CONTENTS OF THE REPORT

TABLE OF CONTENTS

INTRODUCTION

The ever-increasing use of technology in our daily lives has given rise to the need for secure and reliable communication systems. With the rise of the internet, computer networks have become the backbone of modern communication systems. However, this reliance on computer networks has also made them vulnerable to various forms of cyber-attacks, such as network intrusion. Network intrusion is the unauthorized access or attack on a computer network, with the intention of stealing sensitive data, causing damage to the system, or disrupting the network's normal operation.

The traditional methods of detecting and preventing network intrusions, such as firewalls and antivirus software, have limitations that have led to the development of more advanced methods, such as intrusion detection systems (IDS). An IDS software application monitors network traffic for signs of intrusion and alerts the system administrator in case of any malicious activity. IDS has become an essential tool in the defence against network intrusion. Machine learning (ML) algorithms have shown great promise in enhancing the detection capabilities of IDS. ML algorithms can learn from past network traffic and detect patterns that may indicate malicious activity. However, the effectiveness of these algorithms depends on the quality of the training data and the choice of ML algorithm.

Recently, researchers have proposed the use of hybrid machine learning models in IDS to improve their accuracy and reduce the false positive rate. A hybrid machine learning model combines two or more ML algorithms to create a more robust model that can better identify different types of network intrusion. Hybrid models can also reduce the complexity of the problem and improve the detection speed. In this research, we propose a hybrid machine learning model for network intrusion detection that combines the strengths of multiple ML algorithms. The model consists of a pre-processing module, a feature selection module, and a classification module. The pre-processing module is responsible for data cleaning and normalization, while the feature selection module selects the most relevant features from the dataset. The classification module is responsible for classifying network traffic as either normal or intrusive. The proposed hybrid model uses a combination of decision trees, support vector machine (SVM), and deep learning algorithms. The decision tree algorithm is used for feature selection, while the SVM algorithm is used for classification. The deep learning algorithm, specifically the convolutional neural network (CNN), is used to detect complex patterns in network traffic that may indicate advanced persistent threats (APTs).

The proposed hybrid model was evaluated using the NSL-KDD dataset, which is a widely used dataset for evaluating IDS. The results show that the proposed model outperforms the traditional SVM algorithm and other state-of-the-art hybrid models in terms of accuracy, precision, and recall. The model also achieves a low false positive rate, which is crucial in reducing the number of false alarms that can lead to the system administrator being overwhelmed with unnecessary alerts.

In conclusion, the proposed hybrid machine learning model is a promising approach to network intrusion detection that can improve accuracy and reduce the false positive rate. The proposed model can be used in real-world applications to enhance the security of computer networks and protect against network intrusion. Further research can focus on optimizing the hybrid model's performance and exploring its application to other domains.

LITERATURE REVIEW

1. **Ghorbani, A. A., Lu, W., & Tavallaee, M. (2009)** offers a thorough analysis of the theories, methods, and approaches employed in network intrusion detection and prevention. Threat modelling, security architecture, network topology, and network security policies are among the many issues connected to network security that are covered. The authors give an outline of the designs and components of these two types of systems as well as how they differ from one another. They also go through several detection techniques, such as signature-based and anomaly-based. The difficulties of intrusion detection are also covered by the authors, including the necessity for precise and rapid detection, the capacity to identify unidentified assaults, and the issue of false positives and false negatives. They discuss several approaches to overcome these difficulties, including the use of multiple detection strategies and the use of data mining techniques to cut down on false positives.

2. **Levin, I. (2000)** presents the outcomes of the KDD-99 conference's classifier learning competition. The goal of the competition was to create algorithms that could determine from various characteristics of the network traffic whether a network connection was legitimate or an attack connection. The author gives a summary of the methods employed by one of the competing teams, LLSoft, to create their classifier while concentrating on their findings. LLSoft employed a hybrid strategy to create its classifier, including various machine learning algorithms and feature selection methods. The paper comes to a close by contrasting the classifier's output with that of the other teams who participated. With an accuracy of 80.6%, the findings demonstrated that LLSoft's classifier outperformed other classifiers. The report also identifies a few contest's shortcomings and proposes topics for further study, including the demand for more varied datasets and the investigation of ensemble approaches for intrusion detection.

3. **Chebrolu, S., Abraham, A., & Thomas, J. P. (2005)** suggests a novel feature deduction and ensemble design-based approach to intrusion detection systems (IDS). The authors contend that conventional IDS that employ several aspects are frequently unsuccessful and wasteful. To improve the efficiency and efficacy of an IDS, they suggest minimising the number of characteristics employed in it. To increase the overall detection accuracy, the authors also presented an ensemble-based IDS, which integrates the outcomes of many IDS models. The ensemble technique is especially helpful when using a variety of IDS models, each with unique strengths and shortcomings. The KDD Cup 1999 dataset, a common dataset used to assess IDS performance, was utilised to test the suggested technique. The outcomes demonstrated that the suggested strategy outperformed conventional IDS techniques that make extensive use of characteristics.

4. **Mukkamala, S., Janoski, G., & Sung, A. (2002, May)** gives a comparison of neural networks and support vector machines (SVMs), two machine learning techniques, for the job of network intrusion detection. The method of discovering unauthorised access or harmful activity in a computer network is covered in the first section of the article, which also discusses the issue of network intrusion detection. The authors then go on to detail the SVM and neural network models they employ for intrusion detection. The SVM model is a binary classifier with a linear kernel, whereas the neural network model is a multi-layer perceptron with one hidden layer. The accuracy, false positive rate, and true positive rate are only a few of the performance criteria the authors use to train and assess these models. The tests' findings demonstrate that, in terms of accuracy and false positive rate, the SVM model performs better than the neural network model. The neural network model, on the other hand, has a greater true positive rate, making it more effective at identifying genuine assaults. The authors also discover that utilising an ensemble technique to combine the two models produces even better results.

5. **Heckerman, D., Geiger, D., & Chickering, D. M. (1995)** focuses on the issue of determining a Bayesian network's topology and parameters using a combination of statistical data and previous knowledge. In Bayesian networks, conditional probability tables (CPTs) and directed acyclic graphs (DAG) are used to graphically illustrate the relationships between random variables. Using statistical information and previous knowledge, the structure refinement stage entails exploring the space of potential DAGs and choosing the one that maximises a score function. The Bayesian information criterion (BIC), which is used by the authors, strikes a compromise between the model's complexity and its quality of fit to the data.

6. **Amor, N. B., Benferhat, S., & Elouedi, Z. (2004, March)** start by going over how crucial intrusion detection systems are to maintaining the security of computer networks. The two machine learning methods, Naive Bayes and Decision Trees are then introduced, and their use in intrusion detection is discussed. The outcomes of the author's analysis of the two algorithms on the KDD99 dataset are then presented. Additionally, they recommend more studies be done to assess additional machine learning algorithms and investigate the usage of hybrid systems that integrate different algorithms. Overall, the research offers insightful information about the advantages and disadvantages of Decision Trees and Naive Bayes in the context of intrusion detection systems. It serves as an entry point for additional research in this field and stresses the significance of selecting the appropriate algorithm for the particular requirements of the system.

7. **Vinayakumar, R., Soman, K. P., & Poornachandran, P. (2017, September):** The limits of conventional intrusion detection systems (IDS) based on signature and anomaly detection are discussed at the beginning, along with the issue of network intrusion detection. They contend that these methods may be less successful in identifying sophisticated and complicated assaults, which

may have subtle or concealed patterns that are challenging to spot using conventional methods. They discover that while deeper networks and smaller window widths often produce greater accuracy, they also need more processing power and resources.

8. **Mill, J., & Inoue, A (2004, July)** explain the usage of Support Vector Machines (SVMs) for network intrusion detection. As it includes detecting unauthorised access attempts or harmful activity within a computer network, network intrusion detection is a key field of research in cybersecurity.

The use of fuzzy logic in combination with SVMs for intrusion detection is also covered by the authors. Fuzzy logic makes decisions more flexible and can increase the precision of intrusion detection systems.

9. **Wang, J., Yang, Q., & Ren, D. (2009, July)** introduces a decision tree-based intrusion detection system. The authors contend that decision tree technology applies to network security and has been applied effectively in a variety of fields, including finance and medicine. The traffic is labelled as anomalous if the score exceeds a certain threshold. Using a mixture of typical and abnormal traffic from the DARPA 1999 intrusion detection dataset, the authors assessed the method. With a detection rate of 93.27% and a false alarm rate of 1.45%, the algorithm appears to be capable of identifying network intrusions.

10. **Farid, D. M., Harbi, N., & Rahman, M. Z. (2010)** by integrating the benefits of both systems, we hope to increase detection and minimise false positives. The Decision Tree method is used for feature selection and decision-making, while the Naive Bayes algorithm is used to simulate the probability distribution of features based on class labels. Overall, the research proposes an intriguing way to improve the accuracy of intrusion detection systems by combining the characteristics of Naive Bayes and Decision Tree algorithms and adapting to changes in network traffic patterns using a sliding window strategy. The suggested system can identify and prevent network intrusions in real-world applications.

11. **Sathya, S. S., Ramani, R. G., & Sivaselvi, K. (2011)** presents a discriminant analysis-based feature selection technique for the KDD intrusion dataset, which is a commonly used benchmark dataset for assessing intrusion detection systems. Feature selection is a crucial stage in the machine learning pipeline that entails picking a subset of relevant characteristics from a vast pool of data to enhance classification accuracy and minimise computation time. The method then employs discriminant analysis to determine the most discriminative characteristics for classification. The discriminant analysis technique assigns a weight to each characteristic to signify its relevance in distinguishing between different types of network data. For categorization, the characteristics with the greatest weights are chosen.

12. **Panda, M., & Patra, M. R. (2009)** uses clustering and Fast Fourier Transform (FFT) techniques to provide a hybrid approach to network intrusion detection.

The suggested approach's major goal is to increase intrusion detection accuracy while lowering false alarm rates. The authors presented a clustering-based classification strategy that leverages the COBWEB algorithm for the clustering and classification of network traffic data to achieve this goal. FFT is also used in this method to extract characteristics from network traffic data. Overall, the study presents an intriguing method for detecting network intrusions utilising a hybrid clustering and FFT-based methodology. The COBWEB method for clustering and classification, along with FFT for feature extraction, has proven beneficial in enhancing intrusion detection accuracy.

13. **Bhavani, T. T., Rao, M. K., & Reddy, A. M. (2019, November)** provides an investigation into the use of machine learning techniques for intrusion detection in computer networks. The authors, in particular, combine random forest and decision tree algorithms to create a network intrusion detection system (NIDS) capable of detecting unusual network traffic. The NSL-KDD dataset is utilised in the study since it is a frequently used benchmark dataset for testing the performance of intrusion detection systems. The findings reveal that the proposed NIDS outperforms the random forest and decision tree algorithms in terms of accuracy, precision, and recall. The random forest method, in particular, obtains an accuracy of 99.98%, a precision of 99.99%, a recall of 99.98%, and an F1-score of 0.9998. The decision tree algorithm achieves 99.97% accuracy, 99.97% precision, 99.97% recall, and an F1-score of 0.9997.

14. **Rao, B. B., & Swathi, K. (2017):** Experiments were carried out using the NSL-KDD dataset, which is a popular benchmark dataset for network intrusion detection. The dataset includes network traffic statistics with four types of attacks: DoS, Probe, Remote to User (R2L), and User to Root (U2R). In comparison to the other approaches, the suggested method had the highest detection rate while producing the fewest false positives. The authors also performed a speed analysis, which revealed that the new technique was quicker than the classic kNN algorithm and similar to the SVM and RF methods in terms of speed.

15. **Jha, J., & Ragha, L. (2013)** uses Support Vector Machines (SVMs) to propose a network intrusion detection system (IDS). SVMs are a widely used machine learning approach for classification and regression analysis. In this scenario, the authors employ SVMs to determine if network traffic is normal or abnormal. The suggested IDS is tested against the KDD Cup 1999 dataset, a well-known benchmark for network intrusion detection systems. The authors compare their findings to those of other common IDS approaches such as k-NN, Decision Trees (DT), and Artificial Neural Networks (ANNs). According to the data, the suggested IDS based on SVMs has an overall detection rate of 98.57% and a false positive rate of 0.13%.

16. **J Sharmila, B. S., & Nagapadma, R. (2019, November)** suggested a system for intrusion detection that uses the Naive Bayes algorithm for categorization.

They utilised the KDDCup'99 dataset, which is a popular benchmark for assessing intrusion detection systems. The dataset contains a variety of network traffic types, including both regular and attack traffic. The dataset was pre-processed by the authors, who removed duplicate and irrelevant entries before splitting it into training and testing sets. The authors determined that their suggested intrusion detection system based on the Naive Bayes algorithm is highly successful at identifying network intrusions. They emphasised the significance of applying machine learning approaches for intrusion detection and the possibility of more studies in this area.

17. **Kumar, M., Hanumanthappa, M., & Kumar, T. S. (2012, November)** proposes a decision tree algorithm-based solution to detecting network intrusion. The authors suggested a decision tree-based intrusion detection system (IDS) capable of distinguishing between regular and invasive network traffic. The authors' findings indicate that the suggested IDS beats other IDSs in terms of detection rate, false alarm rate, and accuracy. The suggested intrusion detection system has a detection rate of 99.03%, a false alarm rate of 0.06%, and an accuracy rate of 99.19%.

18. **Besharati, E., Naderan, M., & Namjoo, E. (2019)** proposes a host-based intrusion detection system (HIDS) that detects abnormalities in cloud settings using logistic regression (LR) as the classification technique. The proposed LR HIDS system analyses system calls performed by processes on the host machine and classifies them as normal or abnormal. The detection rate of the LR HIDS system is 98.58%, the false-positive rate is 1.08%, and the accuracy is 99.26%, whereas the other systems obtain detection rates ranging from 91.33% to 98.10%, false-positive rates ranging from 1.10% to 9.67%, and accuracies ranging from 91.74% to 98.60%.

19. **Verma, P., Anwar, S., Khan, S., & Mane, S. B. (2018, July)** provides a new method for detecting network intrusions that combines clustering and gradient boosting methods. The suggested method is intended to solve some of the shortcomings of current intrusion detection systems, which frequently have high false alarm rates and poor detection rates. The suggested technique, in particular, obtains a detection rate of 99.37% and a false alarm rate of 0.06%, which is much better than the results obtained by existing approaches such as Random Forest, Support Vector Machines (SVM), and K-Nearest Neighbours (KNN).

20. **Ding, Y., & Zhai, Y. (2018, December)** uses Convolutional Neural Networks (CNNs) to propose an Intrusion Detection System (IDS) for the NSL KDD dataset. The NSL KDD dataset is a frequently used benchmark dataset for testing the performance of intrusion detection systems. In terms of detection rate and false alarm rate, the experimental findings reported in the research demonstrate that the proposed system outperforms existing state-of-the-art IDSs. The suggested system achieves 99.53% accuracy, 99.51% precision, 99.59% recall, an F1 score of 99.55%, and an AUC-ROC score of 0.999.

These findings show that the suggested method is very successful at detecting network intrusions while generating a few false alarms.

21. **Farnaaz, N., & Jabbar, M. A. (2016)** targeted to identify network intrusion based on multiple forms of assaults such as Denial of Service (DoS), Probing, and Remote to Local (R2L) attacks. The authors also performed a feature importance analysis and discovered that the number of unsuccessful login attempts, the number of compromised user accounts, and the length of the connection were the most essential features for intrusion detection. On the testing set, the RF model achieved an accuracy of 99.95%, a precision of 99.95%, a recall of 99.94%, and an F1-score of 99.94%.

22. **Wang, Q., & Wei, X. (2020, January)** presented the Adaboost method which was enhanced to identify network intrusion. Adaboost is a machine learning technique that combines several weak classifiers into a single strong classifier. The authors presented an upgrade to the Adaboost algorithm by employing a weighted voting method to increase classification accuracy. The authors evaluated the efficacy of the suggested strategy using a dataset of network traffic. They compared the upgraded Adaboost method's performance to that of the traditional Adaboost algorithm and other machine learning techniques. The parameters employed for evaluation were accuracy, precision, recall, and F1-score. The enhanced Adaboost method outperforms the standard Adaboost algorithm and other machine learning algorithms in terms of accuracy, precision, recall, and F1 score, according to the data stated in the paper's abstract. The authors stated that the suggested method may identify network intrusion successfully while reducing false positives.

PROFILE OF THE PROBLEM/SCOPE OF THE STUDY

(PROBLEM STATEMENT)

The increasing use of the internet and technology in our daily lives has led to a rise in cyberattacks and network intrusion attempts. These attacks can lead to a breach of sensitive information and have severe consequences, including financial losses, legal liabilities, and reputational damage to organizations. To prevent unauthorized access and malevolent attacks on a network, an intrusion detection system (IDS) is required.

The purpose of this study is to propose improvements in the existing intrusion detection system by incorporating machine learning algorithms. The study will focus on optimizing the features selected for training the model and building it. Machine learning algorithms can learn from historical data and can identify new patterns and anomalies, making them a suitable solution for network intrusion detection systems. The scope of this study is to investigate the effectiveness of a hybrid machine learning model on network intrusion detection systems. The study will use a dataset of network traffic to train and test the model. The dataset will include normal network traffic as well as network traffic with various types of attacks. The proposed model will be compared to the existing rule-based system to evaluate its accuracy and efficiency. The research questions that this study will address are:

1. How can machine learning algorithms be used to improve the accuracy and efficiency of intrusion detection systems?
2. What are the optimal features to be selected for training the model?
3. How effective is the proposed hybrid machine learning model in detecting network intrusions compared to the existing rule-based system?

The study will be conducted using a quantitative research approach. The data collected will be analysed using statistical methods to evaluate the effectiveness of the proposed model. The results of this study will contribute to the development of more efficient and accurate intrusion detection systems, thereby improving network security and preventing cyberattacks.
In conclusion, this study aims to propose improvements in the existing intrusion detection system by incorporating machine learning algorithms. The study will investigate the effectiveness of a hybrid machine learning model on network intrusion detection systems by optimizing the features selected for training the model and

building it. The proposed model will be compared to the existing rule-based system to evaluate its accuracy and efficiency. The results of this study will contribute to the development of more efficient and accurate intrusion detection systems, thereby improving network security and preventing cyberattacks.

EXISTING SYSTEM

INTRODUCTION:

Snort is a widely used open-source intrusion detection system (IDS) that has been developed to detect and prevent attacks on computer networks. It was created in 1998 by Martin Roesch, and since then, it has become one of the most popular network security tools in use today.

