**Report – Intrusion detection System Using Machine Learning**

Acknowledgment

I, Ashish, hereby declare that the work presented in this project titled "Intrusion Detection System Using Machine Learning Techniques" is the result of my own research and efforts conducted during my academic study. This project was undertaken as part of my computer science at Vellore institute of technology.

The ideas, methodologies, and findings presented in this project are solely mine unless otherwise acknowledged. I affirm that this project has not been submitted for any other degree or qualification at any university or institution.

Wherever external sources of information, including scholarly articles, research papers, or online resources, have been used, they have been duly cited and referenced in accordance with academic standards. Any contributions from other individuals or organizations to this project are clearly acknowledged in the appropriate sections.

I understand the importance of academic integrity and ethical conduct in research. I affirm that all data, figures, and results presented in this project are authentic and accurately represent the outcomes of the experiments and analyses conducted by me.

I acknowledge the support and guidance provided by my supervisor, YOGANAND S throughout the course of this project. Their expertise and feedback have been invaluable in shaping the direction and scope of my research.

I also acknowledge the use of resources provided by Google, including access to datasets, tools, and research publications, which have facilitated the development and evaluation of the intrusion detection system described in this project.

I declare that any breach of academic integrity or misconduct, including plagiarism or fabrication of results, is against my ethical principles and the policies of [Your Institution]. I take full responsibility for the originality and integrity of this project.

|  |  |  |
| --- | --- | --- |
| **S.NO** | **Category** | **PAGE** |
| 1. | **Abstract** | 3 |
| 2. | **Introduction** | 3 |
| 3. | **Literary Survey** | 4 |
| 4. | **Methodology** | 5 |
| 5. | **Modelling** | 7 |
| 6. | **Remedial Measure** | 15 |
| 7. | **Conclusion** | 16 |
| 8. | **Reference** | 17 |
| 9. | **Appendix** | 18 |

**Table of Contents**

**Abstract**

Due to the expansion and development of modern networks, the volume and destructiveness of cyber-attacks are continuously increasing. Intrusion Detection Systems (IDSs) are essential techniques for maintaining and enhancing network security. IDS-ML is an open-source code repository written in Python for developing IDSs from public network traffic datasets using traditional and advanced Machine Learning (ML) algorithms. With optimized ML models, the IDSs developed in the repository can identify various types of cyber-attacks to protect modern networks. This code repository can be easily implemented and reproduced on any intrusion detection datasets to solve problems in the cybersecurity field.

**Introduction to IDS and ML**

With the rapid expansion of the Internet and communication technologies, as well as the vast number of applications accessible on the network, network security has become a serious issue that must be addressed. Various cybersecurity mechanisms and protection systems have been introduced to protect modern networks, such as firewalls, authentication techniques, cryptography methods, and Intrusion Detection Systems (IDSs). IDS monitors network traffic in order to identify abnormal activities or malicious cyber-attacks. When suspicious behaviours are detected, an IDS will generate an alarm and reports it to the network administrator. Additionally, corresponding countermeasures will then be taken to defend against the ongoing attack and prevent future attacks.

IDSs can be categorized as signature-based IDSs, anomaly-based IDSs, and hybrid IDSs. The signature-based IDSs are developed to detect known attacks whose patterns or signatures have already been defined in the system. Although signature-based IDSs usually achieve high performance on known attack detection tasks, they are unable to detect new or zero-day attacks since their patterns are unknown. On the other hand, anomaly-based IDSs are designed to detect zero-day attacks by distinguishing unknown attacks from pre-defined normal activities. However, their performance on known attack detection is often lower than the performance of signature-based IDSs. Hybrid IDSs are designed to detect both known and unknown attacks by integrating signature-based IDSs and anomaly-based IDSs.

Machine Learning (ML) techniques have recently become promising solutions for developing IDSs. ML is a collection of techniques that employ mathematical formulae to automatically discover, examine, and extract patterns from data. Extracting and acquiring meaningful information helps ML models make informed judgments and predictions. ML algorithms can be classified as supervised and unsupervised learning algorithms. Supervised learning algorithms are a class of ML algorithms that map input variables to a target variable using labelled data for training, such as K-Nearest Neighbours (KNN), Decision Tree (DT) based models, and Deep Learning (DL) algorithms, etc. Unsupervised learning algorithms are utilized to discover patterns from unlabelled data, such as k-means, Gaussian Mixture Model (GMM), isolation forest, etc. For IDS development, supervised learning algorithms are often used to develop signature-based IDSs by training on labelled network datasets, while unsupervised learning algorithms can be used in anomaly-based IDSs to distinguish outliers from normal data.

