting in high accuracy but low recall [8]. To overcome this problem, previous studies have recommended incorporating additional performance metrics beyond accuracy alone [16] [8]. Therefore, to obtain a more comprehensive performance evaluation, the accuracy score was combined with precision, recall, and F1-score.

For all three models, the accuracy scores, and classification reports for each fold were computed, and averaged across all folds. The classification report provides precision, recall, and F1-score for each class (vulnerable and non-vulnerable observations) along with the average across both classes. The F1-score represents the weighted average of precision and recall, where a score of 1 indicates perfect precision and recall, and a score of 0 indicates poor performance.

4 | Results

This chapter presents a detailed description of the result based on the methodology applied. First, the results from the data collection and preparation step is described, followed by a description of the build feature space. The results obtained from the application of data mining techniques to the feature space are described subsequently. Finally, the results of predicting the severity level and severity indicator of a repository using the three different machine learning models - Random Forest, SVM, and ANFIS, are presented.

4.1 Collected Data

Data was collected from the data sources GitHub and NVD through their respective APIs. To gather data from GitHub, the two different steps described in the methodology section were followed. The column CVE from the dataset was used to gather information about the reported severities using the NVD API. This information was needed to address Research Question 1 and 2 as it provides the ground truth for the predictive task. The NVD metadata was added to the dataset with the collected information from GitHub, resulting in a dataset that contained 12890 rows of repositories with reported CVEs. Table 4.1 lists the extracted column values in this dataset.

Column	Type	Description	Source
id	String	CVE-id	Key
withdrawn	Boolean	Indicator if the CVE has been withdrawn	GitHub
vendor	String	The software ecosystem the package belongs to	GitHub
name	String	Name of the package	GitHub
package_name	String	Full name of the package *owner/package*	GitHub
repo_size	Integer	Size of the package in bytes	GitHub
stars	Integer	Number of stars	GitHub
a has_issues	Integer	Indicator if the repository has an *Issues* tab	GitHub
forks_no	Integer	Number of forks	GitHub
open_issues	Integer	Number of reported open issues	GitHub
watchers	Integer	Number of users watching the package	GitHub
language	String	The primary programming language	GitHub
repo_createdAt	Date	Date and time the package was created	GitHub
repo_updatedAt	Date	Date and time the package was last updated	GitHub
repo_topics	List	Topics associated with the package	GitHub
org_name	String	Organization responsible for the package	GitHub
contributors_count	Integer	Number of contributors	GitHub
pull_request	Integer	Number of pull requests	GitHub
CWE_Git	List	CWEs associated with the package	GitHub
CWE_NVD	String	CWE associated with the package	NVD
metrics	List	Quantitative measures of the CVE severity	NVD
vulnStatus	String	Current status of the vulnerability	NVD
cpeMatch	String	CPE identifier for the affected software	NVD

Table 4.1: Dataset columns with GitHub repository and NVD metadata

Duplicate rows were handled during the data cleaning step, specifically those based on the GitHub repository column *package_name*, as explained in the methodology section. Furthermore, features with

many missing values were removed from the initial dataset. After addressing the data cleaning steps, the cleaned dataframe consisted of 5091 rows representing unique GitHub repositories with reported CVEs. To create a balanced dataset with both positive and negative classes, additional repositories were collected using the method described in 3.2.3. Table 4.2 illustrates the distribution of vulnerabilities according to the provided labels. It is important to note that this analysis reflects the real-world context of organizations that had at least one repository with a reported vulnerability. Thus, the presented results offer insights into the vulnerability landscape within these organizations.

Data	Label	Count
Repositories with CVEs	Positive	5091
Repositories without CVEs	Negative	27747

Table 4.2: Balanced dataset between repositories with and without reported CVEs

4.2 Built Feature Space

The feature space was built using a combination of feature extraction and feature selection techniques. In accordance with the methodology described, the CVSS scores were transformed to accommodate different versions. This involved converting approximately 500 scores, with many receiving higher categorical values. A detailed breakdown of the conversions can be found in Table 4.3. Additionally, the resulting severity levels were encoded and are displayed in Table 4.4. This information was needed as it provides the ground truth for predicting the severity level, thus addressing Research Question 4.

CVSS v2	CVSS v3	Count
LOW	MEDIUM	36
MEDIUM	HIGH	278
MEDIUM	CRITICAL	80
HIGH	MEDIUM	4
HIGH	CRITICAL	109

Table 4.3: Mapping of severity level from CVSS v2 to v3

Severity Level	Encoded Severity	Count
NONE	0	27747
LOW	1	49
MEDIUM	2	1646
HIGH	3	1983
CRITICAL	4	1413

Table 4.4: The encoded severity level and their distribution

To identify variables with significant correlations, a correlation matrix was created which is shown in Figure 4.1. The variables *stars* and *forks_no* were found to have a correlation coefficient of 0.82. To reduce the dimensionality of these variables, they were decomposed into one principal component and added as a new column *stars/forks* to the dataset using PCA. Moreover, the correlation matrix shows that there are no significant correlations to the target variables. The feature with the highest correlation is stars which has a value of 0.24 for severity_encoded and 0.26 for has_CVE. However, the matrix shows that the feature open_issues has a relatively high correlation to stars and forks, of 0.47 and 0.43. Even though this is an interesting finding the correlation value is not significant enough for the variables to be combined, which is why they are kept separately.

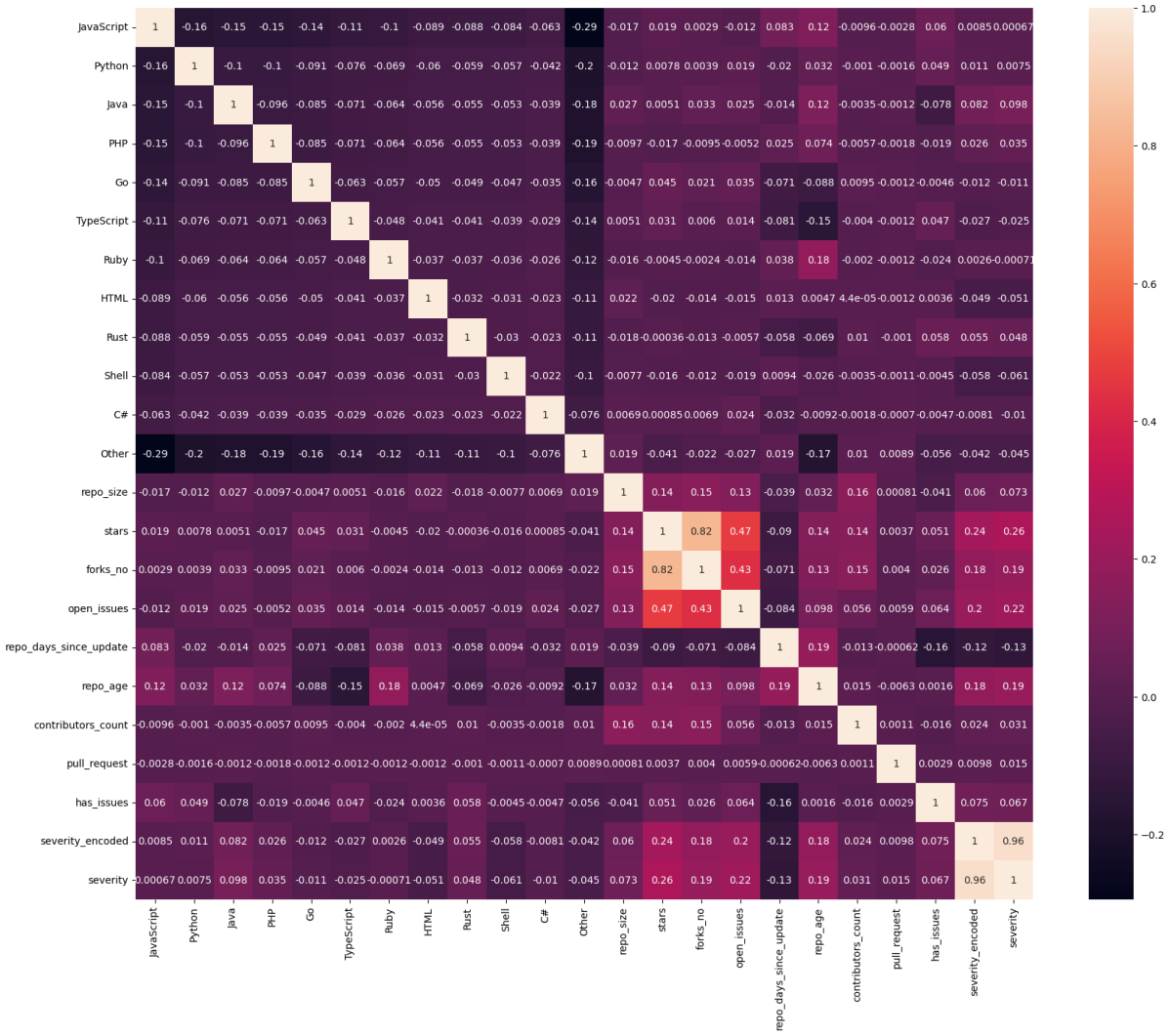

Figure 4.1: The correlation matrix for the feature space

In the feature selection phase, most of the features extracted from NVD were removed since all of these features could solely be added to repositories with reported CVE:s thus making the features unnecessary as they do not provide any additional information to predict the target variable. These features include *CWE_NVD*, *vulnStatus*, and *cpeMatch*. The features representing the number of forks and stars for the corresponding GitHub repository were also removed as they had been combined into a new feature using PCA in the previous step. The resulting feature space with encoded features is presented in Table 4.5. In this table, the two first rows represents the target variables, followed by the feature variables generated from Github. Additionally, the stars/forks variable resembles the conjoined variable of stars and forks, followed by the one hot encoded programming language.