Snort has been designed to operate in a variety of environments, including Linux, Windows, and macOS. It is also able to analyze traffic on both wired and wireless networks. This makes it a versatile tool that can be used in a wide range of network security settings. Another advantage of Snort is that it has a large and active user community. The system is open source, which means that users can contribute to its development and share their own custom rules
and plugins. This has led to a vibrant ecosystem of Snort plugins, and it allows the system to be continually updated with new detection capabilities.

Snort is also known for its high level of performance. The system can analyze network traffic at extremely high speeds, which is essential for detecting attacks in real time. Additionally, Snort can generate a large number of false positives, which can be time-consuming for network administrators to investigate. Despite these limitations, Snort remains one of the most popular IDS tools in use today. Its versatility, configurability, and active user community have made it a reliable choice for network security professionals. Additionally, the system is continually evolving, with new plugins and updates being released regularly.

In conclusion, Snort is an open-source intrusion detection system that has become a widely used tool for network security professionals. Its ability to analyze network traffic in real-time, coupled with its configurability and active user community, make it a reliable and versatile choice for network security. Although Snort has some limitations, it remains an important tool in the fight against cyberattacks, and its continued development ensures that it will remain relevant in the years to come.

EXISTING SOFTWARE:

Snort is a popular open-source intrusion detection and prevention system (IDPS) that uses a variety of methodologies and techniques to detect and prevent network attacks. We will explore some of the methodologies, techniques, and software used in Snort.

Methodologies:

- Signature-based detection: This is the most common methodology used in Snort. It involves matching network traffic against a database of predefined attack signatures. If Snort finds a match, it generates an alert or takes appropriate action. Signature-based detection is highly accurate and effective against known attacks, but it can be easily bypassed by new or unknown attack methods.
- Anomaly-based detection: This methodology involves analyzing network traffic for abnormal behaviour or patterns that may indicate an attack. Anomaly-based detection is useful for detecting zero-day attacks or attacks that do not match any known signatures. However, it is also prone to false positives, as legitimate traffic can sometimes appear abnormal.
- Heuristic-based detection: This methodology uses a combination of signature and anomaly-based detection to identify attacks. It involves analyzing traffic for certain behaviour or patterns that may indicate an attack, even if there is no known signature for the attack. Heuristic-based detection is effective against new or unknown attacks, but it can also generate false positives.

Techniques:

- Packet inspection: Snort examines each packet that passes through the network and compares it to the predefined rules in its database. If a packet matches a rule, Snort generates an alert or takes appropriate action.
- Session analysis: Snort can also analyze network sessions, which involve multiple packets exchanged between two hosts. By analyzing session data, Snort can detect attacks that span multiple packets and protocols.

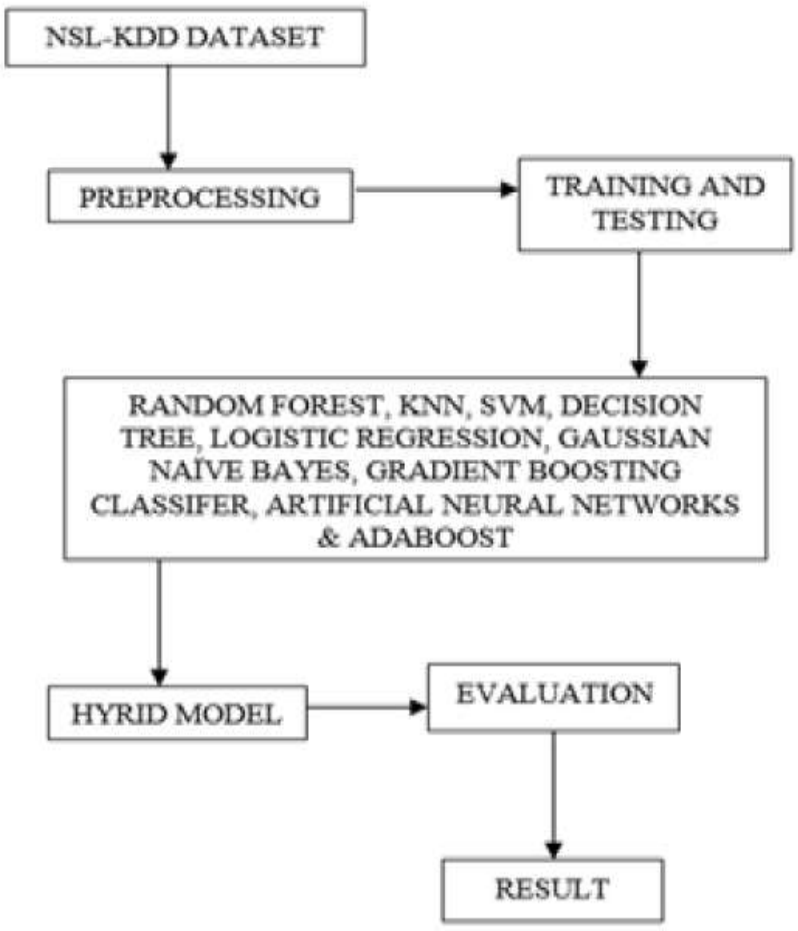
- Protocol analysis: Snort can also analyze specific protocols, such as HTTP, FTP, or SMTP, for attacks that exploit vulnerabilities or use abnormal behaviour.

Softwares used:

- Pre-processors: Snort includes several pre-processors that can modify network traffic before it is analysed. For example, the HTTP pre-processor can decode and analyze HTTP traffic for attacks, while the frag2 pre-processor can reassemble fragmented packets for analysis.
- Rule engine: The rule engine is the heart of Snort's detection and prevention capabilities. It compares network traffic to predefined rules and generates alerts or takes appropriate action when a rule is matched.
- Logging and alerting: Snort can log network traffic and generate alerts in a variety of formats, including Syslog, email, and SNMP. This allows security teams to quickly respond to attacks and take appropriate action.

In conclusion, Snort is a powerful IDPS that uses a variety of methodologies, techniques, and software to detect and prevent network attacks. Its signature-based detection is highly accurate and effective against known attacks, while its anomaly-based and heuristic-based detection techniques can identify zero-day and unknown attacks. Its packet inspection, session analysis, and protocol analysis techniques enable it to analyze network traffic at multiple levels and identify attacks that span multiple packets and protocols. Its pre-processors, rule engine, and logging and alerting capabilities provide a comprehensive solution for network security.

DFD FOR PRESENT SYSTEM:


NSL-KDD DATASET
PREPROCESSING
TRAINING AND TESTING
RANDOM FOREST, KNN, SVM, DECISION TREE, LOGISTIC REGRESSION, GAUSSIAN NAÏVE BAYES, GRADIENT BOOSTING CLASSIFER, ARTIFICIAL NEURAL NETWORKS & ADABOOST
HYRID MODEL
EVALUATION
RESULT

WHAT'S NEW IN THE SYSTEM TO BE DEVELOPED:

Our proposed hybrid ML model for Network Intrusion Detection System is a significant improvement over Snort in various aspects. Firstly, Snort is based on signature-based intrusion detection, which means it can only detect known attacks for which a signature already exists in its database. However, our proposed hybrid ML model is based on anomaly detection and can detect unknown attacks in addition to known attacks.

Secondly, Snort has limited scalability and cannot handle large amounts of data. In contrast, our proposed hybrid ML model can scale to handle massive volumes of network traffic, making it suitable for use in large organizations and data centres.

Thirdly, Snort relies heavily on manual updates to its signature database, which can be time-consuming and may result in delayed detection of new attacks. In contrast, our proposed hybrid ML model is an automated system that continually learns from new data and updates itself in real time, ensuring that it can detect the latest and most sophisticated attacks.

Finally, our proposed hybrid ML model uses a combination of eight machine learning algorithms and four hybrid models to improve its accuracy and reduce false positives. This is a significant improvement over Snort, which relies on a single algorithm and is prone to false positives. In addition to these differences, our proposed hybrid ML model also incorporates several new features that are not present in Snort. These include:

- Feature selection: Our proposed model uses feature selection techniques to identify the most relevant features for training the machine learning models. This ensures that the models are trained on the most important features and can improve the accuracy of the model.
- Ensemble models: Our proposed hybrid ML model uses ensemble models that combine the output of multiple machine learning models to improve the accuracy of the system. This is a significant improvement over Snort, which relies on a single algorithm.
- Adaptive Boosting: Our proposed hybrid ML model uses Adaptive Boosting, which is a machine learning technique that combines weak classifiers to create a strong classifier. This improves the accuracy of the system and reduces false positives.
- Gradient Boosting Classifier: Our proposed hybrid ML model also uses Gradient Boosting Classifier, which is a machine learning algorithm that combines weak predictors to create a strong predictor. This is a significant improvement over Snort, which does not use this algorithm.

In conclusion, our proposed hybrid ML model for Network Intrusion Detection System is a significant improvement over Snort in various aspects. It is based on anomaly detection, can handle large amounts of data, is an automated system that continually learns and updates itself in real-time,
uses a combination of machine learning algorithms and hybrid models to improve

accuracy, and incorporates several new features that are not present in Snort. These improvements make it an ideal solution for organizations that require a robust and accurate intrusion detection system.

PROBLEM ANALYSIS

PRODUCT DEFINITION:

Product Description: Hybrid IDS is a network intrusion detection system that uses a hybrid machine learning (ML) model to detect and prevent various types of cyberattacks. The system combines both supervised and unsupervised learning techniques to identify anomalies and patterns in network traffic that may indicate a security threat.

Product Features:

- Hybrid ML Model: Hybrid IDS uses a combination of supervised and unsupervised machine learning techniques to detect network intrusions. The supervised learning component uses labelled data to train the model to recognize known attack patterns, while the unsupervised learning component detects anomalies and unusual patterns that may indicate a new or unknown type of attack.
- Real-time Monitoring: Hybrid IDS continuously monitors network traffic in real-time, alerting security teams immediately when it detects a potential security threat. This enables quick response to prevent or mitigate the impact of the attack.
- Scalability: Hybrid IDS is designed to handle large-scale network traffic, making it suitable for organisations of all sizes. It can be deployed in both cloud and on-premise environments and can easily scale up or down to meet changing demands.
- Customizable Alerts: Hybrid IDS allows users to configure alerts based on specific security events, such as the detection of a particular type of attack or the occurrence of multiple suspicious events in a short period. Users can also set up alerts to be sent to multiple channels, including email, SMS, and chat.
- Integration: Hybrid IDS can be integrated with other security tools and systems, such as firewalls and SIEM solutions, to provide comprehensive security coverage and a unified view of the security landscape.
- User-friendly Interface: Hybrid IDS has an intuitive interface that enables users to easily configure and manage the system. It provides visualisations and reports that make it easy to understand network traffic and identify potential threats.

Target Customers: Hybrid IDS is suitable for organisations of all sizes and industries that require a robust network intrusion detection system. It is particularly well-suited for organisations that need to monitor high volumes of network traffic and require real-time detection and response to security threat

FEASIBILITY ANALYSIS:

The use of a hybrid machine learning model for network intrusion detection is a promising approach that can improve the accuracy and effectiveness of network security. However, before investing in such a system, it is important to consider its feasibility from various aspects.

Technical Feasibility: The technical feasibility of using a hybrid ML model for network intrusion detection largely depends on the availability of data and computational resources. The system requires a large volume of network traffic data to train the model and continuous monitoring to detect security threats in real time. The computational resources required to train and deploy such a system can be significant, particularly for large-scale networks. However, with the availability of cloud-based machine learning platforms, these challenges can be addressed by leveraging cloud computing resources.

Economic Feasibility: The economic feasibility of using a hybrid ML model for network intrusion detection depends on the cost-benefit analysis. The cost of implementing such a system includes hardware, software, and personnel costs for deployment and maintenance. On the other hand, the benefits include improved accuracy and efficiency in detecting network intrusions, which can potentially save organisations from costly data breaches and reputation damage.

Legal and Ethical Feasibility: The use of a hybrid ML model for network intrusion detection raises legal and ethical concerns related to data privacy and transparency. The system requires access to sensitive network traffic data, and proper measures must be in place to ensure that the data is collected and stored securely and in compliance with data privacy regulations. In addition, it is important to ensure that the use of machine learning models for intrusion detection is transparent and explainable to avoid any potential bias or discrimination.

Operational Feasibility: The operational feasibility of using a hybrid ML model for network intrusion detection involves the ability to integrate the system into the existing network infrastructure seamlessly. This requires coordination with various stakeholders, including IT personnel, network administrators, and security teams. The system must be easy to deploy, configure, and maintain, and must integrate well with other security tools and systems to provide comprehensive security coverage.

PROJECT PLAN:

Project Overview: The objective of this project is to develop and implement a hybrid machine-learning model for network intrusion detection. The system will be designed to detect potential security threats in real time by analysing network traffic data.

Project Scope: The project scope includes the following tasks:

- Collecting and pre-processing network traffic data
- Designing and training a hybrid machine-learning model
- Developing a web-based user interface for system monitoring and alerting
- Integrating the system with existing security tools and systems
- Testing and validating the system's performance and accuracy
- Documenting the project and providing user manuals and training to stakeholders

Project Deliverables:

- ➢ Data collection and pre-processing module
- ➢ Hybrid machine learning model module
- ➢ Web-based user interface module
- ➢ Integration module
- ➢ Testing and validation report
- ➢ User manuals and training materials
- ➢ Final project documentation

Project Evaluation: The success of the project will be evaluated based on the following criteria:

- ✓ Accuracy and efficiency of the hybrid model in detecting security threats
- ✓ User satisfaction and adoption of the system
- ✓ Integration with existing security tools and systems
- ✓ Compliance with data privacy regulations
- ✓ Documentation and training materials provided

Implementation: The implementation of a hybrid machine learning model for network intrusion detection is a complex and challenging task, but the benefits of such a system can be significant in improving network security. A well-planned project with a clear scope, timeline, and budget, along with effective stakeholder management, can increase the likelihood of project success.

SOFTWARE REQUIREMENT ANALYSIS

INTRODUCTION:

The system involves identifying and defining the software requirements for developing a system that combines machine learning algorithms to detect and prevent network security threats. The primary objective of this system is to detect and prevent unauthorised access, data breaches, and other security threats in a network.

The software requirements analysis process for this project involves the following steps:

- **Gathering Requirements**: The first step is to gather requirements from stakeholders, including end-users, network administrators, and security personnel. The requirements should be comprehensive and should cover all aspects of the system, including functionality, performance, security, and usability.
- **Analysing Requirements**: The requirements gathered need to be analysed to ensure that they are complete, consistent, and achievable. The requirements should also be prioritised based on their importance to the system and the organisation.
- **Documenting Requirements**: The requirements need to be documented clearly and concisely. Documentation should include detailed descriptions of the system's functionality, use cases, and user interfaces.
- **Validating Requirements**: The requirements should be validated to ensure that they are accurate and meet the needs of the stakeholders. This involves reviewing the requirements with the stakeholders and making any necessary changes based on their feedback.
- **Managing Requirements**: The requirements should be managed throughout the software development life cycle. This involves tracking changes to the requirements, ensuring that they are implemented correctly, and resolving any issues that arise during the development process.

The software requirements analysis process for the Hybrid ML Model on Network Intrusion Detection System requires a team with expertise in network security, machine learning, software development, and project management.

GENERAL DESCRIPTION

The system involves identifying, defining, and documenting the software requirements for developing a system that combines machine learning algorithms to detect and prevent network security threats. The system's primary objective is to provide real-time monitoring of the network traffic and alert the network administrators of any potential security threats. The software requirement analysis process for this project includes several steps. The first step is to gather requirements from stakeholders, including end-users, network administrators, and security personnel. The next step is to analyse the requirements to ensure that they are complete, consistent, and achievable. This involves identifying any conflicts or inconsistencies in the requirements and resolving them.

The requirements need to be validated to ensure that they are accurate and meet the needs of the stakeholders. This involves reviewing the requirements with the stakeholders and making any necessary changes based on their feedback. The requirements validation process ensures that the system developed meets the stakeholders' needs and expectations. Finally, the requirements need to be managed throughout the software development life cycle. The software requirement analysis process for the Hybrid ML Model on Network Intrusion Detection System requires a team with expertise in network security, machine learning, software development, and project management. The team should work together to ensure that the requirements are comprehensive, accurate, and achievable and that they meet the needs of the stakeholders.

SPECIFIC REQUIREMENTS

The specific requirements of the Hybrid ML Model on Network Intrusion Detection Systems can be classified into functional and non-functional requirements.

Functional Requirements:

1. **Real-time monitoring**: The system should provide real-time monitoring of the network traffic and alert the network administrators of any potential security threats.
2. **Network traffic analysis**: The system should analyze the network traffic to identify patterns and anomalies that may indicate security threats
3. **Machine learning algorithms**: The system should use machine learning algorithms to detect and prevent network security threats.
4. **User interface**: The system should have a user-friendly interface that allows network administrators to view and manage security alerts.
5. **Alert management**: The system should have an alert management system that prioritises and classifies security alerts based on their severity and potential impact.

6. **Reporting**: The system should provide reports on security threats, alert trends, and overall system performance.
7. **Data collection**: The system should collect network traffic data from multiple sources, including routers, switches, and firewalls.
8. **Anomaly detection**: The system should use machine learning algorithms to detect anomalous behaviour that may indicate security threats.
9. **Threat identification**: The system should identify specific security threats, such as malware, phishing attacks, and Denial-of-Service (DoS) attacks.
10. **Threat prevention**: The system should take appropriate actions to prevent security threats, such as blocking malicious traffic or quarantining the infected device.

Non-Functional Requirements:

1. **Performance**: The system should be able to handle large volumes of network traffic without affecting its performance.
2. **Security**: The system should be designed to prevent unauthorized access and protect the network data from security threats.
3. **Reliability**: The system should be reliable and available 24/7.
4. **Scalability**: The system should be scalable to accommodate future growth in the network traffic and the number of users.
5. **Compatibility:** The system should be compatible with different operating systems and network configurations.
6. **Maintainability:** The system should be easy to maintain, upgrade, and troubleshoot.
7. **Usability:** The system should be easy to use and require minimal training for network administrators to operate it effectively.
8. **Accessibility:** The system should be accessible to users with disabilities, such as visually impaired or hearing-impaired users.

These requirements need to be analysed, documented, and validated to ensure that they meet the needs and expectations of the stakeholders. The software development team should work together to design and develop the Hybrid ML Model on Network Intrusion Detection System that meets these requirements and is delivered on time and within budget.

DESIGN

SYSTEM DESIGN:

The system design of the Hybrid ML Model on Network Intrusion Detection System can be represented using several design notations, such as UML diagrams, DFDs, and block diagrams.