**Literary Survey**

Effectively identifying cyberattacks is a critical challenge for network operators and managers, particularly in the rapidly evolving modern networks. To improve intrusion detection accuracy and defend against more attacks, many advanced ML techniques can be used to develop IDSs, including ensemble learning, Transfer Learning (TL), and Hyper-Parameter Optimization (HPO). Ensemble learning techniques are designed to improve model learning performance by integrating the output of multiple single ML algorithms as base models, including voting, bagging, stacking, etc. TL is an advanced technology that transfers pre-trained models on other datasets or tasks to the target data to improve model training efficiency. HPO is the process of automatically tuning the hyperparameters of ML models to obtain optimized ML models with improved performance. In the IDS-ML code repository, three novel IDS frameworks are provided using advanced ML techniques.

* Host-Based IDS (HIDS): A host-based IDS is deployed on a particular endpoint and designed to protect it against internal and external threats. Such an IDS may have the ability to monitor network traffic to and from the machine, observe running processes, and inspect the system’s logs. A host-based IDS’s visibility is limited to its host machine, decreasing the available context for decision-making, but has deep visibility into the host computer’s internals.
* Network-Based IDS (NIDS): A network-based IDS solution is designed to monitor an entire protected network. It has visibility into all traffic flowing through the network and makes determinations based upon packet metadata and contents. This wider viewpoint provides more context and the ability to detect widespread threats; however, these systems lack visibility into the internals of the endpoints that they protect.