Column	Type	Description	Range
has_CVE	Integer	Indicator if the repository has a reported CVE	[0, 1]
severity_encoded	Integer	Severity Level	[0, 4]
repo_size	Integer	Scaled size of the package in bytes	[0, 107]
repo_age	Integer	Scaled age of the repository	[-2, 3]
repo_days_since_update	Integer	Scaled days since updated	[-1, 5]
has_issues	Integer	Indicator if the repository has an *Issues* tab	[0, 1]
open_issues	Integer	Number of reported open issues	[0, 60]
contributors_count	Integer	Scaled number of contributors	[0, 78]
pull_request	Integer	Scaled number of pull requests	[0, 181]
stars/forks	Integer	Scaled principal component of stars and forks	[0, 74]
JavaScript	Integer	One-hot encoded programming language	[0, 1]
Python	Integer	One-hot encoded programming language	[0, 1]
Java	Integer	One-hot encoded programming language	[0, 1]
PHP	Integer	One-hot encoded programming language	[0, 1]
Go	Integer	One-hot encoded programming language	[0, 1]
TypeScript	Integer	One-hot encoded programming language	[0, 1]
Ruby	Integer	One-hot encoded programming language	[0, 1]
HTML	Integer	One-hot encoded programming language	[0, 1]
Rust	Integer	One-hot encoded programming language	[0, 1]
Shell	Integer	One-hot encoded programming language	[0, 1]
C#	Integer	One-hot encoded programming language	[0, 1]
Other	Integer	One-hot encoded programming language	[0, 1]

Table 4.5: Columns and their description of the built feature space

4.3 Found Patterns in the Dataset

The following section describes the outcomes of applying data mining techniques to the dataset presented in Table 4.5, with the purpose of identifying patterns and connections within the constructed feature space. The results from this section addresses the data mining aspect in Research Question 1 as it investigates patterns in vulnerable and non-vulnerable observations.

4.3.1 Detected Outliers

After implementing the isolation forest algorithm, the most significant features were extracted and sorted based on their outlier score. The extracted features, arranged in descending order, were as follows:

1. stars/forks
2. open_issues
3. pull_request
4. contributors_count
5. repo_size
6. repo_days_since_update
7. repo_age

Furthermore, the top features were visualized together as boxplots in a diagram. This process was carried out in order to further investigate the independent features. The resulting boxplots are displayed in 4.2.

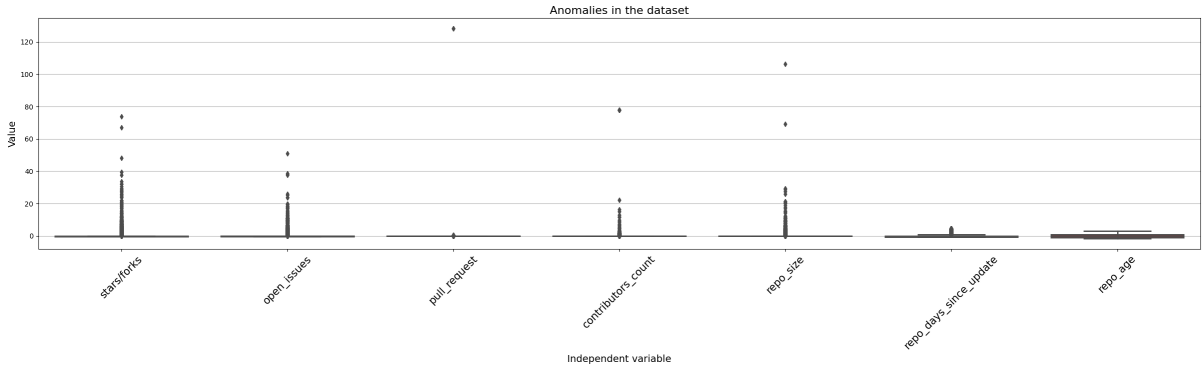

Figure 4.2: outlier boxplot of the top features

With the outliers visualized, it was possible to examine which of the top features included significant outliers. From the boxplot diagram and the outlier scores, it was concluded that the top five features seen in 4.2 contained significant outliers.

These features were further investigated and visualized in a scatterplot, where the color represented the outlier indicator. The color of the data points ranges from red to blue where red indicates an outlier, and blue indicates a non outlier. The scatterplots are displayed in Figure 4.3. Upon further investigation of the data points classified as outliers, it was found that out of the 1300 data points classified as outliers, 866 of them (67%), were repositories with reported CVEs.

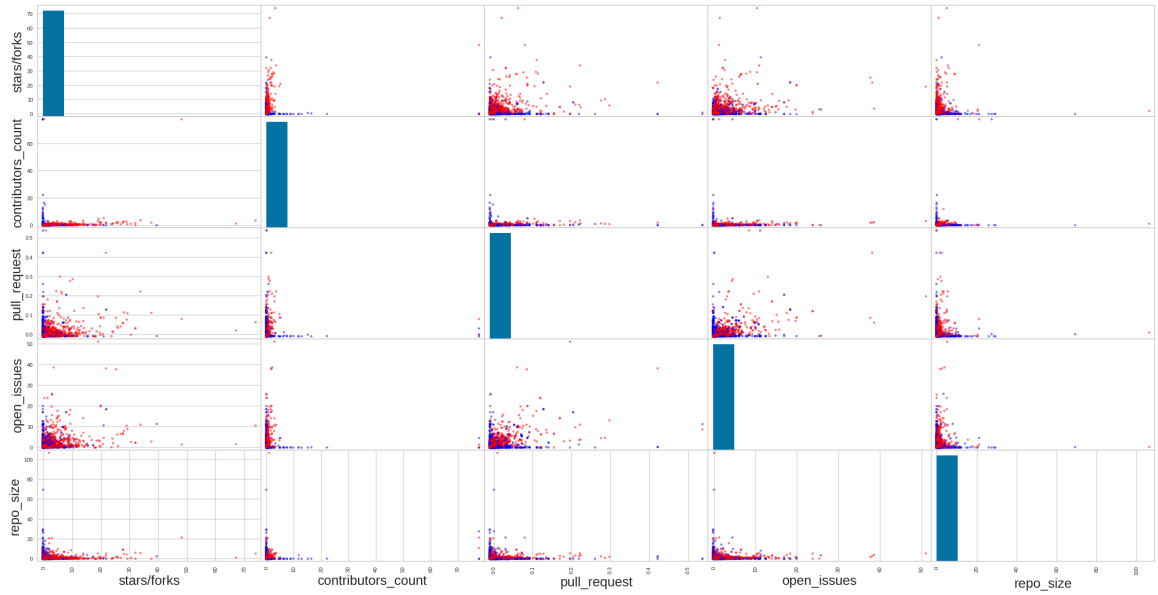

Figure 4.3: Scatterplots features with the highest outlier score

4.3.2 Grouped Data

The unsupervised K-means clustering method was applied on the features *repo_size*, *stars/forks*, *open_issues*, *contributors_count*, *pull_request*, *repo_days_since_update*, *has_issues*, *repo_age*, *outlier*, and the one-hot encoded language feature from the feature space. The results from the K-means clustering methods revealed that the optimal number of clusters for the features was 6, and the resulting distribution for each cluster is presented in Table 4.6. The majority of the repositories were assigned to cluster 0 and 1. The two clusters that contained a small number of data points was further investigated to assess the correlation with the outlier score. In Cluster 4, all data points, except for one, had an outlier score of -1, indicating an outlier. Cluster 5 had one outlier and one non-outlier. The average silhouette score for the clusters was 0.258.

Cluster	Count
0	12255
1	16361
2	3970
3	245
4	5
5	2

Table 4.6: K-means cluster distribution

Feature	Correlation
repo_days_since_update	0.611
repo_age	0.371
stars/forks	0.140

Table 4.7: Top cluster-correlated features

As described in the methodology section, the features that showed the highest correlation with the variable *cluster*, as indicated in Table 4.7, were used to create a 3-dimensional scatter plot. The corresponding plot is presented in Figure 4.4, where the color of each data point represents its cluster, and the symbol indicates the severity indicator *has_CVE*. However, due to the limited number of data points in certain clusters, such as cluster 5 with only 2 data points, it becomes challenging to visually identify these clusters in the figure. Analyzing the figure does not reveal any apparent patterns or relationships between specific clusters and the presence or absence of a reported CVE. Additionally, a 2-dimensional scatter plot illustrating the relationship between the features *repo_age* and *repo_days_since_update* is presented in Figure 4.5.