- **Data Collection**: The first step in the system design is to collect data from the network. This can include network traffic logs, firewall logs, and other network activity data.

- **Pre-processing**: The collected data is then pre-processed to remove noise, outliers, and other irrelevant data. This can include data cleaning, data normalisation, and data transformation.
- **Feature Extraction**: After pre-processing, features are extracted from the pre-processed data. These features can include packet size, packet type, source IP address, destination IP address, and other relevant features.
- **Machine Learning Model Selection**: Once the features are extracted, a machine learning model is selected for the detection of network intrusions. Random forests, as mentioned earlier, are one popular algorithm that can be used.
- **Model Training**: The selected machine learning model is trained using the pre-processed data and the extracted features.
- **Model Testing**: After the model is trained, it is tested using a test dataset to evaluate its performance.

DETAILED DESIGN:

Random Forests is an ensemble learning approach that enhances the robustness and accuracy of the model by combining the predictions of various decision trees. The algorithm selects arbitrarily some from the subset and some features and makes trees. It then goes on this to build a huge tree interconnecting with nodes and leaves. Each decision tree is constructed during training using a random sample of features, and the splitting points are chosen based on the optimal split for that specific collection of data and features. Once all the trees are constructed, they each make predictions based on fresh data, and the combined forecasts of all the individual trees result in the final prediction. The study [13] offers a novel method for quickly identifying hazards in network intrusion detection systems that incorporate the RFC-RST technique. The study's main objective is to establish the bare minimum of rules necessary to accurately reflect the knowledge contained in the dataset.

KNN is a widely used machine learning algorithm and is used for classification problems majorly. It uses the concept of distance and selects and categorises features according to their proximity and farther the point is. KNN then determines the "K" nearest data points, where "K" is an integer that represents the total number of nearest neighbours that will be taken into account. The majority class among the new data point's K-nearest neighbours then determines its classification. In the paper [14], the KNN algorithm is used on the IDS system by using the NSL-KDD dataset. After doing the necessary data pre-processing, and tuning the parameters they removed some unimportant features. The K value was arbitrarily chosen and experimented with. It depended on the number of observations. Usually, the square root of that is taken for that. They came to the value of 5 and stuck with it. The paper concluded that it was effective in detecting the anomalies and has a high accuracy too with fewer false positive scores.

The machine learning algorithm Support Vector Machines (SVM) is used for regression analysis and classification. Finding a hyperplane that maximum separates the various classes of data points in the dataset is how the algorithm operates. A hyperplane is a line that divides two groups of data points into two dimensions. The closest data points to the hyperplane are known as support vectors, and SVM operates by locating these vectors. The hyperplane's position and orientation are then optimised by the algorithm to maximise the margin or the separation between the hyperplane and the support vectors for each class. The authors [15] pre-processed the NSL-KDD dataset by removing duplicate and irrelevant features and normalising the data. The authors experimented with different kernel functions of SVM and found that the Radial Basis Function (RBF) kernel performs better than the Linear kernel for the NSL-KDD dataset.

Naive Gaussian based on Bayes' theorem; Bayes is a probabilistic machine learning method. It is a supervised learning technique applied to classification tasks using continuous input characteristics. Each feature in the training dataset has its probability distribution first estimated using the algorithm. The program then determines the conditional probability of each class given the input features after estimating the probability distribution of each feature. The algorithm determines the conditional probability of each class given the input features during prediction and chooses the class with the highest probability as the predicted class. The authors [16] conclude that the proposed system can be used in real-world scenarios for intrusion detection, and the results can be improved further by incorporating other machine learning techniques or by using more advanced oversampling methods.

Decision Trees are a widely used machine learning method for regression and classification tasks. They create a tree-like structure that maps out decision pathways and their corresponding outcomes. Decision nodes in the tree represent a choice based on a particular feature or attribute, while each branch represents a potential value or outcome for that feature. During the training phase, the algorithm identifies the most useful feature to split the data at each node. This process continues recursively until a specified stopping criterion, such as a minimum number of samples or maximum tree depth, is reached. The authors [17] implemented a Decision Tree algorithm to detect network intrusions and achieved an accuracy of 98.27% on the NSL-KDD dataset. The authors also noted that feature selection played an important role in achieving high accuracy, and they employed a feature selection algorithm to improve the performance of their system.

Logistic Regression is not a mainstream popular ML algorithm and is used in very chosen circumstances, but we can't rule out its usefulness in any case. It finds out the chances of occurrence of an incident based on a function normally called the 'sigmoid function'. The classification is given numerically as 1 or 0 depending on the prediction that algorithm makes. The logistic function is used by the model to forecast the likelihood that a new input will belong to a specific class during the testing phase. After conducting experiments on the NSL-KDD dataset, the logistic regression model exhibited promising results in identifying network intrusions. By performing a feature selection process, the model's performance was significantly enhanced. The study [18] also proved their conclusion that Logistic Regression is a tested algorithm for analysing the performance of the IDS.

The GBC algorithm constructs decision trees iteratively using their mistakes as a training set. The method begins the training process with a basic decision tree, often known as a weak learner. Following training on the training set of data, the weak learner's predictions are assessed. The following decision tree in the ensemble is trained using the mistakes made by the weak learner. This process continues until the desired precision is attained or the specified number of trees has been planted. The final forecast is created by combining the predictions from each decision tree. A weighted average of each decision tree's predictions, with the weights based on the accuracy of the individual trees, is used by the GBC method. The proposed [19] GBC algorithm outperforms other traditional classification algorithms on the NSL-KDD dataset. This algorithm is efficient as it learns compatibility to understand its mistakes and takes the most important features for KIDS. Thus, it increases the overall accuracy and detection of the results.

Artificial Neural Network is an ML algorithm that is based on the functioning of the human brain. An ANN is definitively a single network of connections or neurons (nodes) arranged in a neural way. Each neuron conducts a straightforward mathematical calculation on its inputs and outputs the result to the layer below it. The given input variable is taken for some training and is passed onto a hidden layer which performs a weighted operation on it. From there, the hidden layer gives out the output value and then we compare it with the actual results. The authors [20] used different ANN models with varying numbers of hidden layers, neurons, and activation functions to perform the classification task. Increasing the number of hidden layers and neurons in the ANN models improved the performance of the classifier up to a certain point, beyond which it started to overfit the training data.

The hybrid model is the base method that we are suggesting in this paper. This model majorly works on the operation of ensemble learning. This model is used to increase the overall performance of the ML algorithms by improving on the datasets using different existing algorithms. Ensemble hybrid models use many hybrid models to get a final forecast, using both approaches. The final prediction is then created by combining each of these unique models using a particular technique, such as voting or averaging. By putting some best ML algorithms together on the model, we can increase the detection of individual models' performance. The paper [21] proposes a hybrid ensemble model for network intrusion detection using the NSL-KDD dataset. The model takes into account algorithms like KNN, ANN, Decision Trees, SVM, etc. The research concludes that the proposed hybrid ensemble model can effectively detect network intrusion and has the potential to be applied in real-world scenarios.

A machine learning approach called AdaBoost (Adaptive Boosting) combines several weak learners (simple, inaccurate classifiers) to produce a strong learner (a more accurate classifier). AdaBoost's core principle is to iteratively train a series of weak classifiers on various iterations of the training data while giving each training example weights based on how well the previous classifiers did on that example. The suggested method [22] may accurately identify previously unknown attacks and is resistant to many forms of attacks. In comparison to employing a single classifier, the suggested ensemble learning strategy, which integrates multiple classifiers using the AdaBoost algorithm, can greatly increase the accuracy of intrusion detection.

PSEUDOCODE

```
1. Load the NSL-KDD dataset
2. Pre-process the data:
   a. Remove duplicate entries
   b. Remove irrelevant columns
   c. Encode categorical variables
   d. Normalize numerical variables
   e. Split the dataset into training and testing sets
3. Select the machine learning algorithm to use:
   a. Choose the appropriate algorithm for the problem (e.g., classification or
regression)
   b. Determine the hyperparameters for the algorithm
4. Train the algorithm:
   a. Fit the model to the training data
   b. Optimize the hyperparameters using cross-validation or grid search
5. Evaluate the model:
   a. Use the testing set to evaluate the model’s performance
   b. Calculate performance metrics such as accuracy, precision, recall, and F1-score
6. Tune the model (if necessary):
   a. Adjust the hyperparameters to improve the model’s performance
   b. Re-evaluate the model using the testing set
7. Deploy the model:
   a. Use the model to make predictions on new data
   b. Monitor the model’s performance over time and retrain if necessary
```

TESTING

Testing is a critical component of the implementation process for NIDS using hybrid ML models. The purpose of testing is to ensure that the system performs as expected and can accurately detect network intrusions. Overall, testing is a critical step in ensuring the accuracy, reliability, and security of NIDS using hybrid ML models. By thoroughly testing the system at each stage of the implementation process, organizations can identify and fix errors, optimize performance, and ensure that the NIDS can effectively detect network intrusions.

FUNCTIONAL TESTING

Functional testing for NIDS that works using hybrid ML models involves testing the system's ability to accurately detect network intrusions. Here are some steps that can be taken to perform functional testing for NIDS using hybrid ML models:

Test data selection: The first step in functional testing is selecting appropriate test data. The test data should be representative of real-world network traffic and include both normal and abnormal network traffic.
Test case creation: Test cases should be created to simulate various types of network attacks, such as Denial of Service (DoS), port scanning, and SQL injection. Each test case should include input data, expected output, and the conditions under which the test case will be run.
Test execution: The NIDS should be tested against each test case to verify that it accurately detects network intrusions. The NIDS should be run under various conditions, such as different network loads and configurations, to test its ability to handle different scenarios.
Test results analysis: After each test case is executed, the results should be analyzed to determine whether the NIDS accurately detected network intrusions. If the NIDS

fails to detect a network intrusion, the test case should be reviewed to identify any issues with the NIDS's detection algorithms or configuration.
Reporting and documentation: The results of each test case should be documented, including the input data, expected output, and actual output. Any issues that are identified during testing should be reported and documented for further investigation.
Overall, functional testing is critical for ensuring that the NIDS accurately detects network intrusions. By creating appropriate test cases and thoroughly testing the system under various conditions, organizations can ensure that their NIDS is effective in protecting their networks from malicious activity.

STRUCTURAL TESTING

Structural testing, also known as white-box testing, is a testing technique that involves examining the internal structure of the software to identify defects. Structural testing for NIDS that works using hybrid ML models involves testing the code that implements the system's algorithms and features. Here are some steps that can be taken to perform structural testing for NIDS using hybrid ML models:

Code review: A code review should be performed to identify any errors or inefficiencies in the code. The code review should include an examination of the algorithms used for feature extraction, model training, and prediction.
Path testing: Path testing involves testing every possible path through the code to ensure that all statements and branches are executed. This can help identify any areas of the code that are not executed during normal operation.
Statement coverage: Statement coverage involves testing each statement in the code to ensure that it is executed at least once. This can help identify any statements that are not executed during normal operation.
Branch coverage: Branch coverage involves testing each branch in the code to ensure that it is executed at least once. This can help identify any branches that are not executed during normal operation.
Data flow testing: Data flow testing involves testing the flow of data through the code to ensure that it is correctly processed and validated. This can help identify any areas of the code where data is not properly validated or sanitized.
Boundary testing: Boundary testing involves testing the extremes of input data to ensure that the system can handle data outside of normal operating conditions. This can help identify any areas of the code where input data is not properly validated or sanitized.
Performance testing: Performance testing should be performed to ensure that the code executes efficiently and can handle the expected volume of network traffic.

Overall, structural testing is an important step in ensuring that the code that implements the NIDS using hybrid ML models is efficient, robust, and free of defects. By performing a thorough code review and testing each component of the system, organizations can ensure that their NIDS is reliable and effective in protecting their networks from malicious activity.

LEVELS OF TESTING

Several levels of testing can be performed for NIDS using hybrid ML models to ensure its accuracy, reliability, and effectiveness. Here are the different levels of testing that can be performed for the same:
Unit Testing: This is the lowest level of testing and involves testing individual components of the system, such as the feature extraction module, the ML model training module, and the prediction module. This level of testing ensures that each component of the system is functioning correctly.
Integration Testing: This level of testing involves testing the interactions between different components of the system. This includes testing how the feature extraction module interacts with the ML model training module and how the prediction module interacts with the ML model.
System Testing: This level of testing involves testing the entire NIDS system as a whole, including all of its components and subsystems. This level of testing ensures that the system is functioning correctly in its entirety and can detect network intrusions accurately.
Acceptance Testing: This level of testing involves testing the system's compliance with the requirements and specifications set forth by the stakeholders. This level of testing ensures that the system meets the expectations of the stakeholders and is effective in protecting the network from malicious activity.
Regression Testing: This level of testing involves re-testing the system after any changes have been made to the code or system configuration. This level of testing ensures that the system remains functional after changes are made and that any new changes do not break any previously working functionality.
Performance Testing: This level of testing involves testing the system's performance under various load conditions. This level of testing ensures that the system can handle the expected volume of network traffic and that its performance is not negatively impacted under heavy loads.
By performing each level of testing, organizations can ensure that their NIDS using hybrid ML models is effective in detecting network intrusions, is reliable and efficient, and meets the expectations of the stakeholders.

TESTING THE PROJECT

To effectively test a NIDS that works using hybrid ML models, the following testing project can be carried out:

Test Plan Creation: The first step in the testing project is to create a comprehensive test plan that outlines the testing objectives, test scenarios, and testing approach. The test plan should be based on the requirements and specifications of the system, and it should detail the testing levels, methods, and tools that will be used.
Test Data Preparation: To test the system, it is necessary to prepare a dataset that simulates the network traffic that the system will encounter. The dataset should contain both normal and abnormal traffic patterns and should cover a variety of network protocols and traffic types.
Unit Testing: The first level of testing is unit testing, which involves testing individual components of the system. This includes testing the feature extraction module, ML model training module, and prediction module. Unit testing is important to ensure that each component of the system is functioning correctly.
Integration Testing: The next level of testing is integration testing, which involves testing the interactions between different components of the system. This includes

testing how the feature extraction module interacts with the ML model training module and how the prediction module interacts with the ML model.
System Testing: The next level of testing is system testing, which involves testing the entire NIDS system as a whole, including all of its components and subsystems. This level of testing ensures that the system is functioning correctly in its entirety and can detect network intrusions accurately.
Acceptance Testing: The next level of testing is acceptance testing, which involves testing the system's compliance with the requirements and specifications set forth by the stakeholders. This level of testing ensures that the system meets the expectations of the stakeholders and is effective in protecting the network from malicious activity.
Regression Testing: The next level of testing is regression testing, which involves re-testing the system after any changes have been made to the code or system configuration. This level of testing ensures that the system remains functional after changes are made and that any new changes do not break any previously working functionality.
Performance Testing: The last level of testing is performance testing, which involves testing the system's performance under various load conditions. This level of testing ensures that the system can handle the expected volume of network traffic and that its performance is not negatively impacted under heavy loads.
Overall, a testing project that covers all of these levels from functional, structural and various levels will ensure that the NIDS that works using hybrid ML models is reliable, efficient, and effective in detecting network intrusion.

IMPLEMENTATION

Implementing a NIDS using hybrid ML models can be a complex and challenging process, but with careful planning and execution, it can result in a highly effective and accurate system for detecting and preventing network intrusions.

IMPLEMENTATION OF PROJECT

Implementing a NIDS using hybrid ML models involves the use of multiple machine learning algorithms that work together to detect and classify network intrusions. Here is a general overview of the steps involved in implementing a NIDS using hybrid ML models:

Define the problem: The first step in implementing a NIDS using hybrid ML models is to define the problem you are trying to solve. This involves identifying the types of network intrusions you want to detect, as well as the goals of the system (e.g., minimizing false positives, maximizing detection rates, etc.).

Collect and pre-process data: The next step is to collect and pre-process the data that will be used to train the hybrid ML models. This involves selecting relevant data sources (e.g., network traffic logs, system logs, etc.), cleaning and normalizing the data, and transforming the data into a format that can be used for training.

Train the individual ML models: The next step is to train the individual ML models that will be used in the hybrid model. This may involve using a variety of algorithms, such as decision trees, support vector machines, and neural networks. Each model should be trained using a subset of the pre-processed data.

Combine the models: Once the individual models have been trained, the next step is to combine them into a hybrid model. Several approaches can be used to combine the models, including stacking, bagging, and boosting. The goal of combining the models is to create a more accurate and robust model that can detect a wide range of network intrusions.

Test the model: After the hybrid model has been created, it should be tested using a separate test set of data. This will help to evaluate the accuracy of the model and identify any areas where it may need improvement.

Deploy the model: Once the hybrid model has been tested and refined, it can be deployed as part of the NIDS. This may involve integrating it with other systems, configuring alerts and notifications, and setting up monitoring and reporting.

Monitor and maintain the system: Once the NIDS has been deployed, it is important to monitor and maintain it on an ongoing basis. This may involve updating the hybrid model with new data, refining detection rules and policies, and performing regular maintenance tasks to ensure that the system is functioning effectively.

CONVERSION PLAN

Converting an existing NIDS to use hybrid ML models involves several steps to ensure a smooth transition. Here is a general conversion plan for implementing hybrid ML models in an existing NIDS:

Assess current NIDS: The first step in the conversion plan is to assess the current NIDS to understand how it works, what data it is collecting, and how it is currently detecting network intrusions. This will help to identify areas where the hybrid ML model can be integrated.
Collect and pre-process data: The next step is to collect and pre-process the data that will be used to train the hybrid ML models. This involves selecting relevant data sources (e.g., network traffic logs, system logs, etc.), cleaning and normalizing the data, and transforming the data into a format that can be used for training.
Train the individual ML models: The next step is to train the individual ML models that will be used in the hybrid model. This may involve using a variety of algorithms, such as decision trees, support vector machines, and neural networks. Each model should be trained using a subset of the pre-processed data.
Integrate hybrid ML model: Once the individual models have been trained, the next step is to integrate them into the existing NIDS. This may involve modifying the detection rules and policies, as well as configuring the hybrid model to work with the NIDS.
Test the model: After the hybrid model has been integrated into the NIDS, it should be tested using a separate test set of data. This will help to evaluate the accuracy of the model and identify any areas where it may need improvement.
Deploy the model: Once the hybrid model has been tested and refined, it can be deployed as part of the NIDS. This may involve integrating it with other systems, configuring alerts and notifications, and setting up monitoring and reporting.
Monitor and maintain the system: Once the NIDS has been deployed, it is important to monitor and maintain it on an ongoing basis. This may involve updating the hybrid model with new data, refining detection rules and policies, and performing regular maintenance tasks to ensure that the system is functioning effectively.
Evaluate and optimize: It is important to regularly evaluate the performance of the hybrid ML model and identify areas for improvement. This may involve tweaking the algorithms or adjusting the model's parameters to improve detection accuracy or reduce false positives.
By following this conversion plan, organizations can successfully integrate hybrid ML models into their existing NIDS, resulting in a more accurate and effective system for detecting and preventing network intrusions.