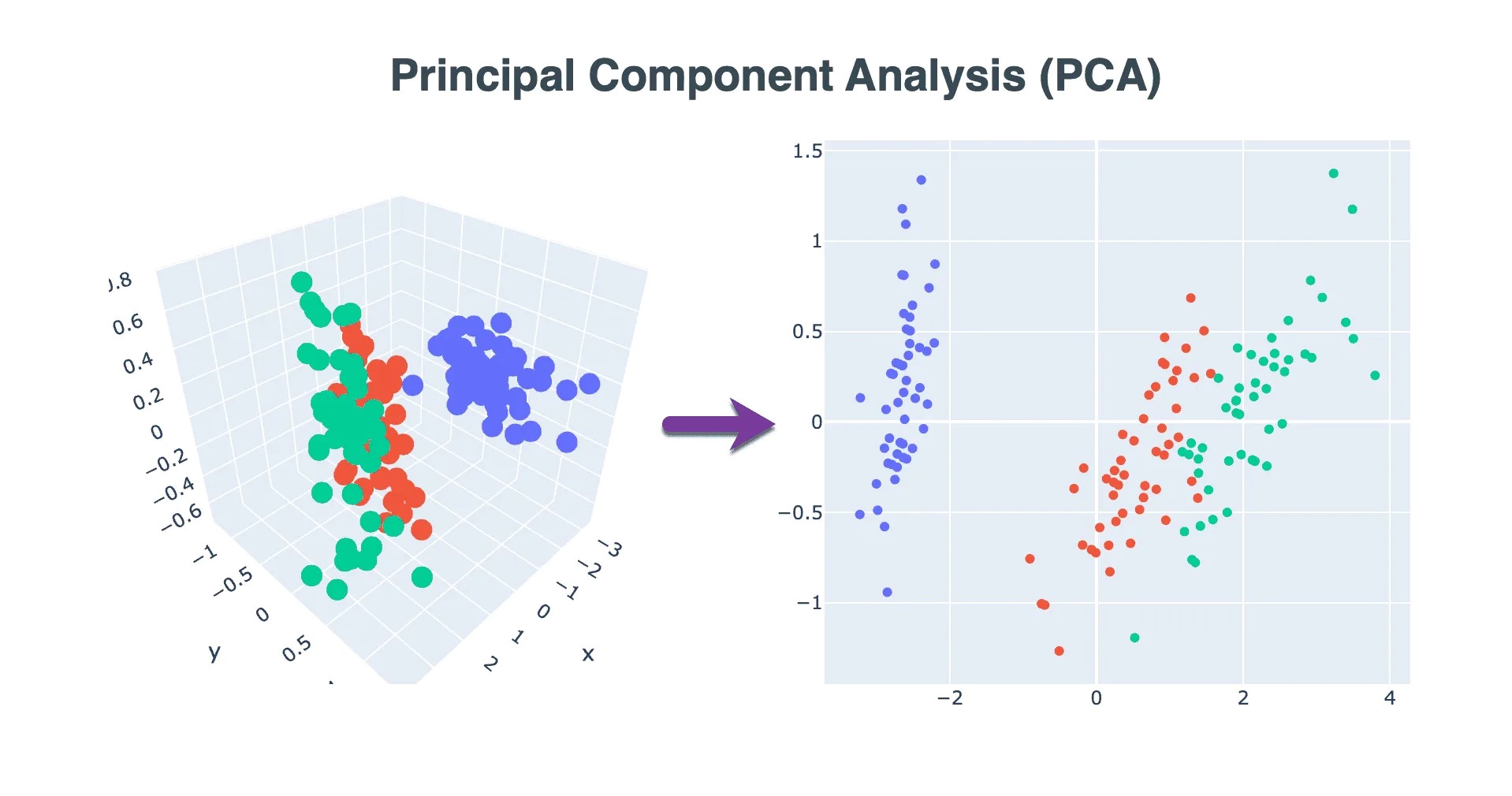
**Detection Method of IDS:**

* Signature-based Method: Signature-based IDS detects the attacks based on the specific patterns such as number of bytes or number of 1’s or number of 0’s in the network traffic. It also detects based on the already known malicious instruction sequence that is used by the malware. The detected patterns in the IDS are known as signatures. Signature-based IDS can easily detect the attacks whose pattern (signature) already exists in system but it is quite difficult to detect the new malware attacks as their pattern (signature) is not known.
* Anomaly-based Method: Anomaly-based IDS was introduced to detect unknown malware attacks as new malware are developed rapidly. In anomaly-based IDS there is use of machine learning to create a trustful activity model and anything coming is compared with that model and it is declared suspicious if it is not found in model. Machine learning-based method has a better-generalized property in comparison to signature-based IDS as these models can be trained according to the applications and hardware configurations.

**Methodology**

Principal Component Analysis

Principal component analysis, or PCA, is a statistical technique to convert high dimensional data to low dimensional data by selecting the most important features that capture maximum information about the dataset. The features are selected based on variance that they cause in the output. The feature that causes highest variance is the first principal component. The feature that is responsible for second highest variance is considered the second principal component, and so on. It is important to mention that principal components do not have any correlation with each other.



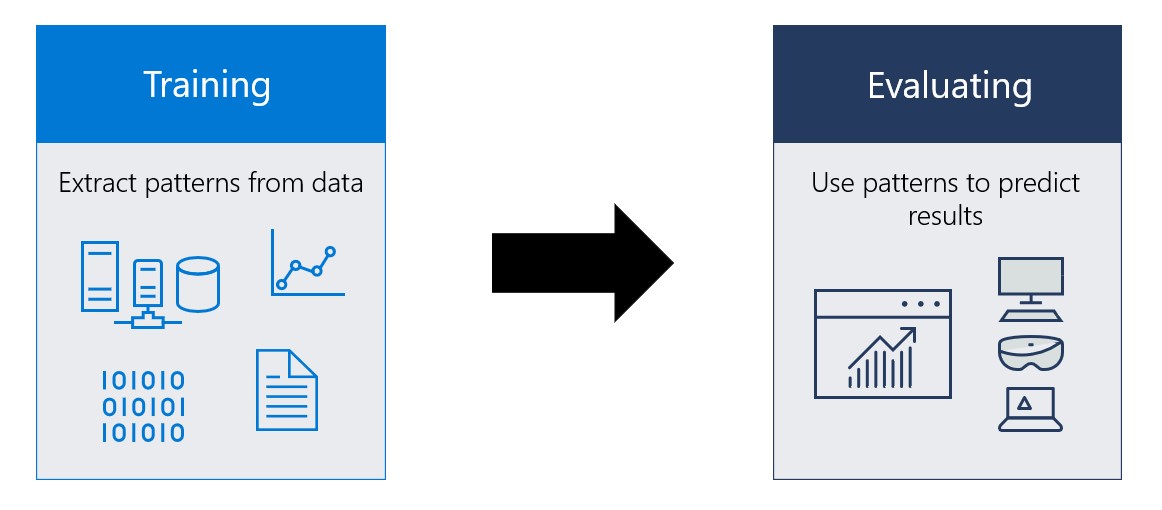
Advantages of PCA

There are two main advantages of dimensionality reduction with PCA.