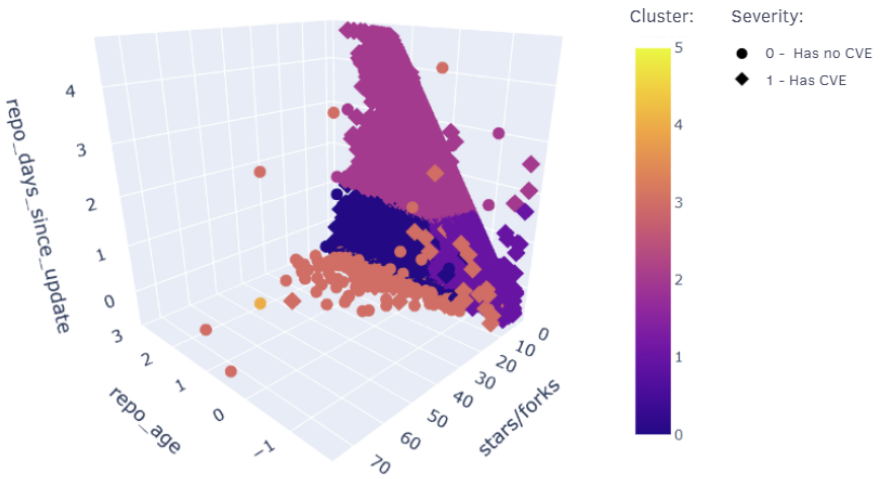

Figure 4.4: 3D scatter plot for top 3 correlated features to cluster variable

Figure 4.5: Scatter plot for repository age and last update with cluster variable

4.3.3 Discovered Relationships

In the methodology chapter, it is mentioned that the association rule mining technique was applied to the built feature space. Additionally, short labels were applied to the names of the different bins to improve the readability of the visualizations, displayed in Table 4.8.

Full name	**Short label**
has_issues	HI
open_issues_bin_0	OI0
repo_days_since_update_bin_0	RU0
repo_days_since_update_bin_2	RU2
repo_age_bin_3	RA3
contributors_count_bin_0	CC0
pull_request_bin_0	PR0
stars/forks_bin_0	SF0
stars/forks_bin_4	SF4
cluster_bin_0	CL0
cluster_bin_1	CL1
outlier_bin_0	A0
outlier_bin_1	A1

Table 4.8: Short labels generated for association rules visualization

Initially, the *min_support* parameter was set to 0.1 to identify rules which can be applied to at least 10% of the dataset. The top five most confident association rules with severity set to 0 as a consequent were extracted and are displayed in Figure 4.6. The figure represents each categorical value with a tag (bin), where the lowest quantile is denoted as *bin_0*, and the highest quantile is represented as *bin_4*.

The confidence values for the rules in the figure have been rounded and the rules themselves have been sorted in ascending order. The rule with the highest confidence corresponds to the antecedents *OI0, RU0, CL1, PR0*. This rule indicates that repositories with a low number of open issues, recently updated, assigned to cluster 1, and having few pull requests, are unlikely to have reported CVEs in 94% of the cases. The support value for this rule is 0.11, suggesting that it applies to 11% of the dataset. Overall, the generated rules exhibit variations in their parameters. However, certain parameters, namely *OI0, RU0, CL1* and *PR0*, appear in all the rules. This indicates that repositories having these features are less likely to contain a CVE.

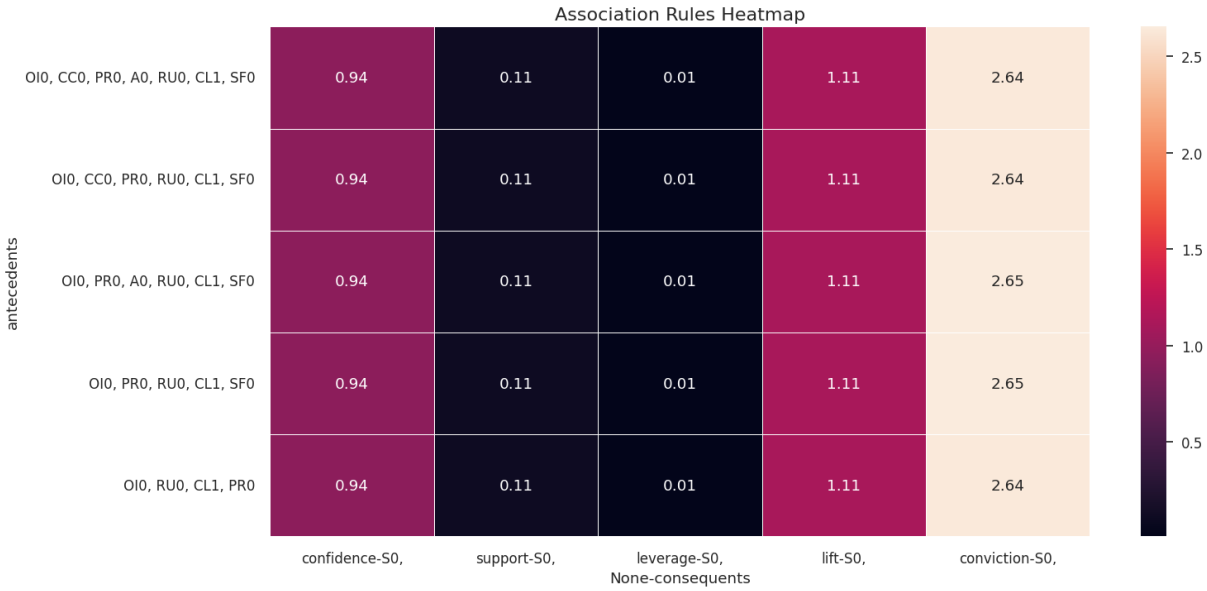

Figure 4.6: Top five rules with the highest confidence, support level 0.1 and severity set to 0

The next step in the procedure involved extracting the consequents that included a severity set to 1. However, none of these instances resulted in any association rules, indicating their infrequency. As a result, the value of the *min_support* parameter was reduced to 0.01 to identify rules that may have a lesser impact but could still be considered important. Figure 4.7 illustrates the top five association rules with the highest confidence.

In the figure, the rule with the highest confidence is the third rule, characterized by the antecedents *RA3, RU2, A1*. This rule suggests that among repositories that are older, have not been recently updated, and are classified as outliers, 83% of them contain a reported CVE. Notably, all the generated rules include the parameter *A1*, which indicates that if a repository is classified as an outlier by the isolation forest algorithm, it increases the likelihood of it containing a CVE.

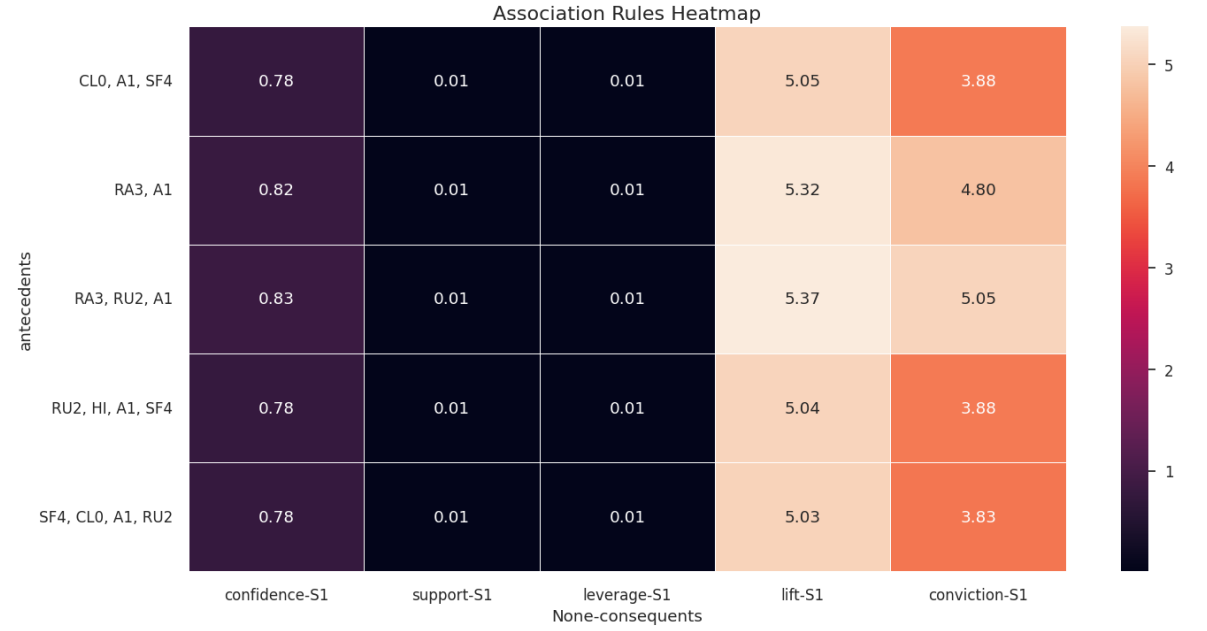

Figure 4.7: Top five rules with the highest confidence, support level 0.01 and severity set to 1

The above-described process was repeated, but this time with a focus on the target variable *severity_encoded* as consequents. As expected, the same association rules that were generated for *severity_encoded_bin_0*, was generated for *severity_bin_0* since they represent the same data points in the dataset. This result is illustrated in Figure 4.6. However, no rules were generated for the other severity levels, even when the support threshold was lowered to 0.05.