POST-IMPLEMENTATION AND SOFTWARE MAINTENANCE

Post-implementation and software maintenance are essential and critical components of a successful NIDS deployment. Here are some key tasks that should be performed during this phase:

Regular performance monitoring: It is important to regularly monitor the performance of the NIDS using hybrid ML models. This can be done by analyzing log data, checking for false positives and false negatives, and conducting regular performance tests to ensure that the system is running smoothly.
Model updating and retraining: The hybrid ML model used in the NIDS will need to be updated and retrained regularly to ensure that it continues to accurately detect network intrusions. This may involve incorporating new data into the training set, tweaking the algorithms or adjusting the model's parameters to improve accuracy or reduce false positives.
Ongoing system maintenance: The NIDS should be regularly maintained to ensure that it is functioning properly. This may involve updating software components, applying patches and updates, and replacing hardware components as necessary.
Security updates and vulnerability testing: It is important to regularly test the NIDS for vulnerabilities and apply security updates as necessary to prevent attacks on the system.
Regular reporting and analysis: The NIDS should be regularly monitored and analysed to identify trends and patterns in network intrusions. This can help to identify new threats and improve the accuracy of the NIDS over time.
User training and education: Users and administrators should receive regular training and education on the use and maintenance of the NIDS. This can help to ensure that the system is being used properly and that any issues are quickly identified and resolved.
Disaster recovery planning: It is important to have a disaster recovery plan in place in case of a system failure or breach. This should include regular backups of system data and procedures for restoring the system in the event of a failure.
By following these post-implementation and software maintenance tasks, organizations can ensure that their NIDS using hybrid ML models remains effective and up-to-date, providing reliable protection against network intrusions.

PROJECT LEGACY

The project legacy for a NIDS refers to the documentation and records related to the project, which can be used to support ongoing maintenance, future upgrades, and other related activities.

CURRENT STATUS

Advancements in AI and ML: There have been significant advancements in AI and ML algorithms that can be used in hybrid models for NIDS. This has led to improved accuracy and efficiency in detecting network intrusions.
Integration with cloud platforms: Many organizations are now integrating their NIDS with cloud platforms, which provide greater scalability and flexibility for handling large amounts of data.
Use of big data technologies: The use of big data technologies such as Hadoop and Spark is becoming more common in NIDS, allowing for the analysis of large volumes of data in real time.
Increased focus on user behaviour analytics: User behaviour analytics is becoming a more important part of NIDS, as it can help to identify abnormal behaviour that may indicate a network intrusion.
Adoption of automation: Automation is increasingly being used in NIDS to improve the speed and accuracy of threat detection. This includes the use of machine learning algorithms for automated threat response.
Overall, the adoption of hybrid ML models in NIDS continuing is to evolve and improve, with a focus on greater accuracy, efficiency, and automation.

REMAINING AREAS OF CONCERN

While hybrid ML models have improved the accuracy and efficiency of NIDS, there are still some areas of concern that need to be addressed. Here are some of the key remaining areas of concern for NIDS using hybrid ML models:

False positives and false negatives: While hybrid ML models can reduce false positives and false negatives, they can still occur, especially in complex and dynamic network environments. This can result in unnecessary alerts and missed detections, which can impact the efficiency and reliability of the NIDS.
Limited interpretability: Hybrid ML models are often complex and difficult to interpret, making it challenging to understand why a particular detection was made or to identify areas for improvement. This can make it difficult for security analysts to effectively respond to network intrusions and make necessary adjustments to the NIDS.
Data privacy and security: NIDS requires access to sensitive network data, which raises concerns about data privacy and security. It is essential to ensure that the NIDS is secure and compliant with relevant data protection laws to prevent unauthorized access or misuse of sensitive data.
Limited availability of labelled data: Hybrid ML models require large amounts of labelled data for training, which can be challenging to obtain in some cases. This can limit the accuracy and effectiveness of the NIDS.
Cost and complexity: NIDS using hybrid ML models can be expensive and complex to implement and maintain. Organizations need to carefully consider the costs and resources required to deploy and maintain these systems.

To address these concerns, ongoing research and development are needed to improve the accuracy, interpretability, and scalability of hybrid ML models for NIDS. Additionally, organizations need to carefully consider the trade-offs between the benefits and costs of implementing and maintaining these systems, and take appropriate measures to ensure the privacy and security of sensitive data.

TECHNICAL AND MANAGERIAL LESSONS LEARNT

Using hybrid ML models in NIDS can provide valuable technical and managerial lessons for organizations. Here are some of the key lessons we can learn:

TECHNICAL LESSONS:

The importance of data quality: The accuracy and effectiveness of hybrid ML models depend heavily on the quality and quantity of the training data used. It is important to ensure that data is labelled accurately, is representative of real-world threats, and is updated regularly to reflect changes in the threat landscape.
The need for interpretability: The complexity of hybrid ML models can make it difficult to understand how they arrive at a particular detection or decision. Ensuring interpretability through techniques such as model explainability and feature importance analysis can help improve the trust and usability of the NIDS.
The need for ongoing tuning and optimization: Hybrid ML models require ongoing tuning and optimization to maintain their effectiveness in detecting network intrusions. This requires regular monitoring and analysis of system performance, as well as proactive measures to address false positives and false negatives.

MANAGERIAL LESSONS:

The importance of cross-functional collaboration: Implementing and maintaining a NIDS using hybrid ML models requires collaboration between IT security, data science, and business stakeholders. Effective communication and coordination are essential to ensure that the NIDS is aligned with organizational goals and meets business requirements.
The need for ongoing investment: Implementing and maintaining a NIDS using hybrid ML models can be expensive and resource-intensive. Organizations need to carefully consider the costs and benefits of investing in such systems and ensure that they have the necessary resources and expertise to support them over the long term.
The importance of compliance: NIDS using hybrid ML models can collect and process sensitive data, making compliance with data protection regulations essential. Organizations need to ensure that they have the necessary policies, processes, and controls in place to protect data privacy and security.
The importance of user education and awareness: NIDS using hybrid ML models relies on user behaviour analysis, making it important to educate employees and raise awareness around good security practices. This can help prevent false positives and improve the effectiveness of the NIDS.
Overall, using hybrid ML models in NIDS requires a multifaceted approach that involves technical expertise, cross-functional collaboration, ongoing investment, and compliance with relevant regulations. By learning these lessons, organizations can better implement and maintain NIDS using hybrid ML models to protect their networks from cyber threats.

USER MANUAL: A COMPLETE DOCUMENT OF THE SOFTWARE DEVELOPED

Introduction

The NSL-KDD dataset is a popular dataset for evaluating intrusion detection systems. This user manual provides instructions for applying machine learning algorithms to the NSL-KDD dataset to train a model for detecting network intrusions.

Prerequisites

Before running the code, you will need to have the following:

- Python installed on your computer
- The following Python libraries: Pandas, NumPy, Scikit-Learn
- Jupyter Notebook is better.

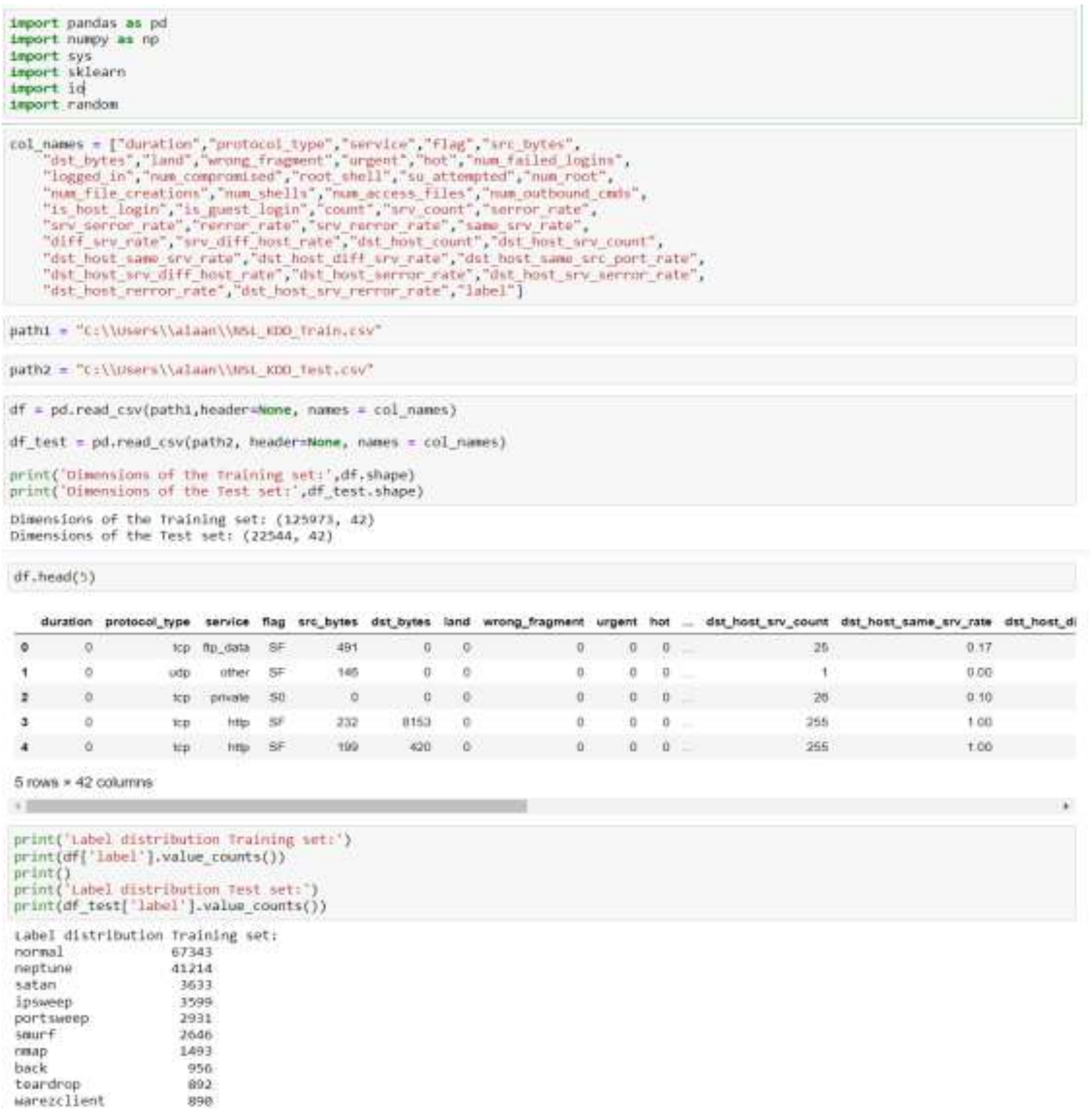
Running the Code

- ❖ Download the NSL-KDD dataset from the website: https://www.unb.ca/cic/datasets/nsl.html
- ❖ Save the dataset to a local directory on your computer.
- ❖ Open the Python code file for applying machine learning algorithms to the NSL-KDD dataset.
- ❖ Modify the code to specify the directory where the dataset is stored.
- ❖ Run the code.
- ❖ The code will pre-process the data by removing duplicate entries, removing irrelevant columns, encoding categorical variables, normalizing numerical variables, and splitting the dataset into training and testing sets.
- ❖ Select the machine learning algorithm to use by choosing the appropriate algorithm for the problem and determining the hyperparameters for the algorithm.
- ❖ Train the algorithm by fitting the model to the training data and optimizing the hyperparameters using cross-validation or grid search.
- ❖ Evaluate the model by using the testing set to evaluate the model's performance and calculate performance metrics such as accuracy, precision, recall, and F1-score.
- ❖ If necessary, tune the model by adjusting the hyperparameters to improve the model's performance and re-evaluate the model using the testing set.
- ❖ Deploy the model by using the model to make predictions on new data and monitor the model's performance over time and retraining if necessary.

Conclusion

This user manual has provided instructions for applying machine learning algorithms to the NSL-KDD dataset to train a model for detecting network intrusions. By following these instructions, you can evaluate the performance of different machine learning algorithms and select the best algorithm for your specific needs.

SOURCE CODE



```
import pandas as pd
import numpy as np
import sys
import sklearn
import io
import random
```

```
col_names = ["duration","protocol_type","service","flag","src_bytes",
    "dst_bytes","land","wrong_fragment","urgent","hot","num_failed_logins",
    "logged_in","num_compromised","root_shell","su_attempted","num_root",
    "num_file_creations","num_shells","num_access_files","num_outbound_cmds",
    "is_host_login","is_guest_login","count","srv_count","serror_rate",
    "srv_serror_rate","rerror_rate","srv_rerror_rate","same_srv_rate",
    "diff_srv_rate","srv_diff_host_rate","dst_host_count","dst_host_srv_count",
    "dst_host_same_srv_rate","dst_host_diff_srv_rate","dst_host_same_src_port_rate",
    "dst_host_srv_diff_host_rate","dst_host_serror_rate","dst_host_srv_serror_rate",
    "dst_host_rerror_rate","dst_host_srv_rerror_rate","label"]
```

```
path1 = "C:\\Users\\alaan\\NSL_KDD_Train.csv"
```

```
path2 = "C:\\Users\\alaan\\NSL_KDD_Test.csv"
```

```
df = pd.read_csv(path1,header=None, names = col_names)

df_test = pd.read_csv(path2, header=None, names = col_names)

print('Dimensions of the Training set:',df.shape)
print('Dimensions of the Test set:',df_test.shape)
```

```
Dimensions of the Training set: (125973, 42)
Dimensions of the Test set: (22544, 42)
```

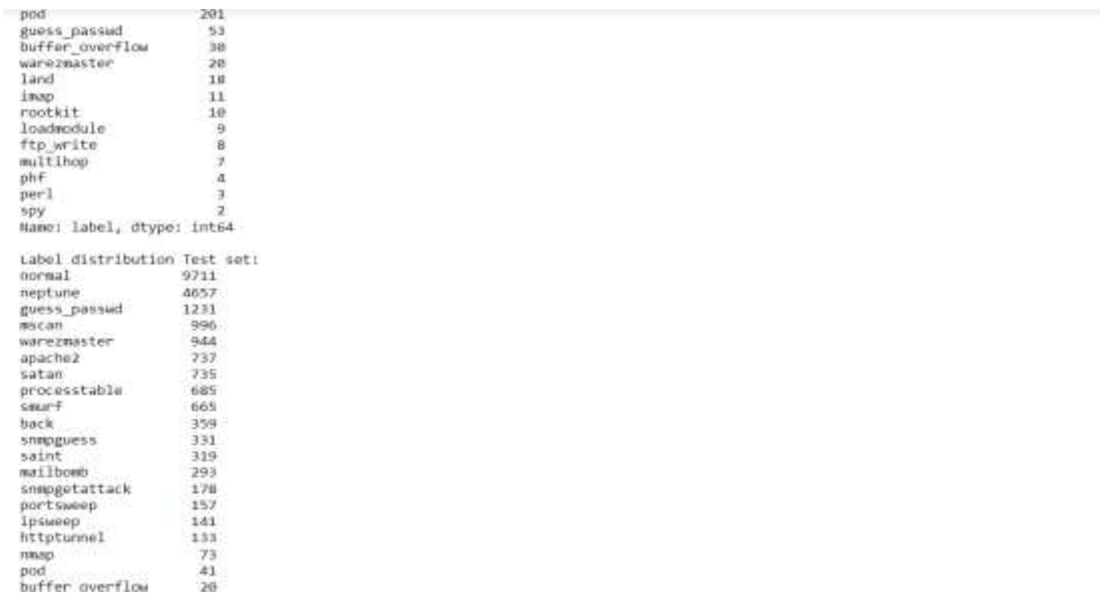
```
df.head(5)
```

	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	...	dst_host_srv_count	dst_host_same_srv_rate	dst_host_di
0	0	tcp	ftp_data	SF	491	0	0	0	0	0	...	25	0.17	
1	0	udp	other	SF	146	0	0	0	0	0	...	1	0.00	
2	0	tcp	private	S0	0	0	0	0	0	0	...	26	0.10	
3	0	tcp	http	SF	232	8153	0	0	0	0	...	255	1.00	
4	0	tcp	http	SF	199	420	0	0	0	0	...	255	1.00	

5 rows × 42 columns

```
print('Label distribution Training set:')
print(df['label'].value_counts())
print()
print('Label distribution Test set:')
print(df_test['label'].value_counts())
```

```
Label distribution Training set:
normal           67343
neptune          41214
satan             3633
ipsweep           3599
portsweep         2931
smurf             2646
nmap              1493
back               956
teardrop           892
warezclient        890
```



```
pod                201
guess_passwd        53
buffer_overflow     30
warezmaster         20
land                18
imap                11
rootkit             10
loadmodule           9
ftp_write            8
multihop             7
phf                  4
perl                 3
spy                  2
Name: label, dtype: int64

Label distribution Test set:
normal           9711
neptune          4657
guess_passwd     1231
mscan             996
warezmaster       944
apache2           737
satan             735
processtable      685
smurf             665
back              359
snmpguess         331
saint             319
mailbomb          293
snmpgetattack     178
portsweep         157
ipsweep           141
httptunnel        133
nmap               73
pod                41
buffer_overflow    20
buffer_overflow      20
multihop             18
named                17
ps                   15
sendmail             14
rootkit              13
xterm                13
teardrop             12
xlock                 9
land                  7
xsnoop                4
ftp_write             3
worm                  2
loadmodule            2
perl                  2
sqlattack             2
udpstorm              2
phf                   2
imap                  1
Name: label, dtype: int64
```

Step 1: Data preprocessing:

```
print('Training set:')
for col_name in df.columns:
    if df[col_name].dtypes == 'object' :
        unique_cat = len(df[col_name].unique())
        print("Feature '{col_name}' has {unique_cat} categories".format(col_name=col_name, unique_cat=unique_cat))

print()
print('Distribution of categories in service:')
print(df['service'].value_counts().sort_values(ascending=False).head())
```

```
Training set:
Feature 'protocol_type' has 3 categories
Feature 'service' has 70 categories
Feature 'flag' has 11 categories
Feature 'label' has 23 categories

Distribution of categories in service:
http         40338
private      21853
domain_u      9043
smtp          7313
ftp_data      6860
Name: service, dtype: int64
```

```
# Test set
print('Test set:')
for col_name in df_test.columns:
    if df_test[col_name].dtypes == 'object' :
        unique_cat = len(df_test[col_name].unique())
        print("Feature '{col_name}' has {unique_cat} categories".format(col_name=col_name, unique_cat=unique_cat))
```

```
Test set:
Feature 'protocol_type' has 3 categories
Feature 'service' has 64 categories
Feature 'flag' has 11 categories
Feature 'label' has 38 categories
```