* The training time of the algorithms reduces significantly with a smaller number of features.
* It is not always possible to analyse data in high dimensions. For instance, if there are 100 features in a dataset. Total number of scatter plots required to visualize the data would be 100(100-1)2 = 4950. Practically it is not possible to analyse data this way.

**Modelling**

The process of modelling means training a machine learning algorithm to predict the labels from the features, tuning it for the business need, and validating it on holdout data. The output from modelling is a trained model that can be used for inference, making predictions on new data points.

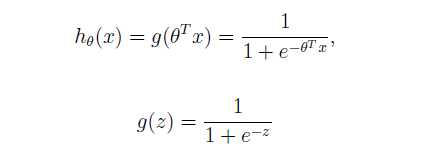


A machine learning model itself is a file that has been trained to recognize certain types of patterns. You train a model over a set of data, providing it an algorithm that it can use to reason over and learn from those data. Once you have trained the model, you can use it to reason over data that it hasn't seen before, and make predictions about those data. For example, let's say you want to build an application that can recognize a user's emotions based on their facial expressions. You can train a model by providing it with images of faces that are each tagged with a certain emotion, and then you can use that model in an application that can recognize any user's emotion.

**Logistic Regression**

[¶](https://www.kaggleusercontent.com/kf/105827768/eyJhbGciOiJkaXIiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0..zzfWcK-8sY1SFS5pJQ5wkQ.4PdWykRgqXYhItYwdgm21guM7Lb4kYatYwtTTM0Aw6wUt9yUBKjqENAP0qBlYpDbVCg5pNsdQGZ08DZo-o-yzfJl4XuXIwOHIt8d5BGz9xh7k2sR08OtLNz-1LsHpIdlnkGVznMEGXh0ptffAQZ2z62vxzai6V0xqaykhyaIUgkz9N5lhQZiwL0eyHKXlvv81AQg905bb3GWWxf0rtAY6e_yYKBAXrzaH9aqXYf7JPwuppCC9eaP7OrFOCEdCO51Ywm-9xxhb9cepABI7DPOpuwCRtVmVfVM1r3Zn1H_pR774mQLSebm5nUreUDenKcijwQtDRYpGr1lcxla_QXeEHsL7tF9WgUxOj2-9OBQaydfBkWEyZMHxMQx_dhShNict0CF3jq2ZUCtxVWvc89hTk3dtQNyrab4GcYAlPtXFDf1QJ4frl4R1EFT8JPEV7xaZv9gpLeoNjTb2IbV2DKnphKxHVmqRpI4oL7uCMaZYBJ7dS0_smFHeVSaZTDku9g1t7Nr2Z7qqbcpj9O7WMwgruk71v5Vi5m2zKBhbxA7LT9qYUV0oomPV88V0FukbFwOrAYcIxO9K3hnS130n_-MCZlSnh7f8Ud-pkKBgjeuQgSSEHaiudo2mx1h6-ew1xRdsV2OiTK8i1B51QoM5Vy4u7HH2VU0IdGg1hHDNAWZgtI.VkMUeuRlzAf87bMxmjPoIA/__resultx__.html?sharingControls=true#Logistic-Regression)

This type of statistical model (also known as logit model) is often used for classification and predictive analytics. Logistic regression estimates the probability of an event occurring, such as voted or did not vote, based on a given dataset of independent variables. Since the outcome is a probability, the dependent variable is bounded between 0 and 1. In logistic regression, a logit transformation is applied on the odds—that is, the probability of success divided by the probability of failure. This is also commonly known as the log odds, or the natural logarithm of odds, and this logistic function is represented by the following formulas:



In this logistic regression equation, h is the dependent or response variable and x is the independent variable. The beta parameter, or coefficient, in this model is commonly estimated via maximum likelihood estimation (MLE). This method tests different values of beta through multiple iterations to optimize for the best fit of log odds. All these iterations produce the log likelihood function, and logistic regression seeks to maximize this function to find the best parameter estimate. Once the optimal coefficient (or coefficients if there is more than one independent variable) is found, the conditional probabilities for each observation can be calculated, logged, and summed together to yield a predicted probability. For binary classification, a probability less than .5 will predict 0 while a probability greater than 0 will predict 1. After the model has been computed, it is best practice to evaluate the how well the model predicts the dependent variable, which is called goodness of fit.