In summary, the findings for the generated association rules can be narrowed down to:

- Repositories with few open issues, recent updates, assigned to cluster 1, and few pull requests are less likely to contain reported CVEs. The antecedents *OI0, RU0, CL1, PR0* consistently appear in all generated rules, highlighting their relevance in predicting the absence of CVEs (*has_CVE* = 0).
- Repositories that are older, not recently updated, and identified as outliers have a higher probability of containing reported CVEs. The parameter *A1* consistently appears in all generated rules, indicating its significance in predicting the presence of CVEs (*has_CVE* = 1).
- No association rules were generated for different severity levels when applying association rule mining to the target variable *severity_encoded*.

4.4 Predicted Vulnerability of a Repository

This section presents the results for predicting if a GitHub repository's data point contains a vulnerability or not, based on the target variable *has_CVE* in the built feature space. These results are used to answer Research Question 1 and investigate how machine learning can be used to predict vulnerabilities in third-party open source software.

Firstly, the result for the overall algorithm benchmark is presented for the RFC, SVC, and ANFIS. This includes tuning the hyperparameters using grid search. Subsequently, the algorithms' predictive performance for identifying whether a repository has a reported severity or not is presented.

4.4.1 Algorithm Benchmark

The algorithm benchmark section presents the results of the hyperparameter tuning for three different machine learning algorithms: RFC, SVC, and ANFIS, the first two implemented using the scikit-learn library in Python and the third by scratch. Hyperparameters were optimized for all three algorithms using grid search, and the final choices for each respective model can be found in Table 4.9, highlighted in bold text.

Algorithm	Parameter	Values
RFC	Number of trees in the forest	*n_estimators* = [50, **100**, 200]
	Max levels in each decision tree	*max_depth* = [**None**, 5, 10]
	Min data points placed in a node before the node is split	*min_samples_split* = [**5**, 10]
	Min data points allowed in a leaf node	*min_samples_leaf* = [**2**, 4]
SVC	Penalty parameter C of the error term	*C* = [0.1, **1**]
	Kernel function to be used	*kernel* = ['linear', **'rbf'**]
	Kernel coefficient for 'rbf'	*gamma* = [**'scale'**, 'auto']
ANFIS	Number of agents (population)	*nPop* = [20, **40**, 60]
	Number of iterations	*epochs* = [100, **200**, 300]

Table 4.9: Optimal hyperparameters for target variable *has_CVE*

4.4.2 Algorithm Performance

The results of the machine learning algorithms for predicting the presence of vulnerabilities in GitHub repositories are presented in this section. The performance of the models in predicting the severity indicator (0 for no severity and 1 for severity) was evaluated using Stratified 5-fold cross-validation.

This implies that for each fold, 80% of the dataset is used to train the model, while the remaining unseen data points (20%) were used to evaluate the model.

To measure their predictive capability, the averaged accuracy and the values of the classification report were computed, thus addressing Research Question 2. Table 4.10 shows the averaged accuracy scores for the RFC, SVC, and ANFIS models. The RFC had the highest average accuracy of 0.917, followed by the SVC model with an accuracy of 0.871, and the ANFIS with an accuracy of 0.860.

Algorithm	Accuracy
RFC	0.917
SVC	0.871
ANFIS	0.860

Table 4.10: Averaged accuracy for target variable *has_CVE*

The results of the evaluation are summarized in Table 4.11, which displays the average classification report for the target variable *has_CVE*. The precision score represents the proportion of true positive predictions over the total number of positive predictions, while the recall score is the proportion of true positive predictions over the total number of actual positive instances. The F1-score is the harmonic mean of precision and recall, providing an overall performance score that considers both metrics.

Algorithm	Class	Precision	Recall	F1-score
RFC	**0**	0.92	0.99	0.95
	1	0.90	0.53	0.67
SVC	**0**	0.88	0.99	0.93
	1	0.75	0.25	0.38
ANFIS	**0**	0.86	0.99	0.92
	1	0.75	0.16	0.27

Table 4.11: Averaged classification report for the severity indicator based on 5-fold cross-validation

4.4.3 Identified Important Features

The feature importance analysis method 3.6.1 was conducted to determine the relative importance of each feature in predicting the severity of an issue for each model. The results from this analysis serves as the foundation for answering Research Question 3. Table 4.12 presents the three five averaged feature importances for the models. The most important feature for all three models is *stars/forks*, but the importance of other features varied depending on the model. The value of the feature importance score is measured relatively to the other features in order to identify which feature that is most important for each model.

Furthermore, different methods were used to calculate the feature importances for the RFC, SVC, and ANFIS models. For the RFC model, the feature importances were recorded for each fold in cross-validation, and the average feature importance's across all folds were calculated and printed in descending order. This approach provides an accurate and robust estimation of feature importance as it takes into account the variation across different folds of the data. For the SVC model, feature importances were computed by averaging the coefficients of each feature across all folds using *permutation_importance()*, which estimates the importance of a feature by measuring the decrease in performance when the values of that feature are randomly permuted. However, it is acknowledged that this approach may not provide an accurate estimate of feature importance as it assumes that the features are independent, which may not be the case in reality. Lastly, the ANFIS models feature importance score was calculated by investigating the value of each membership function, since each function represents a feature in the model. The value resembles the degree to which the feature was used. When referring to Table 4.12, it is crucial to acknowledge that the values exhibit varying ranges and therefore cannot be directly compared between models. Nevertheless, the objective of these results was to determine the most significant feature for each model. Consequently, the focus lies on the relative order of the importance scores rather than the precise numerical values.

Algorithm	Feature	Value
RFC	stars/forks	0.202
	repo_days_since_update	0.186
	contributors_count	0.148
SVC	stars/forks	0.026
	repo_age	0.007
	has_issues	0.003
ANFIS	stars/forks	70.43
	contributors_count	66.42
	repo_age	42.03

Table 4.12: Top 3 feature importances from 5-fold cross-validation

4.5 Predicted Severity Level of a Repository

This section presents the results of predicting the severity level of a GitHub repository's data point, based on the target variable *severity_encoded* in the feature space. By investigating how the models perform on the target variable, it addresses Research Question 4.

Firstly, the result for the overall algorithm benchmark is presented for RFC, SVC, and ANFIS, which includes tuning the hyperparameters using grid search. After this, the algorithms' classification performance for identifying whether a repository has a reported severity or not is presented.

4.5.1 Algorithm Benchmark

This section presents the results of the hyperparameter tuning for three different machine learning algorithms: RFC, SVC, and ANFIS. Grid search was performed for all three algorithms, and the final hyperparameter choices can be seen in Table 4.13.

Algorithm	Parameter	Values
RFC	Number of trees in the forest	*n_estimators* = [50, **100**, 200]
	Max levels in each decision tree	*max_depth* = [**None**, 5, 10]
	Min data points placed in a node before the node is split	*min_samples_split* = [5, **10**]
	Min data points allowed in a leaf node	*min_samples_leaf* = [**2**, 4]
SVC	Penalty parameter C of the error term	*C* = [0.1, **1**]
	Kernel function to be used	*kernel* = ['linear', **'rbf'**]
	Kernel coefficient for 'rbf'	*gamma* = [**'scale'**, 'auto']
ANFIS	Number of agents (population)	*nPop* = [20, **40**, 60]
	Number of iterations	*epochs* = [100, **200**, 300]

Table 4.13: Hyperparameter tuning for *severity_encoded* with chosen values in bold

4.5.2 Algorithm Performance

This section presents the machine learning algorithms performance for classifying the severity level for a datapoint representing a GitHub repository. The performance of the models in predicting the target variable *severity_encoded* was evaluated using Stratified 5-fold cross-validation. This implies that for each of the five folds, 80% of the dataset is used to train the model, while the remaining unseen data points (20%) were used to evaluate the model. The averaged accuracy scores of the five folds are presented in Table 4.14, indicating that the RFC model achieved the highest accuracy.

Algorithm	Accuracy
RFC	0.871
SVC	0.849
ANFIS	0.850

Table 4.14: Averaged accuracy for target variable *severity_encoded*

Table 4.15 shows the averaged precision, recall, and F1-score of RFC, SVC, and ANFIS for the severity levels *None, Low, Medium, High*, and *Critical*. The severity levels are represented by classes 0, 1, 2, 3, and 4. For RFC, the severity level None, i.e., no severity, has the highest precision score of 0.90, indicating that when the model predicts class 0, it is correct 90% of the time. For SVC and ANFIS, class 0 has the highest precision score of 0.86. It's important to note that all three algorithms have a precision and recall score of 0 for the severity level *Low*, indicating that they did not correctly predict any instances for this class.

The RFC model achieved the highest accuracy and classification performance when predicting severity levels in GitHub repositories. However, its performance was lower compared to predicting the presence of CVEs. None of the models achieved a precision higher than 0.5 for predicting a severity level except for the severity level None, representing repositories without reported CVEs. The RFC model demonstrated an averaged precision of 0.43 for severity levels 2-4, indicating a correct prediction rate of 43%. In summary, the RFC model performed best overall but improvements are needed for predicting severity levels, particularly for the severity level *Low*.