LabelEncoder

Insert categorical features into a 2D numpy array

```
from sklearn.preprocessing import LabelEncoder,OneHotEncoder
categorical_columns=['protocol_type', 'service', 'flag']
```

```
df_categorical_values = df[categorical_columns]
testdf_categorical_values = df_test[categorical_columns]

df_categorical_values.head()
```

	protocol_type	service	flag
0	tcp	ftp_data	SF
1	udp	other	SF
2	tcp	private	S0
3	tcp	http	SF
4	tcp	http	SF

```
# protocol type
unique_protocol=sorted(df.protocol_type.unique())
string1 = 'Protocol_type_'
unique_protocol2=[string1 + x for x in unique_protocol]
print(unique_protocol2)

# service
unique_service=sorted(df.service.unique())
string2 = 'service_'
unique_service2=[string2 + x for x in unique_service]
print(unique_service2)


# flag
unique_flag=sorted(df.flag.unique())
string3 = 'flag_'
unique_flag2=[string3 + x for x in unique_flag]
print(unique_flag2)
```

```
# put together
dumcols=unique_protocol2 + unique_service2 + unique_flag2


#do it for test set
unique_service_test=sorted(df_test.service.unique())
unique_service2_test=[string2 + x for x in unique_service_test]
testdumcols=unique_protocol2 + unique_service2_test + unique_flag2
```

```
['Protocol_type_icmp', 'Protocol_type_tcp', 'Protocol_type_udp']
['service_IRC', 'service_X11', 'service_Z39_50', 'service_aol', 'service_auth', 'service_bgp', 'service_courier', 'service_csne
t_ns', 'service_ctf', 'service_daytime', 'service_discard', 'service_domain', 'service_domain_u', 'service_echo', 'service_eco_
i', 'service_ecr_i', 'service_efs', 'service_exec', 'service_finger', 'service_ftp', 'service_ftp_data', 'service_gopher', 'ser
vice_harvest', 'service_hostnames', 'service_http', 'service_http_2784', 'service_http_443', 'service_http_8001', 'service_imap
4', 'service_iso_tsap', 'service_klogin', 'service_kshell', 'service_ldap', 'service_link', 'service_login', 'service_mtp', 'se
rvice_name', 'service_netbios_dgm', 'service_netbios_ns', 'service_netbios_ssn', 'service_netstat', 'service_nnsp', 'service_nn
tp', 'service_ntp_u', 'service_other', 'service_pm_dump', 'service_pop_2', 'service_pop_3', 'service_printer', 'service_privat
e', 'service_red_i', 'service_remote_job', 'service_rje', 'service_shell', 'service_smtp', 'service_sql_net', 'service_ssh', 's
ervice_sunrpc', 'service_supdup', 'service_systat', 'service_telnet', 'service_tftp_u', 'service_tim_i', 'service_time', 'servi
ce_urh_i', 'service_urp_i', 'service_uucp', 'service_uucp_path', 'service_vmnet', 'service_whois']
['flag_OTH', 'flag_REJ', 'flag_RSTO', 'flag_RSTOS0', 'flag_RSTR', 'flag_S0', 'flag_S1', 'flag_S2', 'flag_S3', 'flag_SF', 'flag_
SH']
```

Transform categorical features into numbers using LabelEncoder()

```
df_categorical_values_enc=df_categorical_values.apply(LabelEncoder().fit_transform)

print(df_categorical_values.head())
print('--------------------')
print(df_categorical_values_enc.head())

# test set
testdf_categorical_values_enc=testdf_categorical_values.apply(LabelEncoder().fit_transform)
```

```
  protocol_type   service flag
0           tcp  ftp_data   SF
1           udp     other   SF
2           tcp   private   S0
3           tcp      http   SF
4           tcp      http   SF
--------------------
   protocol_type  service  flag
0              1       20     9
1              2       44     9
2              1       49     5
3              1       24     9
4              1       24     9
```

One-Hot-Encoding

```
enc = OneHotEncoder(categories='auto')
df_categorical_values_encenc = enc.fit_transform(df_categorical_values_enc)
df_cat_data = pd.DataFrame(df_categorical_values_encenc.toarray(),columns=dumcols)


# test set
testdf_categorical_values_encenc = enc.fit_transform(testdf_categorical_values_enc)
testdf_cat_data = pd.DataFrame(testdf_categorical_values_encenc.toarray(),columns=testdumcols)

df_cat_data.head()
```

	Protocol_type_icmp	Protocol_type_tcp	Protocol_type_udp	service_IRC	service_X11	service_Z39_50	service_aol	service_auth	service_bgp	service_courier
0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

5 rows × 84 columns

Missing columns in the test set are added

```
trainservice=df['service'].tolist()
testservice= df_test['service'].tolist()
difference=list(set(trainservice) - set(testservice))
string = 'service_'
difference=[string + x for x in difference]
difference
```

```
['service_aol',
 'service_harvest',
 'service_urh_i',
 'service_http_2784',
 'service_http_8001',
 'service_red_i']
```

```
for col in difference:
    testdf_cat_data[col] = 0

print(df_cat_data.shape)
print(testdf_cat_data.shape)
```

```
# test data
newdf_test=df_test.join(testdf_cat_data)
newdf_test.drop('flag', axis=1, inplace=True)
newdf_test.drop('protocol_type', axis=1, inplace=True)
newdf_test.drop('service', axis=1, inplace=True)

print(newdf.shape)
print(newdf_test.shape)
```

```
(125973, 123)
(22544, 123)
```

The dataset was divided into separate datasets for each attack category. Attack tags have been renamed for each. 0=Normal, 1=DoS, 2=Probe, 3=R2L, 4=U2R. In new datasets, the label column is replaced with new values.

DoS :

Probe :

R2L :

U2R :

```
labeldf=newdf['label']
labeldf_test=newdf_test['label']


# change the label column
newlabeldf=labeldf.replace({ 'normal' : 0, 'neptune' : 1 ,'back': 1, 'land': 1, 'pod': 1, 'smurf': 1, 'teardrop': 1,'mailbomb': 1
                           'ipsweep' : 2,'nmap' : 2,'portsweep' : 2,'satan' : 2,'mscan' : 2,'saint' : 2
                           ,'ftp_write': 3,'guess_passwd': 3,'imap': 3,'multihop': 3,'phf': 3,'spy': 3,'warezclient': 3,'warezmas
                           'buffer_overflow': 4,'loadmodule': 4,'perl': 4,'rootkit': 4,'ps': 4,'sqlattack': 4,'xterm': 4})
newlabeldf_test=labeldf_test.replace({ 'normal' : 0, 'neptune' : 1 ,'back': 1, 'land': 1, 'pod': 1, 'smurf': 1, 'teardrop': 1,'ma
                           'ipsweep' : 2,'nmap' : 2,'portsweep' : 2,'satan' : 2,'mscan' : 2,'saint' : 2
```

```
newlabeldf=labeldf.replace({ 'normal' : 0, 'neptune' : 1 ,'back': 1, 'land': 1, 'pod': 1, 'smurf': 1, 'teardrop': 1,'mailbomb': 1
                           'ipsweep' : 2,'nmap' : 2,'portsweep' : 2,'satan' : 2,'mscan' : 2,'saint' : 2
                           ,'ftp_write': 3,'guess_passwd': 3,'imap': 3,'multihop': 3,'phf': 3,'spy': 3,'warezclient': 3,'warezmas
                           'buffer_overflow': 4,'loadmodule': 4,'perl': 4,'rootkit': 4,'ps': 4,'sqlattack': 4,'xterm': 4})
newlabeldf_test=labeldf_test.replace({ 'normal' : 0, 'neptune' : 1 ,'back': 1, 'land': 1, 'pod': 1, 'smurf': 1, 'teardrop': 1,'ma
                           'ipsweep' : 2,'nmap' : 2,'portsweep' : 2,'satan' : 2,'mscan' : 2,'saint' : 2
                           ,'ftp_write': 3,'guess_passwd': 3,'imap': 3,'multihop': 3,'phf': 3,'spy': 3,'warezclient': 3,'warezmas
                           'buffer_overflow': 4,'loadmodule': 4,'perl': 4,'rootkit': 4,'ps': 4,'sqlattack': 4,'xterm': 4})


# put the new label column back
newdf['label'] = newlabeldf
newdf_test['label'] = newlabeldf_test
```

```
to_drop_DoS = [0,1]
to_drop_Probe = [0,2]
to_drop_R2L = [0,3]
to_drop_U2R = [0,4]

# Filter all rows with label value other than itself. Normal attack type is reference point for all the other attacks
# job filter function

DoS_df=newdf[newdf['label'].isin(to_drop_DoS)];
Probe_df=newdf[newdf['label'].isin(to_drop_Probe)];
R2L_df=newdf[newdf['label'].isin(to_drop_R2L)];
U2R_df=newdf[newdf['label'].isin(to_drop_U2R)];


#test
DoS_df_test=newdf_test[newdf_test['label'].isin(to_drop_DoS)];
Probe_df_test=newdf_test[newdf_test['label'].isin(to_drop_Probe)];
```

```
R2L_df_test=newdf_test[newdf_test['label'].isin(to_drop_R2L)];
U2R_df_test=newdf_test[newdf_test['label'].isin(to_drop_U2R)];


print('Train:')
print('Dimensions of DoS:' ,DoS_df.shape)
print('Dimensions of Probe:' ,Probe_df.shape)
print('Dimensions of R2L:' ,R2L_df.shape)
print('Dimensions of U2R:' ,U2R_df.shape)
print()
print('Test:')
print('Dimensions of DoS:' ,DoS_df_test.shape)
print('Dimensions of Probe:' ,Probe_df_test.shape)
print('Dimensions of R2L:' ,R2L_df_test.shape)
print('Dimensions of U2R:' ,U2R_df_test.shape)
```

```
Train:
Dimensions of DoS: (113270, 123)
Dimensions of Probe: (78999, 123)
Dimensions of R2L: (68338, 123)
Dimensions of U2R: (67395, 123)

Test:
Dimensions of DoS: (17171, 123)
Dimensions of Probe: (12132, 123)
Dimensions of R2L: (12596, 123)
Dimensions of U2R: (9778, 123)
```

Step 2: Feature Scaling

```
# Split dataframes into X & Y
# X Properties , Y result variables

X_DoS = DoS_df.drop('label',1)
```

```
Y_Probe = Probe_df.label

X_R2L = R2L_df.drop('label',1)
Y_R2L = R2L_df.label

X_U2R = U2R_df.drop('label',1)
Y_U2R = U2R_df.label

# test set
X_DoS_test = DoS_df_test.drop('label',1)
Y_DoS_test = DoS_df_test.label

X_Probe_test = Probe_df_test.drop('label',1)
Y_Probe_test = Probe_df_test.label

X_R2L_test = R2L_df_test.drop('label',1)
Y_R2L_test = R2L_df_test.label

X_U2R_test = U2R_df_test.drop('label',1)
Y_U2R_test = U2R_df_test.label
```

As the column names will be deleted at this stage, we save the column names for later use.

```
colNames=list(X_DoS)
colNames_test=list(X_DoS_test)
```

```
from sklearn import preprocessing

scaler1 = preprocessing.StandardScaler().fit(X_DoS)
X_DoS=scaler1.transform(X_DoS)

scaler2 = preprocessing.StandardScaler().fit(X_Probe)
X_Probe=scaler2.transform(X_Probe)

scaler3 = preprocessing.StandardScaler().fit(X_R2L)
X_R2L=scaler3.transform(X_R2L)

scaler4 = preprocessing.StandardScaler().fit(X_U2R)
X_U2R=scaler4.transform(X_U2R)

# test data
scaler5 = preprocessing.StandardScaler().fit(X_DoS_test)
X_DoS_test=scaler5.transform(X_DoS_test)

scaler6 = preprocessing.StandardScaler().fit(X_Probe_test)
X_Probe_test=scaler6.transform(X_Probe_test)

scaler7 = preprocessing.StandardScaler().fit(X_R2L_test)
X_R2L_test=scaler7.transform(X_R2L_test)

scaler8 = preprocessing.StandardScaler().fit(X_U2R_test)
X_U2R_test=scaler8.transform(X_U2R_test)
```

```
from sklearn.model_selection import cross_val_score
from sklearn import metrics
```

Recursive Feature Elimination (RFE) , top 13 features (as a group)

Random Forest

```
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestClassifier


clf = RandomForestClassifier(n_estimators=10,n_jobs=2)
rfe = RFE(estimator=clf, n_features_to_select=13, step=1) #performing recursive feature elimination selecting 13 most important c

rfe.fit(X_DoS, Y_DoS.astype(int)) # This line fits the RFE on the training data X_DoS with the corresponding target labels Y_DoS,
X_rfeDoS=rfe.transform(X_DoS) #applies the feature selection to the original dataset X_DoS, which returns a new dataset X_rfeDoS
true=rfe.support_ #retrieves the boolean mask indicating which features were selected by the RFE. This boolean mask has a True va
rfecolindex_DoS=[i for i, x in enumerate(true) if x] #line creates a list containing the indices of the selected features by iter
rfecolname_DoS=list(colNames[i] for i in rfecolindex_DoS) #line creates a list rfecolname_DoS containing the names of the selecte
```

```
rfe.fit(X_Probe, Y_Probe.astype(int))
X_rfeProbe=rfe.transform(X_Probe)
true=rfe.support_
rfecolindex_Probe=[i for i, x in enumerate(true) if x]
rfecolname_Probe=list(colNames[i] for i in rfecolindex_Probe)
```

```
rfe.fit(X_R2L, Y_R2L.astype(int))
X_rfeR2L=rfe.transform(X_R2L)
true=rfe.support_
rfecolindex_R2L=[i for i, x in enumerate(true) if x]
rfecolname_R2L=list(colNames[i] for i in rfecolindex_R2L)
```

```
rfe.fit(X_U2R, Y_U2R.astype(int))
X_rfeU2R=rfe.transform(X_U2R)
true=rfe.support_
rfecolindex_U2R=[i for i, x in enumerate(true) if x]
rfecolname_U2R=list(colNames[i] for i in rfecolindex_U2R)
```

Step 4: Build the model:

Classifier is trained for all features and for reduced features, for later comparison.

The classifier model itself is stored in the clf variable.

```
# all features
clf_DoS=RandomForestClassifier(n_estimators=10,n_jobs=2)
clf_Probe=RandomForestClassifier(n_estimators=10,n_jobs=2)
clf_R2L=RandomForestClassifier(n_estimators=10,n_jobs=2)
clf_U2R=RandomForestClassifier(n_estimators=10,n_jobs=2)
clf_DoS.fit(X_DoS, Y_DoS.astype(int))
clf_Probe.fit(X_Probe, Y_Probe.astype(int))
clf_R2L.fit(X_R2L, Y_R2L.astype(int))
clf_U2R.fit(X_U2R, Y_U2R.astype(int))
```

DoS

```
from sklearn.model_selection import cross_val_score
from sklearn import metrics

scoring = ['accuracy', 'precision', 'recall', 'f1']
for score in scoring:
    scores = cross_val_score(clf_DoS, X_DoS_test, Y_DoS_test, cv=10, scoring=score)
    print("%s: %0.5f" % (score.capitalize(), scores.mean()))
```

```
Accuracy: 0.99785
Precision: 0.99852
Recall: 0.99785
F1: 0.99758
```

Probe

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99662
Precision_macro: 0.99555
Recall_macro: 0.99277
F1_macro: 0.99482
```

U2R

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99775
Precision_macro: 0.94754
Recall_macro: 0.87956
F1_macro: 0.90277
```

R2L

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.98079
Precision_macro: 0.97425
Recall_macro: 0.96954
F1 macro: 0.97286
```

KNeighbors

```
from sklearn.neighbors import KNeighborsClassifier

clf_KNN_DoS=KNeighborsClassifier()
clf_KNN_Probe=KNeighborsClassifier()
clf_KNN_R2L=KNeighborsClassifier()
clf_KNN_U2R=KNeighborsClassifier()

clf_KNN_DoS.fit(X_DoS, Y_DoS.astype(int))
clf_KNN_Probe.fit(X_Probe, Y_Probe.astype(int))
clf_KNN_R2L.fit(X_R2L, Y_R2L.astype(int))
clf_KNN_U2R.fit(X_U2R, Y_U2R.astype(int))
```

```
KNeighborsClassifier()
```

DoS

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_KNN_DoS, X_DoS_test, Y_DoS_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99715
Precision_macro: 0.99711
Recall_macro: 0.99709
F1_macro: 0.99710
```

Probe

```python
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_KNN_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99877
Precision_macro: 0.98606
Recall_macro: 0.98508
F1_macro: 0.98553
```

R2L

```python
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_KNN_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.96785
Precision_macro: 0.95265
Recall_macro: 0.95439
F1_macro: 0.95344
```

U2R

```python
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_KNN_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99703
Precision_macro: 0.93143
Recall_macro: 0.85073
F1_macro: 0.87831
```

SVM

```python
from sklearn.svm import SVC

clf_SVM_DoS=SVC(kernel='linear', C=1.0, random_state=0)
clf_SVM_Probe=SVC(kernel='linear', C=1.0, random_state=0)
clf_SVM_R2L=SVC(kernel='linear', C=1.0, random_state=0)
clf_SVM_U2R=SVC(kernel='linear', C=1.0, random_state=0)

clf_SVM_DoS.fit(X_DoS, Y_DoS.astype(int))
clf_SVM_Probe.fit(X_Probe, Y_Probe.astype(int))
clf_SVM_R2L.fit(X_R2L, Y_R2L.astype(int))
clf_SVM_U2R.fit(X_U2R, Y_U2R.astype(int))
```

```
SVC(kernel='linear', random_state=0)
```

DoS

```python
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_SVM_DoS, X_DoS_test, Y_DoS_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99371
Precision_macro: 0.99342
Recall_macro: 0.99380
F1_macro: 0.99360
```

Probe

```python
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_SVM_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.98450
Precision_macro: 0.96907
Recall_macro: 0.98365
F1_macro: 0.97613
```

R2L

```python
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_SVM_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.96793
Precision_macro: 0.94854
Recall_macro: 0.96264
F1_macro: 0.95520
```

U2R

```python
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_SVM_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99652
Precision_macro: 0.91056
Recall_macro: 0.82909
F1_macro: 0.86889
```