**Binary logistic regression:**

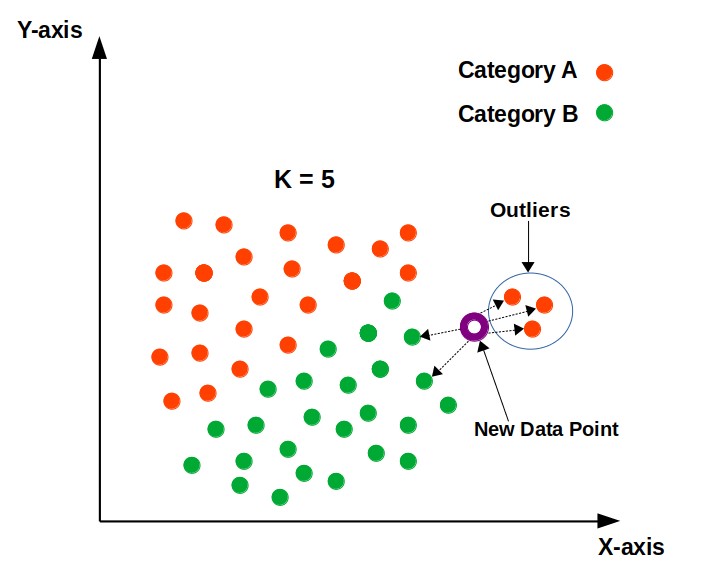
In this approach, the response or dependent variable is dichotomous in nature—i.e. it has only two possible outcomes (e.g. 0 or 1). Some popular examples of its use include predicting if an e-mail is spam or not spam or if a tumour is malignant or not malignant. Within logistic regression, this is the most used approach, and more generally, it is one of the most common classifiers for binary classification.

**Multinomial logistic regression:**

In this type of logistic regression model, the dependent variable has three or more possible outcomes; however, these values have no specified order. For example, movie studios want to predict what genre of film a moviegoer is likely to see to market films more effectively. A multinomial logistic regression model can help the studio to determine the strength of influence a person's age, gender, and dating status may have on the type of film that they prefer. The studio can then orient an advertising campaign of a specific movie toward a group of people likely to go see it.

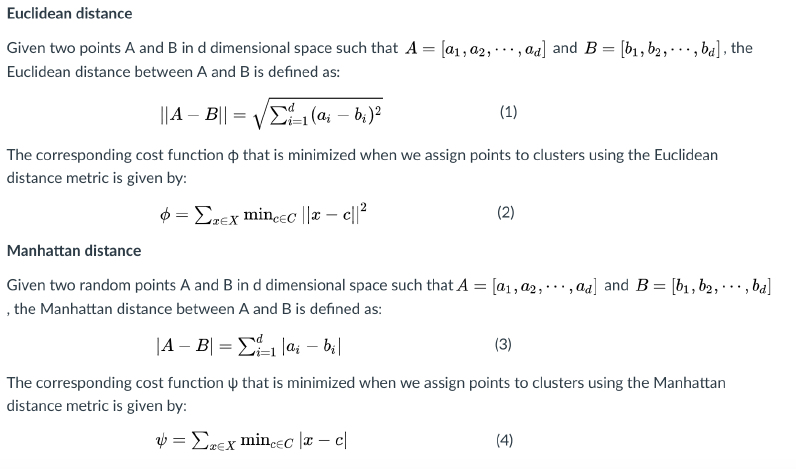
**K-Nearest Neighbours**

The k-nearest neighbors algorithm, also known as KNN or k-NN, is a non-parametric, supervised learning classifier, which uses proximity to make classifications or predictions about the grouping of an individual data point. While it can be used for either regression or classification problems, it is typically used as a classification algorithm, working off the assumption that similar points can be found near one another.



### **Determine your distance metrics**

In order to determine which data points are closest to a given query point, the distance between the query point and the other data points will need to be calculated. These distance metrics help to form decision boundaries, which partitions query points into different regions. You commonly will see decision boundaries visualized with Voronoi diagram.