Algorithm	Class	Precision	Recall	F1-score
RFC	**0**	0.90	1.00	0.95
	1	0.00	0.00	0.00
	2	0.45	0.22	0.29
	3	0.43	0.28	0.34
	4	0.41	0.04	0.08
SVC	**0**	0.86	1.00	0.92
	1	0.00	0.00	0.00
	2	0.38	0.06	0.10
	3	0.36	0.06	0.11
	4	0.00	0.00	0.00
ANFIS	**0**	0.86	0.99	0.92
	1	0.00	0.00	0.00
	2	0.26	0.05	0.08
	3	0.29	0.07	0.11
	4	0.16	0.01	0.01

Table 4.15: Averaged classification report for each severity level based on 5-fold cross-validation

5 | Discussion

This chapter discusses the yielded results and the methodology used to gain these. Additionally, the work is discussed in a broader context, taking into account ethical and societal aspects related to the work.

5.1 Results

This section focuses on notable results that require commentary, and compares them to the related work. The discussion starts with the results obtained from constructing the feature space, including the correlation between features in the feature space. Subsequently, the results achieved through the application of data mining and machine learning techniques are examined and compared to earlier research.

5.1.1 Built Feature Space

The results of the feature space reveals some interesting insights into the data. The conversion of the CVSS score to v3 resulted in an increase in severity levels, indicating that the new scoring system may be more strict in evaluating vulnerabilities. This is in line with the work of Anwar et al. [1], which highlights the need for taking the version of the CVSS score into consideration.

Furthermore, when examining the correlation matrix for the target variable *severity_encoded* and *has_CVE*, it was found that *stars* (watchers) and *open_issues* have the highest positive correlation with vulnerability severity, followed by *forks_no* and *repo_age*. This suggests that popular repositories with a large number of open issues may be more prone to have vulnerabilities. These findings are consistent with the study by Horawalavithana et al. [24], who found that activities such as forking and watching correlated with the number of CVEs.

On the other hand, the negative correlation between vulnerability severity and *repo_days_since_update* suggests that repositories that have been updated more recently may have fewer vulnerabilities. This is contradicting with the findings of Ibrahim et al. [25], who found that committing, pulling, and branching had a moderate positive correlation with the number of vulnerabilities. The result in this thesis suggests that repositories that are frequently updated may have fewer reported CVEs. One of the reasons for this could be because updates can fix existing vulnerabilities and prevent new ones. However, this finding differs from those of Naveed [41], who found no significant correlation between frequency of vulnerabilities and repository metadata such as stars, lines of code, and other attributes.

The discrepancy in findings between this study and others can be attributed to the contextual variations in the research. Ibrahim et al. [25]'s study focused on a specific domain, analyzing the top 100 open source PHP projects on GitHub, while Naveed [41] analyzed a narrower dataset representing a single programming language. As a result, their conclusions regarding the influence of popularity factors on vulnerability presence may not be applicable to the broader scope of this research. It is important to recognize these contextual differences and acknowledge that the contradictions within the results offer an opportunity to contribute unique insights that complement existing knowledge.

5.1.2 Found Patterns in the Dataset

An analysis of the data during the outlier detection process revealed that most features contained significant outliers. Upon further investigation, it was determined that these outliers were actual repositories

with extreme values and were thus deemed relevant to the analysis. When examining the representation of vulnerable and non-vulnerable repositories in the dataset, it became apparent that repositories with reported CVEs were over-represented in the data points classified as outliers. Specifically, repositories with reported CVEs comprised nearly 16% of the total dataset, while in the dataset with outliers, they accounted for almost 67%. The successful identification of vulnerable repositories using the outlier detection algorithm isolation forest implies that this method can be a valuable tool for initial screening and prioritization of repositories with potential vulnerabilities. However, it does not negate the need for other methods, including machine learning techniques, in assessing and validating vulnerabilities. outlier detection can serve as an effective first step in identifying potential outliers, but further analysis and investigation using additional methods are still necessary to confirm the presence of vulnerabilities and assess their severity.

The majority of the repositories were assigned to the same clusters, indicating that the majority of the repositories in the dataset exhibit similar characteristics. Furthermore, there was no apparent correlation between the clusters and the severity level or severity indicator, suggesting that when using the unsupervised clustering technique, it did not group the repositories based on their severity level or severity indicator. This may be attributed to the severity of the issues not being strongly influenced by the features used for clustering or other unexamined features that better capture the severity level. Additionally, the average silhouette score for the clusters was relatively low, suggesting that the clusters were not well-defined. This results indicates that clustering alone is not a great method for identifying vulnerabilities in repositories based on the provided dataset. However, using different clustering algorithms and feature sets could yield different results and potentially reveal stronger correlations.

The strong linear relationship between *repo_age* and *repo_days_since_update* indicates a consistent pattern in their changes. This finding is not entirely unexpected since the age of a repository is directly related to the amount of time that has elapsed since its last update. In addition, the linear relationship could be partly due to the fact that they are calculated in a similar way, as both are based on the current date (date the dataset was created) and either the date the repository was created or the date it was last updated. This means that the two features are not entirely independent of each other and may contain some redundant information. However, the low correlation coefficient (0.19) suggests that other factors may be influencing their association. This could be due to outliers in the data or the presence of nonlinear or complex interactions between the variables. Therefore, while there may be a general linear trend between *repo_age* and *repo_days_since_update*, there could be instances where the relationship deviates from linearity. For example, repositories that are relatively new may have frequent updates initially but later stabilize, leading to a nonlinear pattern. These nonlinearities can reduce the overall correlation even if a linear trend is present. However, despite the low correlation, it is still valuable to include both features in the prediction model as they offer unique information and have the potential to enhance its performance.

Association rule mining revealed some patterns in the data worth discussing. Looking at the rules generated for *has_CVE* set to 0, indicating a repository without reported CVEs, one rule occurred with probability 94%. This rule indicates that when a repository belongs to cluster 1, which was the biggest cluster, have few or no open issues, pull requests, and have recently been updated, it doesn't contain reported CVEs with 94% probability. However, this rule occurs with a relatively low support value of 0.11, which indicates that it is applicable to 11% of the dataset. Nevertheless, the rule could still be useful in identifying patterns in the data and repositories which are unlikely to contain reported CVEs.

Considering that the data points with reported CVEs represented only 16% of the entire dataset, it is reasonable to expect a lower support value for the generated rules targeting repositories with CVEs. The lower occurrence of repositories with CVEs in the dataset naturally leads to a lower support value for these rules. However, it is important to exercise caution when interpreting the high confidence of these rules. While they indicate a strong association between older repositories, classified as outliers, and those that have neither recently nor formerly been updated with the presence of CVEs, the limited occurrence of these rules in just 1% of the transactions raises concerns about their reliability and generalizability. Therefore, relying solely on these rules for identifying repositories with reported CVEs may not be advisable.

Another interesting rule generated for the target variable *has_CVE* with a value of 1 as the consequent is related to repositories classified as outliers, belonging to cluster 0, and falling into the highest quantile for the *stars/forks* feature. According to this rule, a repository with these attributes has a 78% probability of containing reported CVEs. This finding aligns with the positive correlation observed

between the *forks_no* feature and the target variable *has_CVE* in the correlation matrix. The increased attention and scrutiny that popular repositories receive from the developer community may contribute to the higher likelihood of CVE discovery. However, it is important to note that popularity alone does not guarantee the presence of vulnerabilities, and other factors should be considered in assessing the security of a repository.

However, it is worth noting that no association rules could be generated for the target variable *severity_encoded*, which represents the severity level. This can be attributed to the infrequency and imbalance of the dataset, where the occurrence of severe vulnerabilities is relatively rare compared to non-severe ones. As time progresses and more vulnerabilities are reported, and more data points are collected, it is expected that more frequently occurring rules and patterns will emerge, providing greater insights into the relationship between the features and the severity level of vulnerabilities.

The three data mining approaches were applied to the same dataset, providing complementary insights into the vulnerability landscape of GitHub repositories. Despite their distinct methodologies, these approaches collectively contribute to a deeper understanding of the dataset. Each method explores different aspects of the data, revealing unique patterns, relationships, and potential vulnerabilities. The Isolation forest algorithm was effective in identifying repositories with reported CVEs, prioritizing their examination as potential outliers. Association Rule Mining uncovered interesting patterns and dependencies among various attributes, shedding light on factors associated with reported CVEs. Although the unsupervised clustering technique did not directly reveal a correlation with the severity indicator, it played a role in generating association rules related to CVEs.

While there may be some variations in the insights generated by these methods, their combined findings provide valuable information for organizations utilizing GitHub repositories. These insights assist in identifying repositories with potential vulnerabilities, and making informed decisions regarding repository selection and risk mitigation. Therefore, incorporating these findings into security strategies can prioritize vulnerabilities and improve vulnerability assessment and mitigation practices.

5.1.3 Predicted Vulnerabilities and Severity Level in Repositories

The results of this study suggest that the RFC model outperforms both the SVC and ANFIS models in predicting the severity and severity level of a repository. This is in line with the findings of Kamal and Raheja [30], who also used a Random Forest regressor to predict the CVSS v.3.* score of un-scored software vulnerabilities. However, that study used data from NVD to predict the score. Similarly, Chowdhury and Zulkernine [8] found that decision-tree based techniques such as random forest showed the best results in predicting vulnerabilities in software, also obtaining an accuracy over 90%, compared to the accuracy from RFC in this study of 91.7%. However, the study used different independent variables namely, complexity, coupling, and cohesion (CCC) metrics [8].