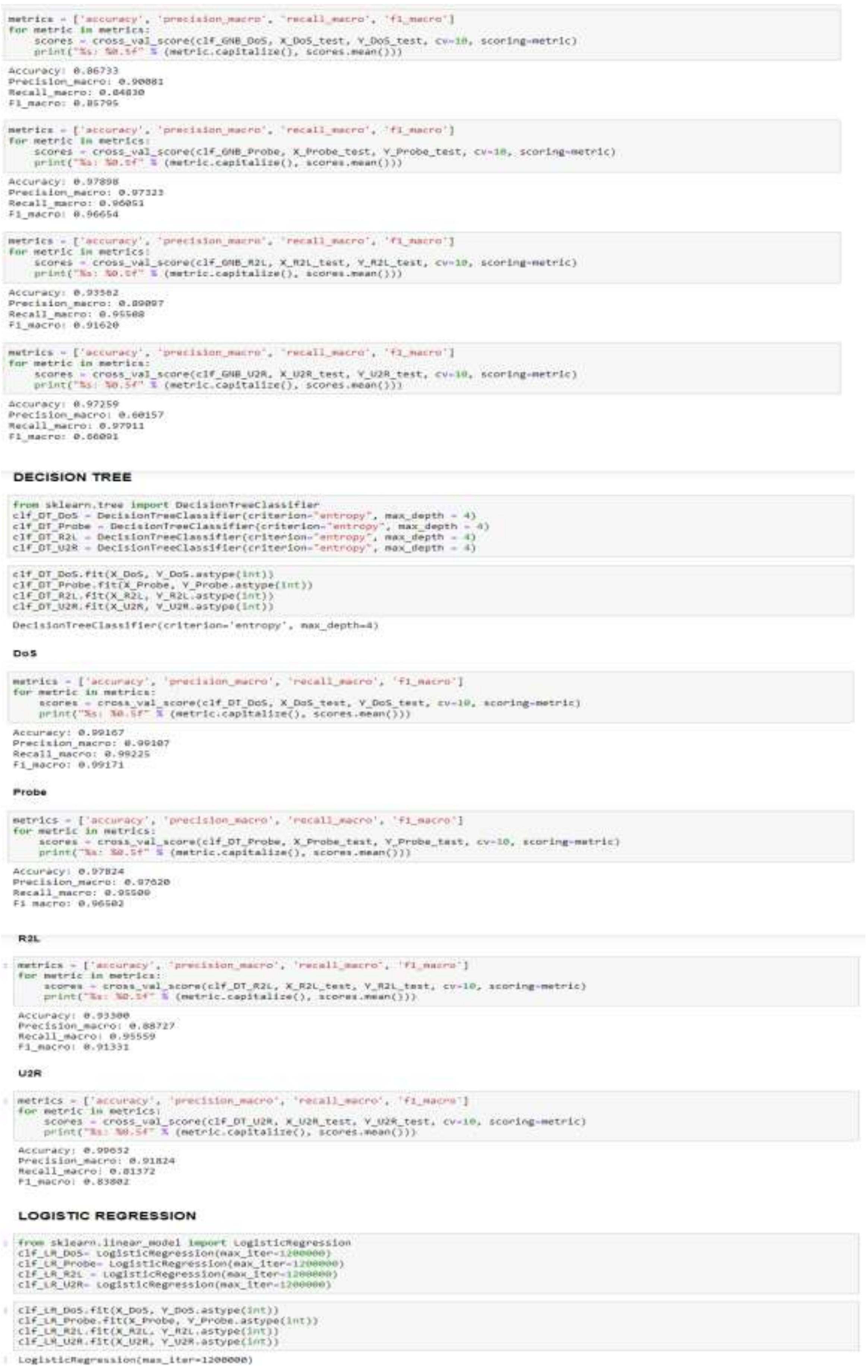
GAUSSIAN NAIVE BAYES

```python
from sklearn.naive_bayes import GaussianNB
```

```python
clf_GNB_DoS=GaussianNB()
clf_GNB_Probe=GaussianNB()
clf_GNB_R2L=GaussianNB()
clf_GNB_U2R=GaussianNB()
```

```python
clf_GNB_DoS.fit(X_DoS, Y_DoS.astype(int))
clf_GNB_Probe.fit(X_Probe, Y_Probe.astype(int))
clf_GNB_R2L.fit(X_R2L, Y_R2L.astype(int))
clf_GNB_U2R.fit(X_U2R, Y_U2R.astype(int))
```

```
GaussianNB()
```



```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_GNB_DoS, X_DoS_test, Y_DoS_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.86733
Precision_macro: 0.90881
Recall_macro: 0.84830
F1_macro: 0.85795
```

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_GNB_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.97898
Precision_macro: 0.97323
Recall_macro: 0.96051
F1_macro: 0.96654
```

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_GNB_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.93582
Precision_macro: 0.89097
Recall_macro: 0.95508
F1_macro: 0.91620
```

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_GNB_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.97259
Precision_macro: 0.60157
Recall_macro: 0.97911
F1_macro: 0.66091
```

DECISION TREE

```
from sklearn.tree import DecisionTreeClassifier
clf_DT_DoS = DecisionTreeClassifier(criterion="entropy", max_depth = 4)
clf_DT_Probe = DecisionTreeClassifier(criterion="entropy", max_depth = 4)
clf_DT_R2L = DecisionTreeClassifier(criterion="entropy", max_depth = 4)
clf_DT_U2R = DecisionTreeClassifier(criterion="entropy", max_depth = 4)
```

```
clf_DT_DoS.fit(X_DoS, Y_DoS.astype(int))
clf_DT_Probe.fit(X_Probe, Y_Probe.astype(int))
clf_DT_R2L.fit(X_R2L, Y_R2L.astype(int))
clf_DT_U2R.fit(X_U2R, Y_U2R.astype(int))
```

```
DecisionTreeClassifier(criterion='entropy', max_depth=4)
```

DoS

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_DT_DoS, X_DoS_test, Y_DoS_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99167
Precision_macro: 0.99107
Recall_macro: 0.99225
F1_macro: 0.99171
```

Probe

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_DT_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.97824
Precision_macro: 0.97620
Recall_macro: 0.95500
F1 macro: 0.96502
```

R2L

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_DT_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.93300
Precision_macro: 0.88727
Recall_macro: 0.95559
F1_macro: 0.91331
```

U2R

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_DT_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

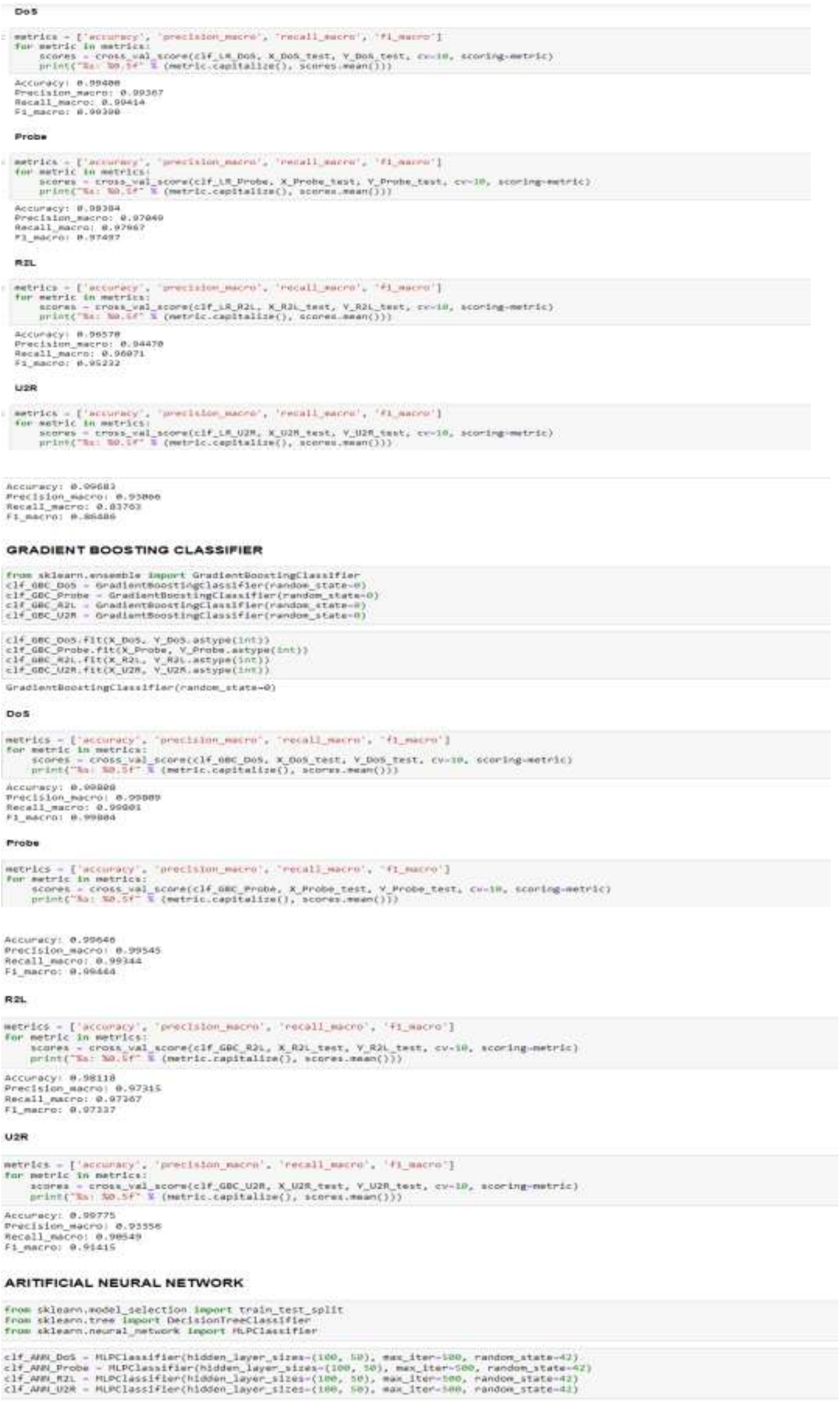
```
Accuracy: 0.99632
Precision_macro: 0.91824
Recall_macro: 0.81372
F1_macro: 0.83802
```

LOGISTIC REGRESSION

```
from sklearn.linear_model import LogisticRegression
clf_LR_DoS= LogisticRegression(max_iter=1200000)
clf_LR_Probe= LogisticRegression(max_iter=1200000)
clf_LR_R2L = LogisticRegression(max_iter=1200000)
clf_LR_U2R= LogisticRegression(max_iter=1200000)
```

```
clf_LR_DoS.fit(X_DoS, Y_DoS.astype(int))
clf_LR_Probe.fit(X_Probe, Y_Probe.astype(int))
clf_LR_R2L.fit(X_R2L, Y_R2L.astype(int))
clf_LR_U2R.fit(X_U2R, Y_U2R.astype(int))
```

```
LogisticRegression(max_iter=1200000)
```



DoS

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_LR_DoS, X_DoS_test, Y_DoS_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99400
Precision_macro: 0.99367
Recall_macro: 0.99414
F1_macro: 0.99390
```

Probe

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_LR_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.98384
Precision_macro: 0.97040
Recall_macro: 0.97967
F1_macro: 0.97497
```

R2L

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_LR_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.96570
Precision_macro: 0.94470
Recall_macro: 0.96071
F1_macro: 0.95232
```

U2R

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_LR_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99683
Precision_macro: 0.93006
Recall_macro: 0.83763
F1_macro: 0.86486
```

GRADIENT BOOSTING CLASSIFIER

```
from sklearn.ensemble import GradientBoostingClassifier
clf_GBC_DoS = GradientBoostingClassifier(random_state=0)
clf_GBC_Probe = GradientBoostingClassifier(random_state=0)
clf_GBC_R2L = GradientBoostingClassifier(random_state=0)
clf_GBC_U2R = GradientBoostingClassifier(random_state=0)
```

```
clf_GBC_DoS.fit(X_DoS, Y_DoS.astype(int))
clf_GBC_Probe.fit(X_Probe, Y_Probe.astype(int))
clf_GBC_R2L.fit(X_R2L, Y_R2L.astype(int))
clf_GBC_U2R.fit(X_U2R, Y_U2R.astype(int))
```

```
GradientBoostingClassifier(random_state=0)
```

DoS

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_GBC_DoS, X_DoS_test, Y_DoS_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99808
Precision_macro: 0.99809
Recall_macro: 0.99801
F1_macro: 0.99804
```

Probe

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_GBC_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99646
Precision_macro: 0.99545
Recall_macro: 0.99344
F1_macro: 0.99444
```

R2L

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_GBC_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.98118
Precision_macro: 0.97315
Recall_macro: 0.97367
F1_macro: 0.97337
```

U2R

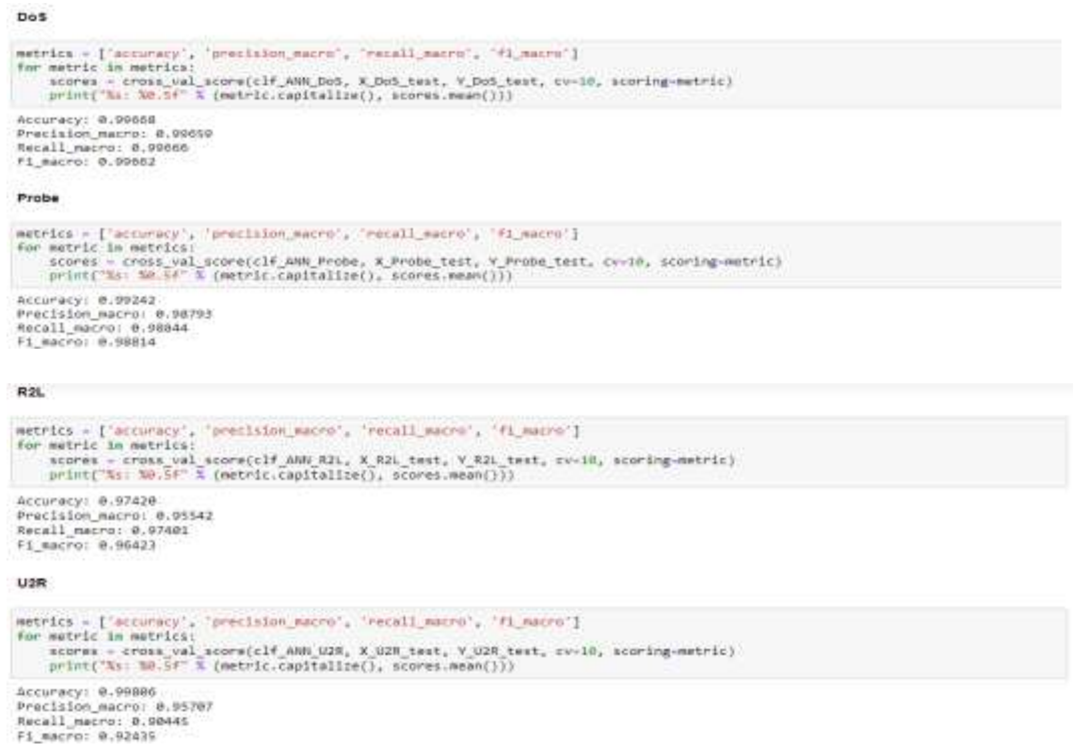
```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_GBC_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99775
Precision_macro: 0.93556
Recall_macro: 0.90549
F1_macro: 0.91415
```

ARITIFICIAL NEURAL NETWORK

```
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.neural_network import MLPClassifier
```

```
clf_ANN_DoS = MLPClassifier(hidden_layer_sizes=(100, 50), max_iter=500, random_state=42)
clf_ANN_Probe = MLPClassifier(hidden_layer_sizes=(100, 50), max_iter=500, random_state=42)
clf_ANN_R2L = MLPClassifier(hidden_layer_sizes=(100, 50), max_iter=500, random_state=42)
clf_ANN_U2R = MLPClassifier(hidden_layer_sizes=(100, 50), max_iter=500, random_state=42)
```



DoS

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_ANN_DoS, X_DoS_test, Y_DoS_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99668
Precision_macro: 0.99659
Recall_macro: 0.99666
F1_macro: 0.99662
```

Probe

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_ANN_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99242
Precision_macro: 0.98793
Recall_macro: 0.98844
F1_macro: 0.98814
```

R2L

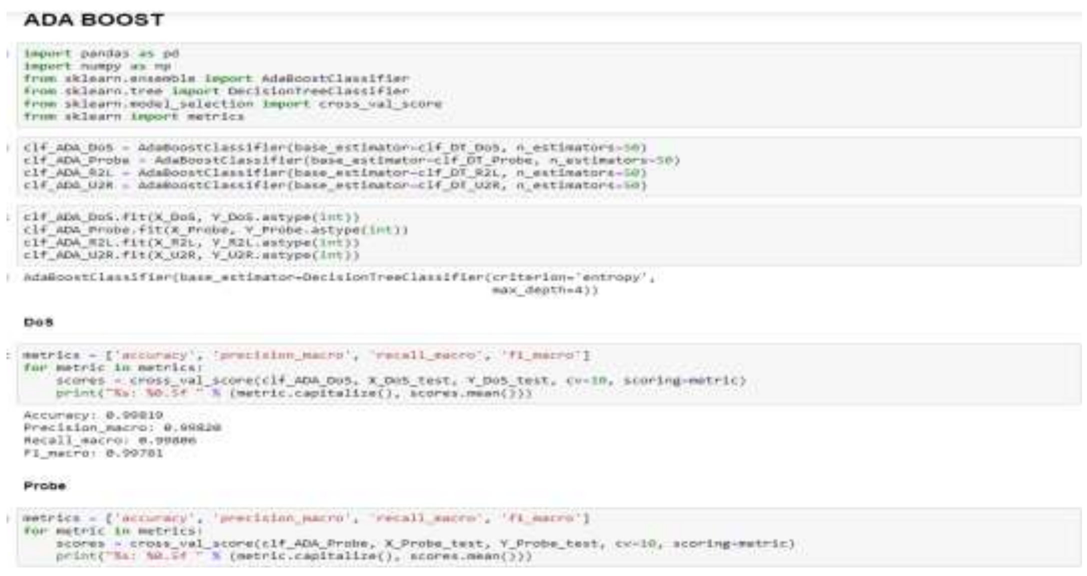
```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_ANN_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.97420
Precision_macro: 0.95542
Recall_macro: 0.97401
F1_macro: 0.96423
```

U2R

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_ANN_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring=metric)
    print("%s: %0.5f" % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99886
Precision_macro: 0.95707
Recall_macro: 0.90445
F1_macro: 0.92435
```