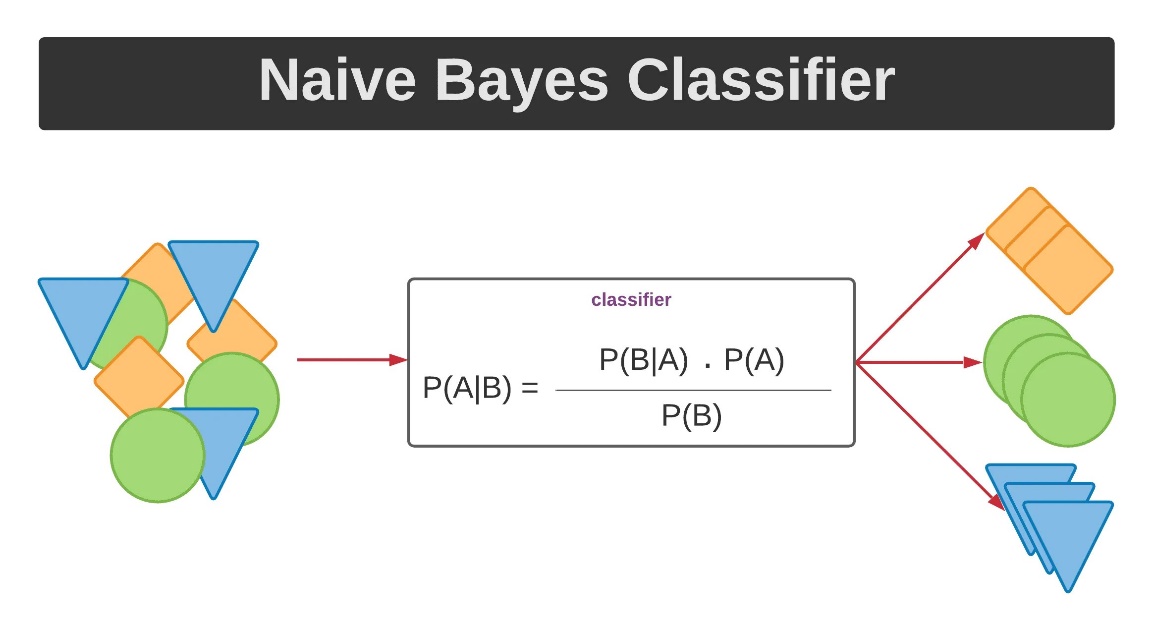
**Naive Bayes**

Naive Bayes classifiers are a collection of classification algorithms based on Bayes’ Theorem. It is not a single algorithm but a family of algorithms where all of them share a common principle. Every pair of features being classified is independent of each other. The assumptions made by Naive Bayes are not generally correct in real-world situations. In-fact, the independence assumption is never correct but often works well in practice.

Now, it is important to know about Bayes’ theorem.

Bayes’ Theorem

Bayes’ Theorem finds the probability of an event occurring given the probability of another event that has already occurred. Bayes’ theorem is stated mathematically as the following equation:



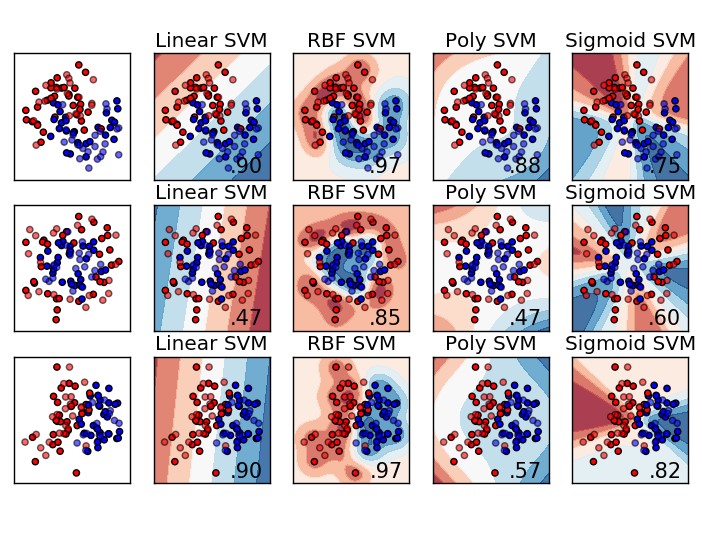
where A and B are events and P(B) ≠ 0.

* Basically, we are trying to find probability of event A, given the event B is true. Event B is also termed as evidence.
* P(A) is the priori of A (the prior probability, i.e. Probability of event before evidence is seen). The evidence is an attribute value of an unknown instance (here, it is event B).
* P(A|B) is a posteriori probability of B, i.e. probability of event after evidence is seen.