Comparing these results to other studies reveals some contradictions. For instance, Jabeen et al. [26] found that the ANFIS model performed better than SVM when predicting the vulnerability rate, whereas Edkrantz [16] showed that the SVM algorithm outperformed random forests when predicting exploit likelihood and time frame for unseen vulnerabilities. It's worth noting, however, that both of these studies used continuous variables, whereas the predictive target variable in this study was discrete, which may have impacted the results. Furthermore, both studies relied on the NVD as the primary data source for independent variables, while this study used GitHub data instead.

The results also show that the models have difficulty predicting the severity of a repository, especially for the severity level *Low*, which may be due to the imbalanced nature of the dataset, as discussed by Chowdhury and Zulkernine [8]. This may indicate that there are not enough samples in the dataset to build a reliable model for predicting the severity level *Low*, which is also seen in the encoded severity distribution where the severity level is represented with only 49 data points in total. Additionally, it is noteworthy that all three models have a higher precision than recall for the severity level *Medium, High*, and *Critical*, suggesting that they are better at identifying true positives than true negatives. This may be due to the fact that the models are trained on a dataset that is imbalanced towards the non-vulnerable cases, inflecting bias towards predicting lack of vulnerability.

The analysis of feature importance reveals that *stars/forks* is the most critical feature in predicting the severity of a repository. This finding is in line with the correlation matrix, which indicated that stars (watchers) had the strongest positive correlations with the dependent variable *has_CVE*. Furthermore, the study by Horawalavithana et al. [24] supports this result, as they found that activities such as

watching were associated with the number of CVEs. The feature importances for the other variables differ between the models, which may be due to the models' distinct characteristics.

5.2 Method

This section discuss the methodology applied to yield the results. The discussion begins by outlining the approach used to collect and prepare the data, including the process for creating a balanced dataset and ensuring real-world representation of the data. Next, the approach for constructing the feature space is discussed. The data mining techniques used are then discussed, with a particular focus on the use of observations identified as outliers. Finally, the methodology for making predictions about severity levels is discussed.

5.2.1 Data Collection and Preparation

The process of data collection and preparation is a crucial step in any machine learning study, and this research is no exception. One important aspect to consider when dealing with vulnerability data in GitHub repositories is the possibility of multiple reported severities for a single package. For instance, the *tensorflow/tensorflow* package has almost 400 connected CVEs. Therefore, it is necessary to carefully curate the dataset to avoid introducing any biases that may affect the results.

However, creating a balanced dataset is not an easy task, as highlighted by other studies [10] [17]. One of the challenges is representing the real-world context [27] without limiting the dataset to specific organizations. Additionally, removing duplicate rows based solely on the package name may not be the best approach, especially when dealing with repositories that have multiple reported vulnerabilities.

Another challenge when creating a balanced dataset is the lack of data for certain severity levels. For example, there may be a low number of vulnerabilities with the severity level *Low*. Although this accurately reflects the real-world context, it becomes difficult to train models on such a dataset, as shown in the results. One possible solution is to use oversampling techniques, but this introduces a trade-off between the risk of overfitting and increasing the sample size [10]. Additionally, this technique removes the real-world representation of the dataset by over-representing the vulnerable class. Excluding the severity level *Low* from the dataset could have been another approach. However, doing so would also remove important real-world context and potentially introduce bias and imbalance into the dataset. This would compromise the models' ability to accurately predict across all severity levels.

5.2.2 Building the Feature Space

The methods used for building the feature space is relevant to discuss in this study. Although the set of relevant features have carefully been selected to represent the GitHub repositories and their vulnerabilities, there are several additional features that could have been included.

For example, vendor and organization are two categorical values that could have been encoded and included in the feature space. This additional information could provide insight into the organizational structure of the repositories and their associated vulnerabilities. However, vendor was excluded due to the high percentage of missing values and their inclusion could have introduced bias and affected the accuracy and reliability of the analysis. Regarding the feature organization, it contained a large number of unique values, making it challenging to apply one-hot encoding without significantly increasing the dimensionality of the feature space. One-hot encoding all the unique values could have led to a sparse representation and potential overfitting. Therefore, the decision not to use one-hot encoding for organization aimed to maintain a more manageable feature space and mitigate the risk of overfitting.

Text mining techniques, such as a bag-of-words approach, could also have been applied to the feature *repo_topics* to gain more insight into the repositories' purposes. Several studies have used the vulnerability description to predict severity [19] and exploit likelihood [16] of vulnerabilities. This additional information could provide more context about the repositories and help the model better understand the potential vulnerabilities.

Another potential feature that could have been investigated is the versions of the GitHub repositories related to the reported vulnerabilities. Similar predictive works have been done to predict the time to the next vulnerability [61], and including version information in the feature space could possibly improve the model's performance.

Finally, it could be useful to consider the dependencies between packages as an independent feature in the feature space. This additional information would provide a more comprehensive view of potential vulnerabilities, particularly in understanding how a specific repository's use of another repository with reported CVEs may impact its own vulnerability profile.

5.2.3 Finding Patterns in the Dataset

In data mining, converting numerical values to categorical values is often necessary to apply certain algorithms such as the Apriori algorithm. However, this process can be challenging, particularly when dealing with outliers and skewed distributions, as highlighted by Chatziasimidis et al. [7]. The solution applied in this study was to bin the numerical values based on their distribution and by applying K-means clustering. This solution have been used in other studies [7], but other approaches could have been considered.

Another point of consideration was the potential strong linear relationship between repository age and days since the last update. While this relationship could have been handled differently, it is important to note that identifying and addressing such relationships is critical in data mining to avoid false findings. Additionally, there were no apparent relationships between the clusters and the vulnerability of a repository, indicating that it may be necessary to explore additional independent features to better capture the relationship between repository characteristics and the likelihood of reported CVEs.

In the context of outlier detection, including actual repositories with extreme values in the analysis raised questions about their treatment as outliers. While these repositories are clearly distinguishable in the charts, there is a trade-off between excluding them or retaining them, considering their rarity and potential value in understanding vulnerabilities. The decision lies in balancing the trade-off between preserving important information and introducing potential bias. Researchers should explore the significance of including these rare repositories, even if they exhibit weak support in association rules, to gain a comprehensive understanding of vulnerability patterns. By reframing the discussion in this manner, the focus shifts towards evaluating the merits of retaining these repositories for meaningful insights. Another important aspect to consider regarding the outliers, is the fact that the outlier indicator feature, *outlier*, occurred frequently in the rules generated by the association rule mining model. This finding could signify the importance of keeping the outliers as they may serve as indicators for vulnerabilities.

5.2.4 Predicting Vulnerabilities and Severity Level in Repositories

The ANFIS model was implemented with a relatively simple architecture due to the large number of features in the dataset, which could have required significant memory usage (up to one terabyte) if a more complex architecture was used. However, it is acknowledged that increasing the complexity of the model architecture could have potentially improved its accuracy. Furthermore, it is important to note that the ANFIS model had to be implemented from scratch, as no existing library could be used. To ensure the model's accuracy, it was tested on a dataset with known results, and the model's predictions was compared to those of another ANFIS model to ensure its correctness. However, it is acknowledged that undiscovered faults in the model's development could lead to erroneous results, and additional measures may be required to handle discrete independent variables.

The feature importance analysis revealed that certain features had a higher impact on vulnerability prediction than others. However, it is important to consider the limitations associated with the feature importance analysis. The assumption of feature independence in the SVC model may not always hold true in real-world scenarios. This assumption can impact the accuracy of the estimated feature importances. Additionally, while the calculated feature importances provide valuable information about the relative importance of features, they do not capture the full complexity and interactions within the dataset.

5.3 The Work in a Wider Context

In the context of this work, it is important to discuss the ethical and societal implications related to the use of data mining and machine learning techniques for predicting vulnerabilities in software reposi-

tories. The use of these techniques can have both positive and negative consequences that needs to be considered.

One of the positive consequences is that the use of these techniques can lead to more efficient and somewhat accurate detection of known vulnerabilities in third-party open source software. From a societal perspective, this can improve the security of software systems, which is essential in today's world where cyber threats are becoming increasingly frequent and more difficult to detect. Thus, by identifying known vulnerable software, data mining and machine learning techniques can help prevent cyber attacks and protect user data, therefore contributing to the overall security of digital systems.

On the negative side, there are several concerns of using machine learning techniques. For example, there are concerns about the potential biases that may be present in the data used to train the machine learning models, which could lead to discriminatory outcomes. As the data used in this thesis represents repositories with reported CVEs found in the GitHub Security Advisory Database, other data is not considered although those repositories may have reported CVEs. Another important ethical aspect that needs to be considered is the privacy of the data used to train and evaluate machine learning models. However, as the data used in this work was gathered from public sources.

6 | Conclusion

The purpose of this thesis was to explore the predictability of vulnerabilities in third-party open source software using data mining and machine learning techniques. A dataset was created using data from GitHub repositories, with reported CVEs used to build the positive class, and repositories without reported CVEs to build the negative class. Feature extraction and selection were applied to the dataset to create a high-dimensional feature space suitable for applying data mining and machine learning techniques.