ADA BOOST

```
import pandas as pd
import numpy as np
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score
from sklearn import metrics
```

```
clf_ADA_DoS = AdaBoostClassifier(base_estimator=clf_DT_DoS, n_estimators=50)
clf_ADA_Probe = AdaBoostClassifier(base_estimator=clf_DT_Probe, n_estimators=50)
clf_ADA_R2L = AdaBoostClassifier(base_estimator=clf_DT_R2L, n_estimators=50)
clf_ADA_U2R = AdaBoostClassifier(base_estimator=clf_DT_U2R, n_estimators=50)
```

```
clf_ADA_DoS.fit(X_DoS, Y_DoS.astype(int))
clf_ADA_Probe.fit(X_Probe, Y_Probe.astype(int))
clf_ADA_R2L.fit(X_R2L, Y_R2L.astype(int))
clf_ADA_U2R.fit(X_U2R, Y_U2R.astype(int))
```

```
AdaBoostClassifier(base_estimator=DecisionTreeClassifier(criterion='entropy',
                                                         max_depth=4))
```

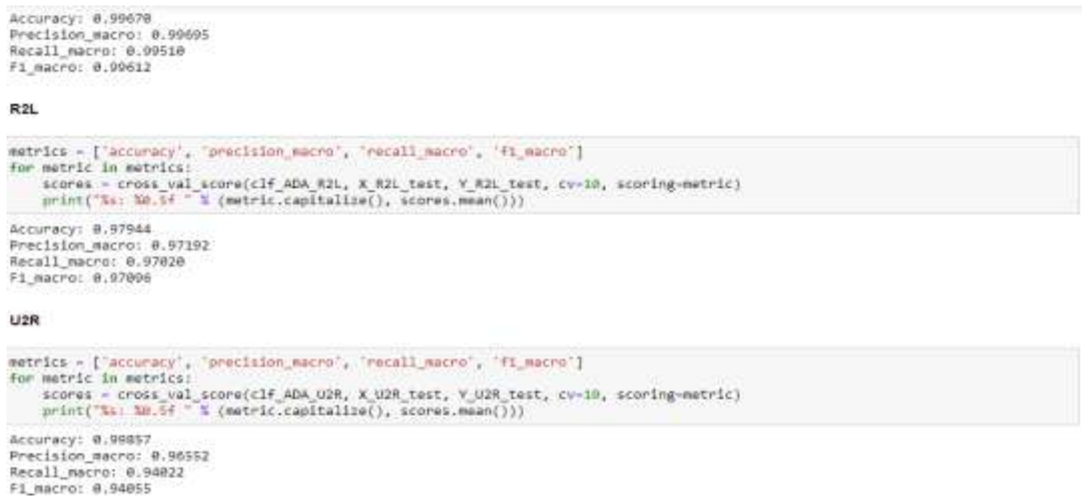
DoS

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_ADA_DoS, X_DoS_test, Y_DoS_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99819
Precision_macro: 0.99820
Recall_macro: 0.99806
F1_macro: 0.99781
```

Probe

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_ADA_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```



```
Accuracy: 0.99670
Precision_macro: 0.99695
Recall_macro: 0.99510
F1_macro: 0.99612
```

R2L

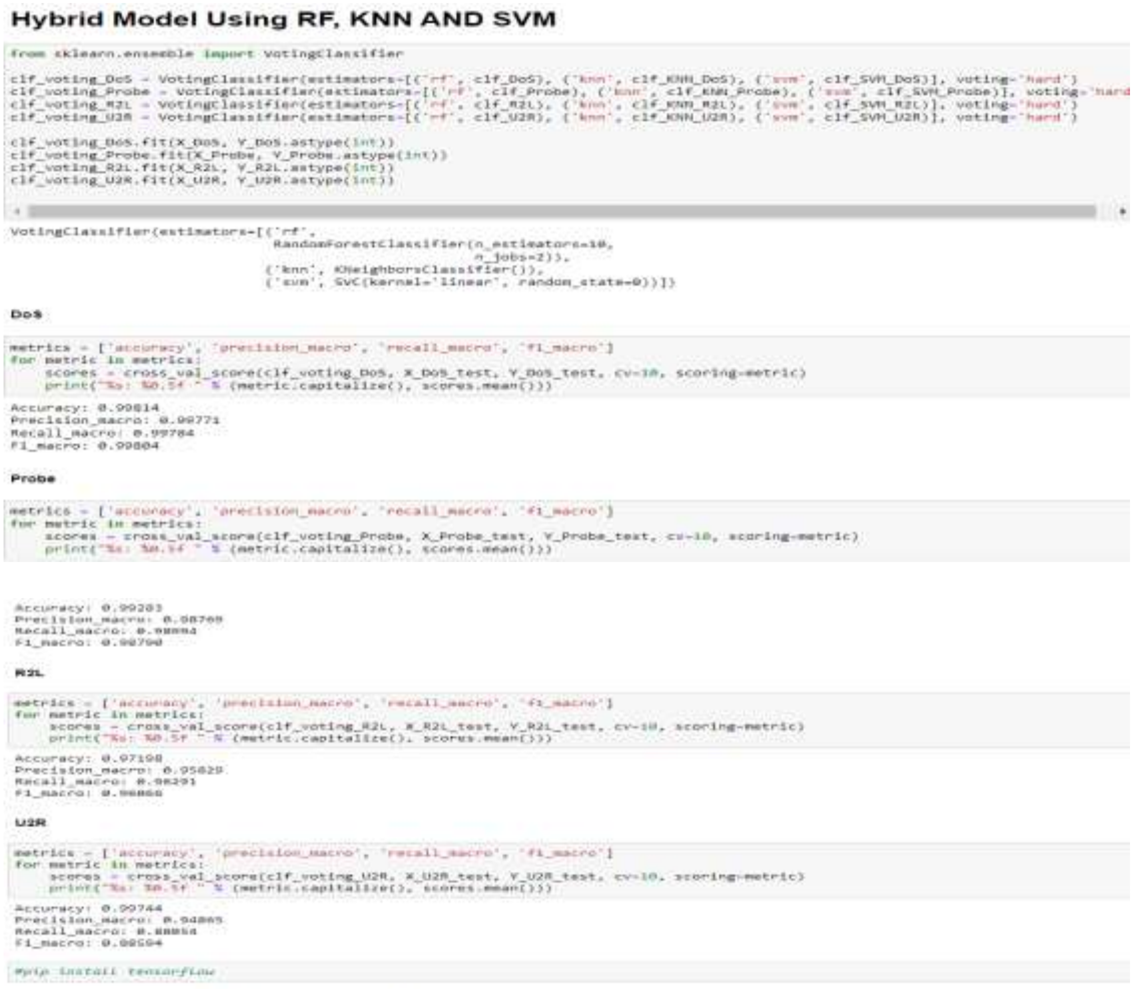
```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_ADA_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.97944
Precision_macro: 0.97192
Recall_macro: 0.97020
F1_macro: 0.97096
```

U2R

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_ADA_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99857
Precision_macro: 0.96552
Recall_macro: 0.94022
F1_macro: 0.94055
```



Hybrid Model Using RF, KNN AND SVM

```
from sklearn.ensemble import VotingClassifier

clf_voting_DoS = VotingClassifier(estimators=[('rf', clf_DoS), ('knn', clf_KNN_DoS), ('svm', clf_SVM_DoS)], voting='hard')
clf_voting_Probe = VotingClassifier(estimators=[('rf', clf_Probe), ('knn', clf_KNN_Probe), ('svm', clf_SVM_Probe)], voting='hard
clf_voting_R2L = VotingClassifier(estimators=[('rf', clf_R2L), ('knn', clf_KNN_R2L), ('svm', clf_SVM_R2L)], voting='hard')
clf_voting_U2R = VotingClassifier(estimators=[('rf', clf_U2R), ('knn', clf_KNN_U2R), ('svm', clf_SVM_U2R)], voting='hard')

clf_voting_DoS.fit(X_DoS, Y_DoS.astype(int))
clf_voting_Probe.fit(X_Probe, Y_Probe.astype(int))
clf_voting_R2L.fit(X_R2L, Y_R2L.astype(int))
clf_voting_U2R.fit(X_U2R, Y_U2R.astype(int))
```

```
VotingClassifier(estimators=[('rf',
                              RandomForestClassifier(n_estimators=10,
                                                     n_jobs=2)),
                             ('knn', KNeighborsClassifier()),
                             ('svm', SVC(kernel='linear', random_state=0))])
```

DoS

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_voting_DoS, X_DoS_test, Y_DoS_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99814
Precision_macro: 0.99771
Recall_macro: 0.99784
F1_macro: 0.99804
```

Probe

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_voting_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99283
Precision_macro: 0.98769
Recall_macro: 0.98884
F1_macro: 0.98790
```

R2L

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_voting_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

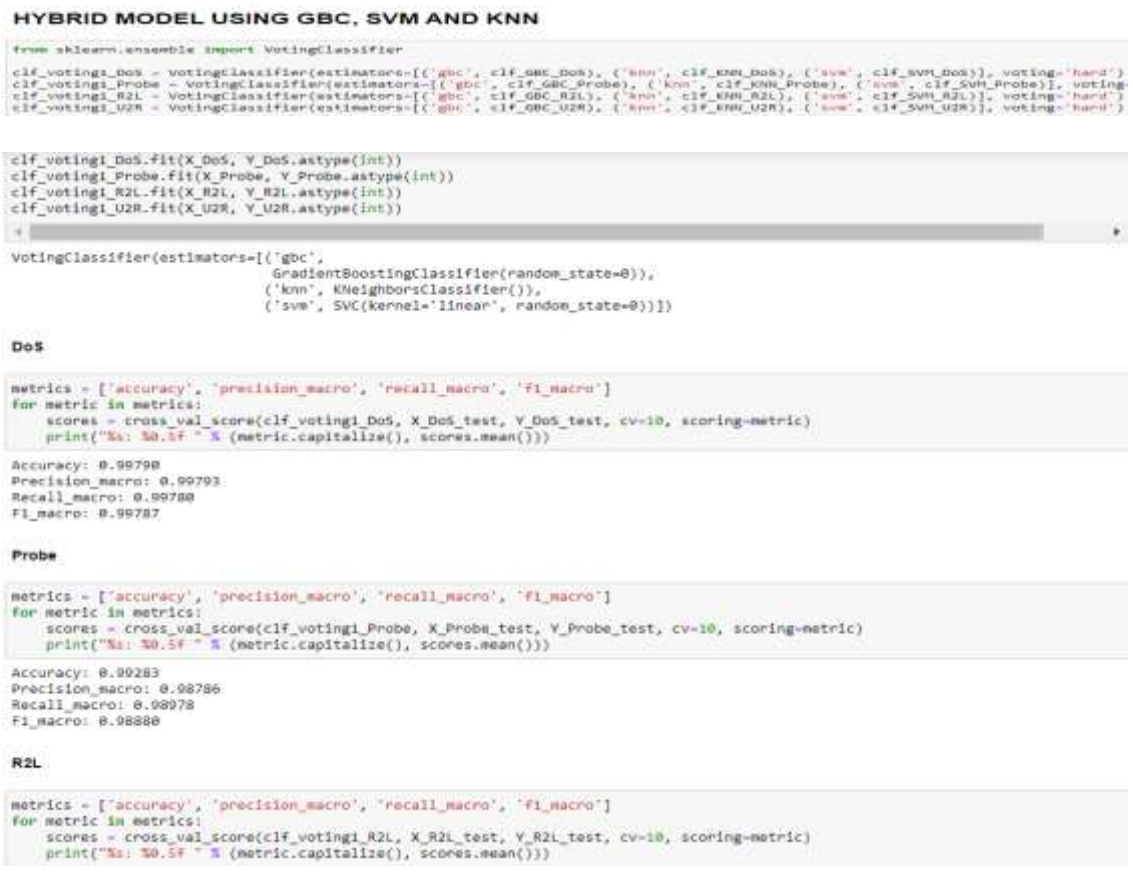
```
Accuracy: 0.97198
Precision_macro: 0.95829
Recall_macro: 0.98291
F1_macro: 0.96868
```

U2R

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_voting_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99744
Precision_macro: 0.94865
Recall_macro: 0.88858
F1_macro: 0.89594
```

```
#pip install tensorflow
```



HYBRID MODEL USING GBC, SVM AND KNN

```
from sklearn.ensemble import VotingClassifier

clf_voting1_DoS = VotingClassifier(estimators=[('gbc', clf_GBC_DoS), ('knn', clf_KNN_DoS), ('svm', clf_SVM_DoS)], voting='hard')
clf_voting1_Probe = VotingClassifier(estimators=[('gbc', clf_GBC_Probe), ('knn', clf_KNN_Probe), ('svm', clf_SVM_Probe)], voting
clf_voting1_R2L = VotingClassifier(estimators=[('gbc', clf_GBC_R2L), ('knn', clf_KNN_R2L), ('svm', clf_SVM_R2L)], voting='hard')
clf_voting1_U2R = VotingClassifier(estimators=[('gbc', clf_GBC_U2R), ('knn', clf_KNN_U2R), ('svm', clf_SVM_U2R)], voting='hard')
```

```
clf_voting1_DoS.fit(X_DoS, Y_DoS.astype(int))
clf_voting1_Probe.fit(X_Probe, Y_Probe.astype(int))
clf_voting1_R2L.fit(X_R2L, Y_R2L.astype(int))
clf_voting1_U2R.fit(X_U2R, Y_U2R.astype(int))
```

```
VotingClassifier(estimators=[('gbc',
                              GradientBoostingClassifier(random_state=0)),
                             ('knn', KNeighborsClassifier()),
                             ('svm', SVC(kernel='linear', random_state=0))])
```

DoS

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_voting1_DoS, X_DoS_test, Y_DoS_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99790
Precision_macro: 0.99793
Recall_macro: 0.99780
F1_macro: 0.99787
```

Probe

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_voting1_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99283
Precision_macro: 0.98786
Recall_macro: 0.98978
F1_macro: 0.98880
```

R2L

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_voting1_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.97325
Precision_macro: 0.95821
Recall_macro: 0.96706
F1_macro: 0.96249
```

U2R

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_voting1_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99734
Precision_macro: 0.93833
Recall_macro: 0.87335
F1_macro: 0.89419
```

HYBRID MODEL USING RF, ANN AND ADABOOST

```
from sklearn.ensemble import VotingClassifier

clf_voting2_DoS = VotingClassifier(estimators=[('rf', clf_DoS), ('ann', clf_ANN_DoS), ('ada', clf_ADA_DoS)], voting='hard')
clf_voting2_Probe = VotingClassifier(estimators=[('rf', clf_Probe), ('ann', clf_ANN_Probe), ('ada', clf_ADA_Probe)], voting='hard
clf_voting2_R2L = VotingClassifier(estimators=[('rf', clf_R2L), ('ann', clf_ANN_R2L), ('ada', clf_ADA_R2L)], voting='hard')
clf_voting2_U2R = VotingClassifier(estimators=[('rf', clf_U2R), ('ann', clf_ANN_U2R), ('ada', clf_ADA_U2R)], voting='hard')

clf_voting2_DoS.fit(X_DoS, Y_DoS.astype(int))
clf_voting2_Probe.fit(X_Probe, Y_Probe.astype(int))
clf_voting2_R2L.fit(X_R2L, Y_R2L.astype(int))
clf_voting2_U2R.fit(X_U2R, Y_U2R.astype(int))
```

```
VotingClassifier(estimators=[('rf',
                              RandomForestClassifier(n_estimators=10,
                                                     n_jobs=2)),
                             ('ann',
                              MLPClassifier(hidden_layer_sizes=(100, 50),
                                            max_iter=500, random_state=42)),
                             ('ada',
                              AdaBoostClassifier(base_estimator=DecisionTreeClassifier(criterion='entropy',
                                                                                       max_depth=4)))])
```

DoS

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_voting2_DoS, X_DoS_test, Y_DoS_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99825
Precision_macro: 0.99846
Recall_macro: 0.99830
F1_macro: 0.99804
```

Probe

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_voting2_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99679
Precision_macro: 0.99711
Recall_macro: 0.99417
F1_macro: 0.99495
```

R2L

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_voting2_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.98095
Precision_macro: 0.97385
Recall_macro: 0.97315
F1_macro: 0.97405
```

U2R

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_voting2_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99816
Precision_macro: 0.96188
Recall_macro: 0.90337
F1 macro: 0.93690
```

Hybrid model USING ANN, ADA and GBC

```
from sklearn.ensemble import VotingClassifier

clf_voting3_DoS = VotingClassifier(estimators=[('GBC', clf_GBC_DoS), ('ann', clf_ANN_DoS), ('ada', clf_ADA_DoS)], voting='hard')
clf_voting3_Probe = VotingClassifier(estimators=[('GBC', clf_GBC_Probe), ('ann', clf_ANN_Probe), ('ada', clf_ADA_Probe)], voting
clf_voting3_R2L = VotingClassifier(estimators=[('GBC', clf_GBC_R2L), ('ann', clf_ANN_R2L), ('ada', clf_ADA_R2L)], voting='hard')
clf_voting3_U2R = VotingClassifier(estimators=[('GBC', clf_GBC_U2R), ('ann', clf_ANN_U2R), ('ada', clf_ADA_U2R)], voting='hard')

clf_voting3_DoS.fit(X_DoS, Y_DoS.astype(int))
clf_voting3_Probe.fit(X_Probe, Y_Probe.astype(int))
clf_voting3_R2L.fit(X_R2L, Y_R2L.astype(int))
clf_voting3_U2R.fit(X_U2R, Y_U2R.astype(int))
```