**Support Vector Machines**

Support Vector Machine (SVM) is a relatively simple Supervised Machine Learning Algorithm used for classification and/or regression. It is more preferred for classification but is sometimes very useful for regression as well. Basically, SVM finds a hyper-plane that creates a boundary between the types of data. In 2-dimensional space, this hyper-plane is nothing but a line. In SVM, we plot each data item in the dataset in an N-dimensional space, where N is the number of features/attributes in the data. Next, find the optimal hyperplane to separate the data. So by this, you must have understood that inherently, SVM can only perform binary classification (i.e., choose between two classes). However, there are various techniques to use for multi-class problems. Support Vector Machine for Multi-Class Problems To perform SVM on multi-class problems, we can create a binary classifier for each class of the data. The two results of each classifier will be:

* The data point belongs to that class OR
* The data point does not belong to that class.



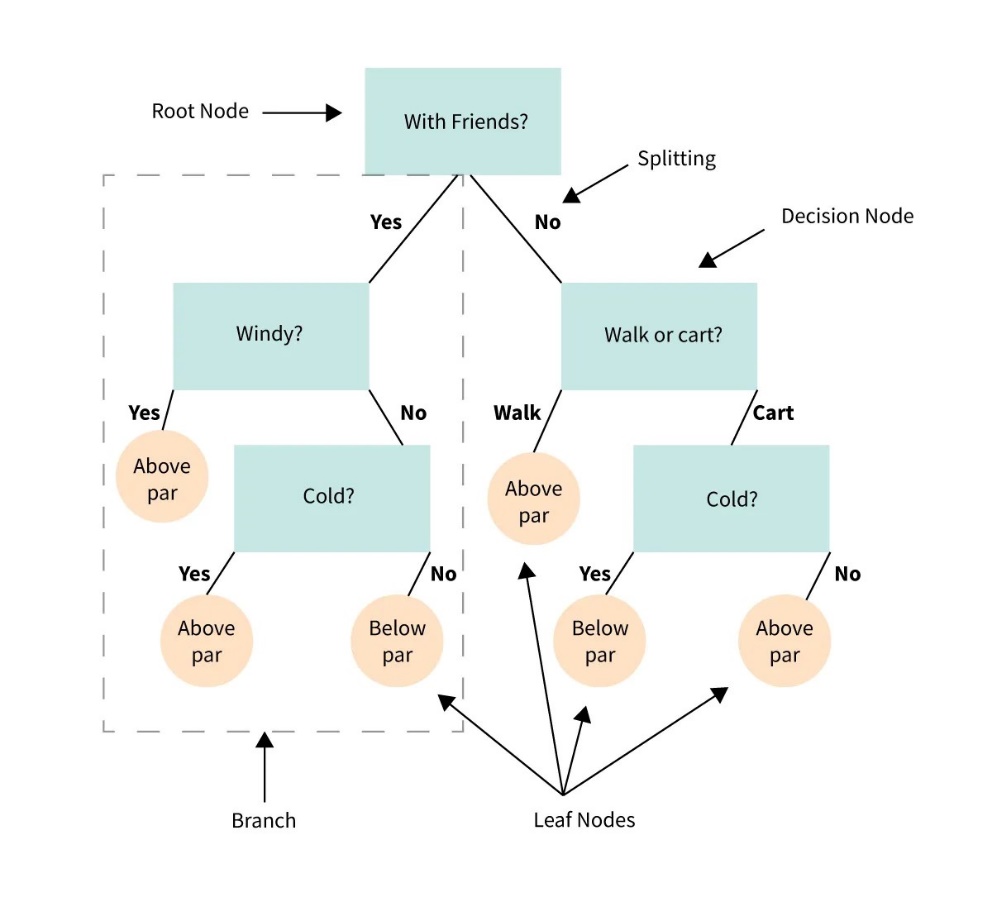
For example, in a class of fruits, to perform multi-class classification, we can create a binary classifier for each fruit. For say, the ‘mango’ class, there will be a binary classifier to predict if it IS a mango OR is NOT a mango. The classifier with the highest score is chosen as the output of the SVM. SVM for complex (Non-Linearly Separable) SVM works very well without any modifications for linearly separable data. Linearly Separable Data is any data that can be plotted in a graph and can be separated into classes using a straight line.