Based on the findings, it was concluded that machine learning models can predict vulnerabilities in third-party OSS with an accuracy of 90%. Out of the three machine learning models, the RFC was found to be most suitable for predicting vulnerabilities in third-party OSS. The feature importance analysis revealed that the *stars/forks* feature had the highest significance in predicting security vulnerabilities. Further investigation is needed to understand the relationship between user count and vulnerability detection in OSS. Popular repositories attract more attention, increasing the likelihood of reported CVEs. This highlights the importance of community engagement in detecting vulnerabilities. Additionally, the study revealed that the imbalanced nature of the dataset made it difficult for the models to predict the severity level of a repository, especially the severity level *Low*. Moreover, the models had a higher precision than recall for the severity indicator, and severity level *Medium*, *High*, and *Critical*, indicating a bias towards predicting non-severity cases.

The results also indicate that data mining techniques can be used to predict vulnerabilities in third-party OSS. The outlier detection algorithm isolation forest was successful in identifying vulnerable repositories. Out of all the data points with listed CVEs, 17% were classified as outliers. In addition, 67% of all the outliers represented a vulnerable repository. This indicates its usefulness in identifying potential vulnerabilities in third-party OSS.

The clustering technique showed that most repositories had similar characteristics, with no apparent correlation between the clusters and severity level or severity indicator. This suggests that clustering may not be the most effective technique for predicting vulnerabilities, and alternative methods such as supervised learning may be more suitable.

The association rules revealed that repositories belonging to cluster 1, those having few or no open issues, pull requests, and which have recently been updated, are less likely to contain reported CVEs. However, it is important to note that the absence of reported CVEs does not guarantee the absence of unreported vulnerabilities. Repositories with limited community engagement may still be vulnerable, potentially containing undiscovered CVEs. Therefore, users should exercise caution and conduct thorough assessments of the security implications before using such OSS.

Furthermore, the study's findings indicate a high probability of older repositories, classified as outliers, containing reported CVEs. Additionally, repositories classified as outliers with a higher value of *stars/forks* also show a high likelihood of containing reported CVEs. However, it is worth noting that the higher community engagement associated with repositories having a higher value of *stars/forks* increases the likelihood of vulnerabilities being discovered. This suggests that the heightened attention and usage of the code contribute to the identification of vulnerabilities. Lastly, the imbalanced nature of the dataset and the infrequency of observations for the severity levels made it difficult to generate rules with target variable *severity_encoded*, indicating the need for more data.

To conclude, this study's findings contributes to the literature on predicting vulnerabilities in third-party OSS and shed light on the relative impact of various software characteristics on the likelihood of a security vulnerability. The potential of machine learning models for predicting the existence of known vulnerabilities and severity level of a vulnerability has been demonstrated, and these findings can be useful for improving software security practices. In addition, data mining techniques have been shown

to be effective for predicting vulnerabilities in third-party OSS, but the choice of technique should be carefully considered based on the specific dataset characteristics. A combination of unsupervised and supervised learning techniques may be the most effective approach for predicting vulnerabilities in third-party OSS.

6.1 Future Work

This study opens up several approaches for future work. One possible direction for this is to explore additional variables that impact the likelihood of a security vulnerability. These variables could be, code complexity, testing coverage, dependencies in the repositories, or number of commits. Additionally, future research could consider more extensive datasets and add databases from other sources than GitHub. One example could be to look into the *Pypi* database[1] and extract severities of the packages listed.

Additionally, it would be valuable for future research to explore the effectiveness of other machine learning algorithms, including deep learning models, in predicting vulnerabilities in third-party OSS. While deep learning models have demonstrated impressive performance in numerous fields, the ANFIS model employed in this study produced the poorest accuracy when compared to the other two models used. Thus, a hybrid model incorporating fuzzy logic and neural networks may not have been the optimal choice, and deep learning models may be more appropriate. Moreover, future work could investigate the use of ensemble methods, which combines multiple machine learning models, to improve the prediction accuracy and robustness of the models.

In future research, it would be interesting to investigate the transferability of machine learning models trained on OSS to commercial software, considering that commercial software has different characteristics such as proprietary code, different development practices, and varying levels of testing which could impact model performance. Additionally, it would be valuable to explore the practical implications of using machine learning models for predicting vulnerabilities in third-party OSS. For example, researchers could examine how software developers can leverage these models' predictions to enhance their software security practices and lower the likelihood of vulnerabilities.

[1]Pypi: `https://pypi.org/`

Bibliography

[1] Afsah Anwar, Ahmed Abusnaina, Songqing Chen, Frank Li, and David Mohaisen. "Cleaning the NVD: Comprehensive quality assessment, improvements, and analyses". In: *IEEE Transactions on Dependable and Secure Computing* 19.6 (2021), pp. 4255–4269.

[2] G. D. Apriliana, T Siswantining, D Sarwinda, and A Bustamam. "Analysis of data mining for classification of Obstructive Sleep Apnea in chronic obstructive pulmonary disease patients". In: *AIP Conference Proceedings*. Vol. 2242. 1. AIP Publishing LLC. 2020, p. 030023.

[3] *association_rules: Association rules generation from frequent itemsets*. 2023. URL: https://rasbt.github.io/mlxtend/user_guide/frequent_patterns/association_rules/.

[4] Daniel Berrar. "Cross-Validation". In: *Encyclopedia of Bioinformatics and Computational Biology*. Ed. by Shoba Ranganathan, Michael Gribskov, Kenta Nakai, and Christian Schönbach. Oxford: Academic Press, 2019, pp. 542–545. ISBN: 978-0-12-811432-2. DOI: https://doi.org/10.1016/B978-0-12-809633-8.20349-X.

[5] Witte G Booth H Rike D. "The National Vulnerability Database (NVD)". In: *National Institute of Standards and Technology, Gaithersburg, MD, [online]* (2013), pp. 1–22.

[6] Leo Breiman. "Random forests". In: vol. 45. Springer, 2001, pp. 5–32. DOI: 10.1023/A:1010950718922.

[7] Fragkiskos Chatziasimidis and Ioannis Stamelos. "Data collection and analysis of GitHub repositories and users". In: *2015 6th International Conference on Information, Intelligence, Systems and Applications (IISA)*. IEEE. 2015, pp. 1–6.

[8] Istehad Chowdhury and Mohammad Zulkernine. "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities". In: *Journal of Systems Architecture*. Vol. 57. 3. Elsevier. 2011, pp. 294–313. DOI: https://doi.org/10.1016/j.sysarc.2010.06.003.

[9] Valerio Cosentino, Javier Luis, and Jordi Cabot. "Findings from GitHub: methods, datasets and limitations". In: *Proceedings of the 13th International Conference on Mining Software Repositories*. 2016, pp. 137–141.

[10] Roland Croft, Yongzheng Xie, and Muhammad Ali Babar. "Data preparation for software vulnerability prediction: A systematic literature review". In: *IEEE Transactions on Software Engineering* (2022).

[11] Mengyao Cui. "Introduction to the k-means clustering algorithm based on the elbow method". In: *Accounting, Auditing and Finance* 1.1 (2020), pp. 5–8.

[12] *CVE and NVD relationship*. 2023. URL: https://cve.mitre.org/about/cve_and_nvd_relationship.html.

[13] *CWE VIEW: Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses*. Accessed on: January 26, 2023. URL: https://cwe.mitre.org/data/definitions/1387.html.

[14] Mwamba Kasongo Dahouda and Inwhee Joe. "A deep-learned embedding technique for categorical features encoding". In: *IEEE Access* 9 (2021), pp. 114381–114391.

[15] Mark Dowd, John McDonald, and Justin Schuh. *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education, 2006. ISBN: 9780321444424.

[16] Michel Edkrantz and A Said. "Predicting exploit likelihood for cyber vulnerabilities with machine learning". In: *Unpublished Master's Thesis, Chalmers Universıty of Technology Department of Computer Science and Engineering, Gothenburg, Sweden* (2015), pp. 1–6.

[17] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey". In: *ACM Computing Surveys (CSUR)* 50.4 (2017), pp. 1–36.

[18] Baptiste Gregorutti, Bertrand Michel, and Philippe Saint-Pierre. "Correlation and variable importance in random forests". In: *Statistics and Computing* 27 (2017), pp. 659–678.

[19] Zhuobing Han, Xiaohong Li, Zhenchang Xing, Hongtao Liu, and Zhiyong Feng. "Learning to predict severity of software vulnerability using only vulnerability description". In: *2017 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE. 2017, pp. 125–136. DOI: 10.1109/ICSME.2017.52.

[20] Sahand Hariri, Matias Carrasco Kind, and Robert J. Brunner. "Extended Isolation Forest". In: *IEEE Transactions on Knowledge and Data Engineering* 33.4 (2021), pp. 1479–1489. DOI: 10.1109/TKDE.2019.2947676.

[21] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*. Vol. 2. Springer, 2009.

[22] Marti A. Hearst, Susan T Dumais, Edgar Osuna, John Platt, and Bernhard Scholkopf. "Support vector machines". In: *IEEE Intelligent Systems and their applications* 13.4 (1998), pp. 18–28.