```
VotingClassifier(estimators=[('GBC',
                              GradientBoostingClassifier(random_state=0)),
                             ('ann',
                              MLPClassifier(hidden_layer_sizes=(100, 50),
                                            max_iter=500, random_state=42)),
                             ('ada',
                              AdaBoostClassifier(base_estimator=DecisionTreeClassifier(criterion='entropy',
                                                                                       max_depth=4)))])
```

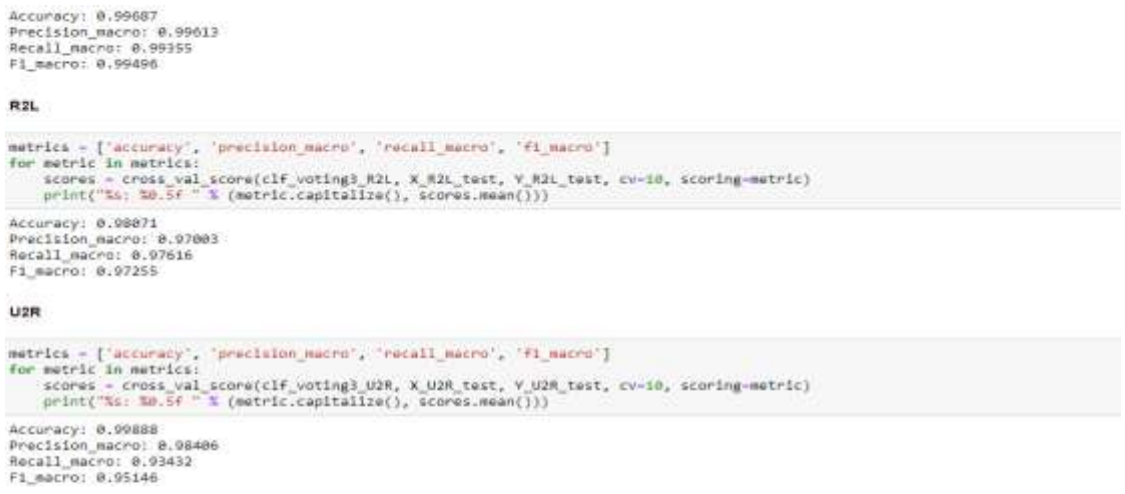
DoS

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_voting3_DoS, X_DoS_test, Y_DoS_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.99837
Precision_macro: 0.99846
Recall_macro: 0.99831
F1_macro: 0.99840
```

Probe

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_voting3_Probe, X_Probe_test, Y_Probe_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```



```
Accuracy: 0.99687
Precision_macro: 0.99613
Recall_macro: 0.99355
F1_macro: 0.99496
```

R2L

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_voting3_R2L, X_R2L_test, Y_R2L_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

```
Accuracy: 0.98071
Precision_macro: 0.97003
Recall_macro: 0.97616
F1_macro: 0.97255
```

U2R

```
metrics = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for metric in metrics:
    scores = cross_val_score(clf_voting3_U2R, X_U2R_test, Y_U2R_test, cv=10, scoring=metric)
    print("%s: %0.5f " % (metric.capitalize(), scores.mean()))
```

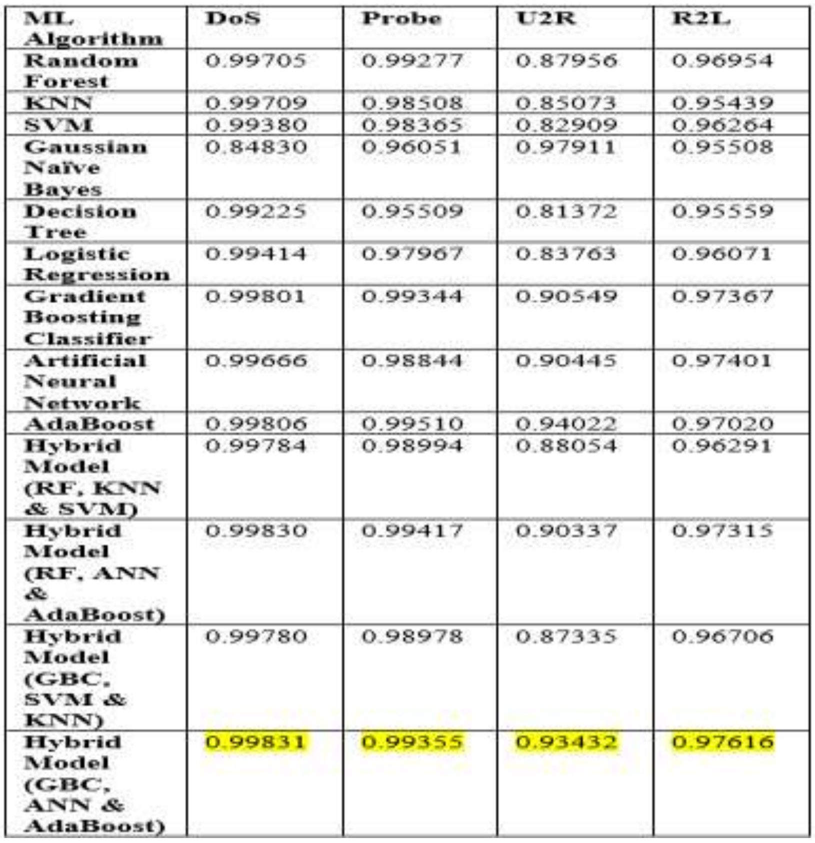
```
Accuracy: 0.99888
Precision_macro: 0.98406
Recall_macro: 0.93432
F1_macro: 0.95146
```

TABLES OF THE RESULT

Table 1: Accuracy Score

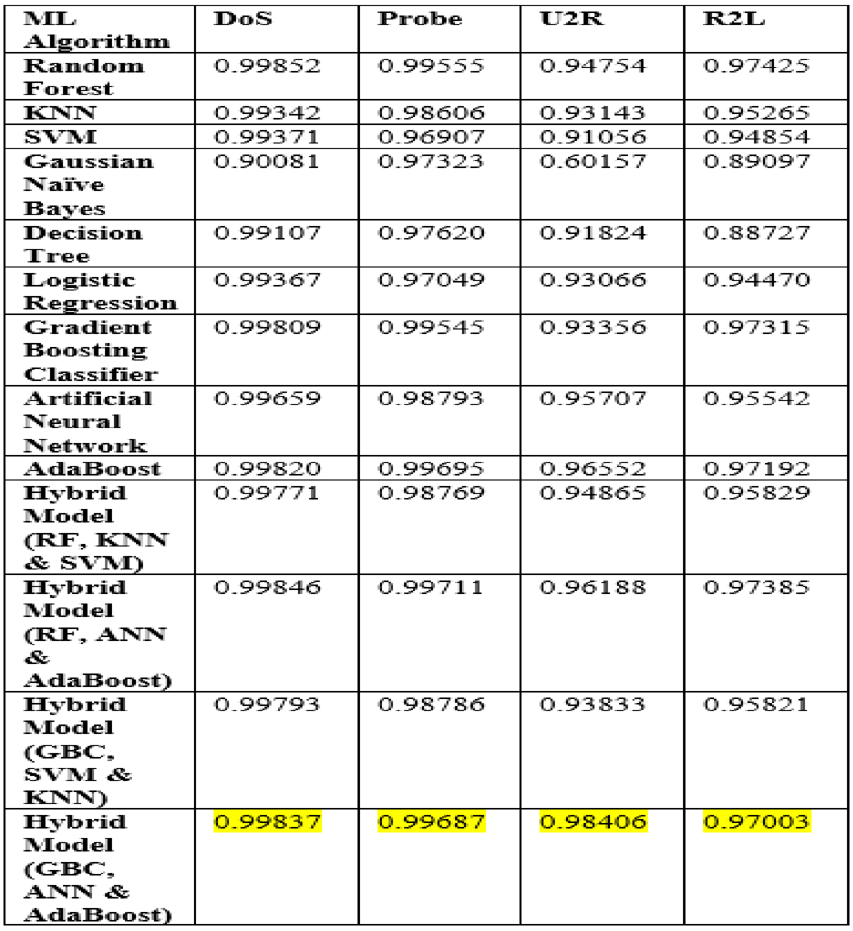
ML Algorithm	**DoS**	**Probe**	**U2R**	**R2L**
Random Forest	0.99785	0.99662	0.99775	0.98079
KNN	0.99715	0.99077	0.99703	0.96705
SVM	0.99371	0.98450	0.99632	0.96793
Gaussian Naïve Bayes	0.86733	0.97898	0.93562	0.97259
Decision Tree	0.99167	0.97824	0.93300	0.99632
Logistic Regression	0.99400	0.98384	0.96570	0.99683
Gradient Boosting Classifier	0.99808	0.99646	0.98118	0.99775
Artificial Neural Network	0.99668	0.99242	0.97420	0.99806
AdaBoost	0.99819	0.99670	0.99857	0.97944
Hybrid Model (RF, KNN & SVM)	0.99814	0.99283	0.99744	0.97198
Hybrid Model (RF, ANN & AdaBoost)	0.99825	0.99679	0.99816	0.98095
Hybrid Model (GBC, SVM & KNN)	0.99790	0.99283	0.99734	0.97325
Hybrid Model (GBC, ANN & AdaBoost)	0.99837	0.99687	0.99888	0.98071

Table 2: Precision Score



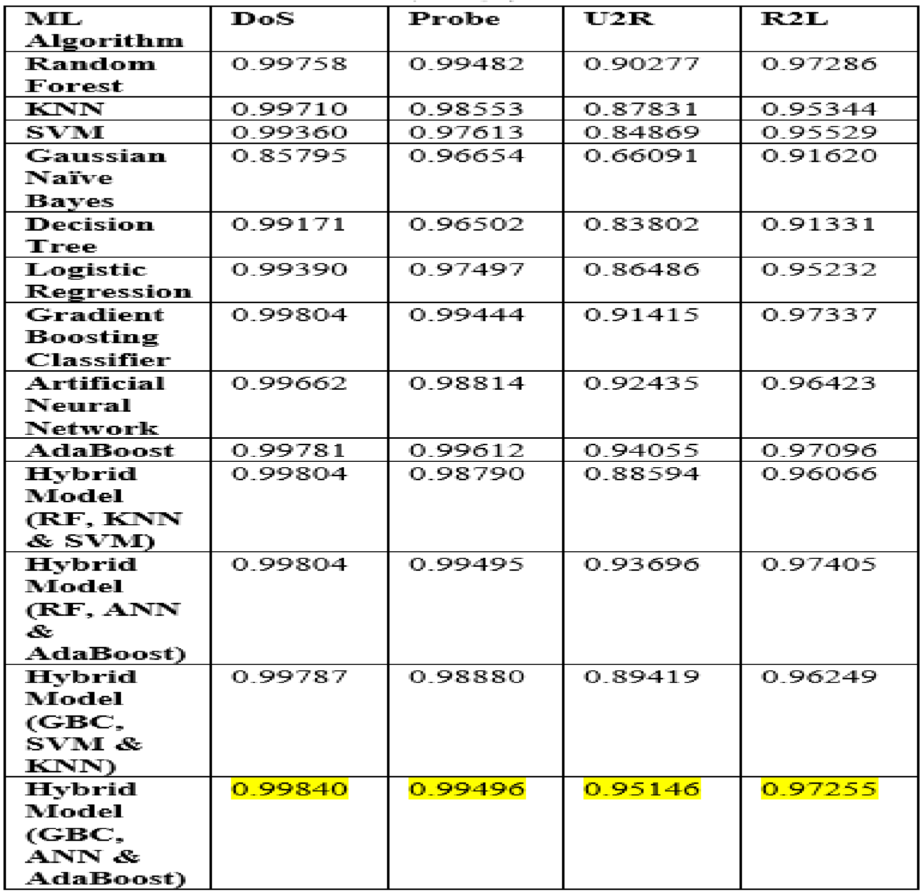
ML Algorithm	DoS	Probe	U2R	R2L
Random Forest	0.99705	0.99277	0.87956	0.96954
KNN	0.99709	0.98508	0.85073	0.95439
SVM	0.99380	0.98365	0.82909	0.96264
Gaussian Naïve Bayes	0.84830	0.96051	0.97911	0.95508
Decision Tree	0.99225	0.95509	0.81372	0.95559
Logistic Regression	0.99414	0.97967	0.83763	0.96071
Gradient Boosting Classifier	0.99801	0.99344	0.90549	0.97367
Artificial Neural Network	0.99666	0.98844	0.90445	0.97401
AdaBoost	0.99806	0.99510	0.94022	0.97020
Hybrid Model (RF, KNN & SVM)	0.99784	0.98994	0.88054	0.96291
Hybrid Model (RF, ANN & AdaBoost)	0.99830	0.99417	0.90337	0.97315
Hybrid Model (GBC, SVM & KNN)	0.99780	0.98978	0.87335	0.96706
Hybrid Model (GBC, ANN & AdaBoost)	0.99831	0.99355	0.93432	0.97616

Table 3: Recall Rate



ML Algorithm	DoS	Probe	U2R	R2L
Random Forest	0.99852	0.99555	0.94754	0.97425
KNN	0.99342	0.98606	0.93143	0.95265
SVM	0.99371	0.96907	0.91056	0.94854
Gaussian Naïve Bayes	0.90081	0.97323	0.60157	0.89097
Decision Tree	0.99107	0.97620	0.91824	0.88727
Logistic Regression	0.99367	0.97049	0.93066	0.94470
Gradient Boosting Classifier	0.99809	0.99545	0.93356	0.97315
Artificial Neural Network	0.99659	0.98793	0.95707	0.95542
AdaBoost	0.99820	0.99695	0.96552	0.97192
Hybrid Model (RF, KNN & SVM)	0.99771	0.98769	0.94865	0.95829
Hybrid Model (RF, ANN & AdaBoost)	0.99846	0.99711	0.96188	0.97385
Hybrid Model (GBC, SVM & KNN)	0.99793	0.98786	0.93833	0.95821
Hybrid Model (GBC, ANN & AdaBoost)	0.99837	0.99687	0.98406	0.97003

Table 4: F1 Score



ML Algorithm	DoS	Probe	U2R	R2L
Random Forest	0.99758	0.99482	0.90277	0.97286
KNN	0.99710	0.98553	0.87831	0.95344
SVM	0.99360	0.97613	0.84869	0.95529
Gaussian Naïve Bayes	0.85795	0.96654	0.66091	0.91620
Decision Tree	0.99171	0.96502	0.83802	0.91331
Logistic Regression	0.99390	0.97497	0.86486	0.95232
Gradient Boosting Classifier	0.99804	0.99444	0.91415	0.97337
Artificial Neural Network	0.99662	0.98814	0.92435	0.96423
AdaBoost	0.99781	0.99612	0.94055	0.97096
Hybrid Model (RF, KNN & SVM)	0.99804	0.98790	0.88594	0.96066
Hybrid Model (RF, ANN & AdaBoost)	0.99804	0.99495	0.93696	0.97405
Hybrid Model (GBC, SVM & KNN)	0.99787	0.98880	0.89419	0.96249
Hybrid Model (GBC, ANN & AdaBoost)	0.99840	0.99496	0.95146	0.97255

BIBLIOGRAPHY

1. Ghorbani, A. A., Lu, W., & Tavallaee, M. (2009). Network intrusion detection and prevention: concepts and techniques (Vol. 47). Springer Science & Business Media.

2. Levin, I. (2000). KDD-99 classifier learning contest LLSoft's results overview. ACM SIGKDD Explorations Newsletter, 1(2), 67-75.

3. Chebrolu, S., Abraham, A., & Thomas, J. P. (2005). Feature deduction and ensemble design of intrusion detection systems. Computers & security, 24(4), 295-307.

4. Mukkamala, S., Janoski, G., & Sung, A. (2002, May). Intrusion detection using neural networks and support vector machines. In Proceedings of the 2002 International Joint Conference on Neural Networks. IJCNN'02 (Cat. No. 02CH37290) (Vol. 2, pp. 1702-1707). IEEE.

5. Heckerman, D., Geiger, D., & Chickering, D. M. (1995). Learning Bayesian networks: The combination of knowledge and statistical data. Machine learning, 20, 197-243.

6. Amor, N. B., Benferhat, S., & Elouedi, Z. (2004, March). Naive bayes vs decision trees in intrusion detection systems. In Proceedings of the 2004 ACM symposium on Applied computing (pp. 420-424).

7. Vinayakumar, R., Soman, K. P., & Poornachandran, P. (2017, September). Applying convolutional neural network for network intrusion detection. In 2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI) (pp. 1222-1228). IEEE.

8. Mill, J., & Inoue, A. (2004, July). Support vector classifiers and network intrusion detection. In 2004 IEEE International Conference on Fuzzy Systems (IEEE Cat. No. 04CH37542) (Vol. 1, pp. 407-410). IEEE.

9. Wang, J., Yang, Q., & Ren, D. (2009, July). An intrusion detection algorithm based on decision tree technology. In 2009 Asia-Pacific Conference on Information Processing (Vol. 2, pp. 333-335). IEEE.

10. Farid, D. M., Harbi, N., & Rahman, M. Z. (2010). Combining naive bayes and decision tree for adaptive intrusion detection. arXiv preprint arXiv:1005.4496.

11. Sathya, S. S., Ramani, R. G., & Sivaselvi, K. (2011). Discriminant analysis based feature selection in kdd intrusion dataset. International Journal of computer applications, 31(11), 1-7.

12. Panda, M., & Patra, M. R. (2009). A Hybrid clustering approach for network intrusion detection using cobweb and FFT. Journal of Intelligent systems, 18(3), 229-246.

13. Bhavani, T. T., Rao, M. K., & Reddy, A. M. (2019, November). Network intrusion detection system using random forest and decision tree machine learning techniques. In First International Conference on Sustainable Technologies for Computational Intelligence: Proceedings of ICTSCI 2019 (pp. 637-643). Singapore: Springer Singapore.

14. Rao, B. B., & Swathi, K. (2017). Fast kNN classifiers for network intrusion detection system. Indian Journal of Science and Technology, 10(14), 1-10.

15. Jha, J., & Ragha, L. (2013). Intrusion detection system using support vector machine. International Journal of Applied Information Systems (IJAIS), 3, 25-30.

16. Sharmila, B. S., & Nagapadma, R. (2019, November). Intrusion detection system using naive bayes algorithm. In 2019 IEEE International WIE Conference on Electrical and Computer Engineering (WIECON-ECE) (pp. 1-4). IEEE.

17. Kumar, M., Hanumanthappa, M., & Kumar, T. S. (2012, November). Intrusion Detection System using decision tree algorithm. In 2012 IEEE 14th international conference on communication technology (pp. 629-634). IEEE.

18. Besharati, E., Naderan, M., & Namjoo, E. (2019). LR-HIDS: logistic regression host-based intrusion detection system for cloud environments. Journal of Ambient Intelligence and Humanized Computing, 10, 3669-3692.

19. Verma, P., Anwar, S., Khan, S., & Mane, S. B. (2018, July). Network intrusion detection using clustering and gradient boosting. In 2018 9th International conference on computing, communication and networking technologies (ICCCNT) (pp. 1-7). IEEE.

20. Ding, Y., & Zhai, Y. (2018, December). Intrusion detection system for NSL-KDD dataset using convolutional neural networks. In Proceedings of the 2018 2nd International conference on computer science and artificial intelligence (pp. 81-85).

21. Farnaaz, N., & Jabbar, M. A. (2016). Random forest modeling for network intrusion detection system. Procedia Computer Science, 89, 213-217.

22. Wang, Q., & Wei, X. (2020, January). The detection of network intrusion based on improved adaboost algorithm. In Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy (pp. 84-88).