[23] Katsuhiro Honda, Satoshi Hyakutake, Seiki Ubukata, and Akira Notsu. "A hybrid robust ANFIS based on noise fuzzy clustering". In: *2021 Joint 10th International Conference on Informatics, Electronics & Vision (ICIEV) and 2021 5th International Conference on Imaging, Vision & Pattern Recognition (icIVPR)*. IEEE. 2021, pp. 1–7.

[24] Sameera Horawalavithana, Abhishek Bhattacharjee, Renhao Liu, Nazim Choudhury, Lawrence O. Hall, and Adriana Iamnitchi. "Mentions of security vulnerabilities on reddit, twitter and github". In: *IEEE/WIC/ACM International Conference on Web Intelligence*. 2019, pp. 200–207.

[25] Ahmed Ibrahim, Mohammad El-Ramly, and Amr Badr. "Beware of the Vulnerability! How Vulnerable are GitHub's Most Popular PHP Applications?" In: *2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA)*. IEEE. 2019, pp. 1–7.

[26] Gul Jabeen, Sabit Rahim, Wasif Afzal, Dawar Khan, Aftab Ahmed Khan, Zahid Hussain, and Tehmina Bibi. "Machine learning techniques for software vulnerability prediction: a comparative study". In: *Applied Intelligence*. Springer. 2022, pp. 1–22. DOI: 10.1007/s10489-022-03350-5.

[27] Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark Harman. "The importance of accounting for real-world labelling when predicting software vulnerabilities". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, 2019, pp. 695–705. DOI: 10.1145/3338906.3338941.

[28] Ian T Jolliffe and Jorge Cadima. "Principal component analysis: a review and recent developments". In: *Philosophical transactions of the royal society A: Mathematical, Physical and Engineering Sciences* 374.2065 (2016), p. 20150202.

[29] Ledisi Kabari and Believe Nwamae. "Principal Component Analysis (PCA) -An Effective Tool in Machine Learning". In: (May 2019).

[30] Navirah Kamal and Supriya Raheja. "Prediction of Software Vulnerabilities Using Random Forest Regressor". In: *Computational Intelligence: Select Proceedings of InCITe 2022*. Springer, 2023, pp. 411–424.

[31] Dervis Karaboga and Ebubekir Kaya. "Adaptive network based fuzzy inference system (ANFIS) training approaches: a comprehensive survey". In: *Artificial Intelligence Review*. Vol. 52. Springer, 2019, pp. 2263–2293. DOI: 10.1007/s10462-017-9610-2.

[32] Trupti A Kumbhare and Santosh V Chobe. "An overview of association rule mining algorithms". In: *International Journal of Computer Science and Information Technologies* 5.1 (2014), pp. 927–930.

[33] Yaguo Lei. "3 - Individual intelligent method-based fault diagnosis". In: *Intelligent Fault Diagnosis and Remaining Useful Life Prediction of Rotating Machinery*. Ed. by Yaguo Lei. Butterworth-Heinemann, 2017, pp. 67–174. ISBN: 978-0-12-811534-3. DOI: https://doi.org/10.1016/B978-0-12-811534-3.00003-2.

[34] Jake Lever, Martin Krzywinski, and Naomi Altman. "Points of significance: Principal component analysis". In: *Nature methods* 14.7 (2017), pp. 641–643.

[35] Robert A Martin, Sean Barnum, and Steve Christey. "Being explicit about security weaknesses". In: *Black Hat Briefings* (2007).

[36] Fabio Massacci and Viet Hung Nguyen. "Which is the right source for vulnerability studies? an empirical analysis on mozilla firefox". In: *Proceedings of the 6th international workshop on security measurements and metrics*. Association for Computing Machinery, 2010, pp. 1–8. DOI: 10.1145/1853919.1853925.

[37] *Microsoft Security Advisory (899588): Worm:Win32/Zotob.A and Worm:Win32/Zotob.B*. 2005. URL: https://learn.microsoft.com/en-us/security-updates/securityadvisories/2005/899588 (visited on 02/01/2023).

[38] Tyler Moore. "On the harms arising from the Equifax data breach of 2017". In: *International Journal of Critical Infrastructure Protection*. Vol. 19. Elsevier, 2017, pp. 47–48. DOI: 10.1016/j.ijcip.2017.10.004.

[39] Anthony J Myles, Robert N Feudale, Yang Liu, Nathaniel A Woody, and Steven D Brown. "An introduction to decision tree modeling". In: *Journal of Chemometrics*. Vol. 18. 6. Wiley, 2004, pp. 275–285. DOI: 10.1002/cem.873.

[40] Sarang Na, Taeeun Kim, and Hwankuk Kim. "A study on the classification of common vulnerabilities and exposures using naıve bayes". In: *Advances on Broad-Band Wireless Computing, Communication and Applications: Proceedings of the 11th International Conference On Broad-Band Wireless Computing, Communication and Applications (BWCCA–2016) November 5–7, 2016, Korea*. Springer. 2017, pp. 657–662.

[41] Muhammad Shumail Naveed. "Correlation Between GitHub Stars and Code Vulnerabilities". In: *Journal of Computing & Biomedical Informatics* 4.01 (2022), pp. 141–151.

[42] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. "Predicting vulnerable software components". In: *Proceedings of the 14th ACM conference on Computer and communications security*. Association for Computing Machinery, 2007, pp. 529–540. DOI: 10.1145/1315245.1315311.

[43] Pegah Nikbakht Bideh, Martin Höst, and Martin Hell. "HAVOSS: A maturity model for handling vulnerabilities in third party oss components". In: *Product-Focused Software Process Improvement: 19th International Conference, PROFES 2018, Wolfsburg, Germany, November 28–30, 2018, Proceedings 19*. Springer. 2018, pp. 81–97.

[44] William S Noble. "What is a support vector machine?" In: *Nature biotechnology*. Vol. 24. 12. Nature Publishing Group UK London, 2006, pp. 1565–1567. DOI: 10.1038/nbt1206-1565.

[45] *Open Source Software Policy*. 2022. URL: https://www.saab.com/contentassets/7937e98676a147f49fe8664aa899ea9f/open20source20software20policy.pdf.

[46] *OWASP Top 10*. 2021. URL: https://owasp.org/Top10/.

[47] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2015, pp. 426–437. DOI: 10.1145/2810103.2813604.

[48] Jamshid Piri, Hoda Ansari, and R Iran. "Daily Pan Evaporation Modelling With ANFIS and NNARX". In: *International Journal of Agricultural Research* 31 (Feb. 2013).

[49] Derek A Pisner and David M Schnyer. "Support vector machine". In: *Machine learning*. Elsevier, 2020, pp. 101–121.

[50] Kristin Potter, Hans Hagen, Andreas Kerren, and Peter Dannenmann. "Methods for presenting statistical information: The box plot." In: *VLUDS*. 2006, pp. 97–106.

[51] Md Omar Faruk Rokon, Risul Islam, Ahmad Darki, Evangelos E Papalexakis, and Michalis Faloutsos. "SourceFinder: Finding Malware Source-Code from Publicly Available Repositories in GitHub." In: *RAID*. 2020, pp. 149–163.

[52] Benedek Rozemberczki, Lauren Watson, Péter Bayer, Hao-Tsung Yang, Olivér Kiss, Sebastian Nilsson, and Rik Sarkar. "The shapley value in machine learning". In: *arXiv preprint arXiv:2202.05594* (2022).

[53] Amit Saxena, Mukesh Prasad, Akshansh Gupta, Neha Bharill, Om Prakash Patel, Aruna Tiwari, Meng Joo Er, Weiping Ding, and Chin-Teng Lin. "A review of clustering techniques and developments". In: *Neurocomputing* 267 (2017), pp. 664–681.

[54] Karen Scarfone and Peter Mell. "An analysis of CVSS version 2 vulnerability scoring". In: *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE. 2009, pp. 516–525. DOI: 10.1109/ESEM.2009.5314220.

[55] *Security*. 2022. URL: https://www.saab.com/products/security.

[56] Ketan Rajshekhar Shahapure and Charles Nicholas. "Cluster quality analysis using silhouette score". In: *2020 IEEE 7th international conference on data science and advanced analytics (DSAA)*. IEEE. 2020, pp. 747–748.

[57] *Software Bill Of Materials*. 2023. URL: https://www.cisa.gov/sbom.

[58] Siniša Sremac, Ilija Tanackov, Miloš Kopić, and Dunja Radović. "ANFIS model for determining the economic order quantity". In: *Decision Making: Applications in Management and Engineering* 1.2 (2018), pp. 81–92.

[59] Erik Štrumbelj and Igor Kononenko. "Explaining prediction models and individual predictions with feature contributions". In: *Knowledge and information systems* 41 (2014), pp. 647–665.

[60] Gustav Svensson. *Improving Vulnerability Assessment through Multiple Vulnerability Sources*. Printed in Sweden, Tryckeriet i E-huset, Lund. Sweden, Oct. 2020.

[61] Su Zhang, Doina Caragea, and Xinming Ou. "An empirical study on using the national vulnerability database to predict software vulnerabilities". In: *International conference on database and expert systems applications*. Springer. 2011, pp. 217–231. DOI: 10.1007/978-3-642-23088-2_15.

[62] Alexander Zien, Nicole Krämer, Sören Sonnenburg, and Gunnar Rätsch. "The feature importance ranking measure". In: *arXiv preprint arXiv:0906.4258* (2009).