We use Kernelized SVM for non-linearly separable data. Say, we have some non-linearly separable data in one dimension. We can transform this data into two dimensions and the data will become linearly separable in two dimensions. This is done by mapping each 1-D data point to a corresponding 2-D ordered pair. So, for any non-linearly separable data in any dimension, we can just map the data to a higher dimension and then make it linearly separable. This is a very powerful and general transformation. A kernel is nothing but a measure of similarity between data points. The kernel function in a kernelized SVM tells you, that given two data points in the original feature space, what the similarity is between the points in the newly transformed feature space. There are various kernel functions available, but two are very popular:

* Radial Basis Function Kernel (RBF): The similarity between two points in the transformed feature space is an exponentially decaying function of the distance between the vectors and the original input space as shown below. RBF is the default kernel used in SVM.
* Polynomial Kernel: The Polynomial kernel takes an additional parameter, ‘degree’ that controls the model’s complexity and computational cost of the transformation

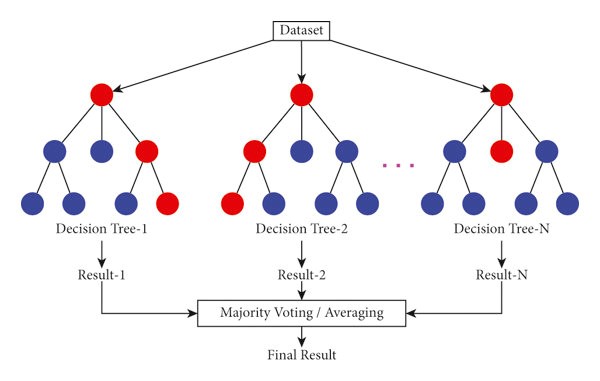
**Decision Tree**

Decision Tree is the most powerful and popular tool for classification and prediction. A Decision tree is a flowchart-like tree structure, where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (terminal node) holds a class label.



A tree can be “learned” by splitting the source set into subsets based on an attribute value test. This process is repeated on each derived subset in a recursive manner called recursive partitioning. The recursion is completed when the subset at a node all has the same value of the target variable, or when splitting no longer adds value to the predictions. Decision trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance. An instance is classified by starting at the root node of the tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute as shown in the above figure. This process is then repeated for the subtree rooted at the new node. The decision tree in above figure classifies a particular morning according to whether it is suitable for playing tennis and returning the classification associated with the leaf. (in this case Yes or No).

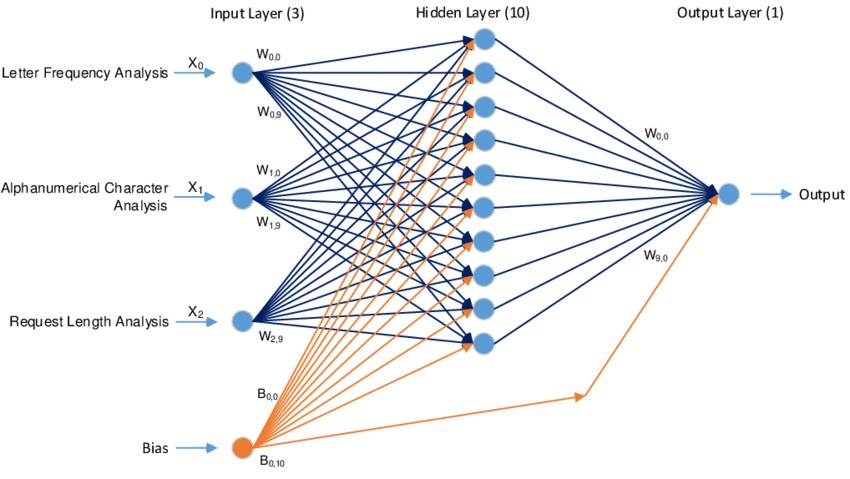
**Random forest**



Random forest is a supervised learning algorithm. The “forest” it builds is an ensemble of decision trees, usually trained with the “bagging” method. The general idea of the bagging method is that a combination of learning models increases the overall result. One big advantage of random forest is that it can be used for both classification and regression problems, which form the majority of current machine learning systems. and It also resists overfitting found in decision trees.

**Neural networks**

Neural networks, also known as artificial neural networks (ANNs) or simulated neural networks (SNNs), are a subset of machine learning and are at the heart of deep learning algorithms. Their name and structure are inspired by the human brain, mimicking the way that biological neurons signal to one another. Artificial neural networks (ANNs) are comprised of a node layers, containing an input layer, one or more hidden layers, and an output layer. Each node, or artificial neuron, connects to another and has an associated weight and threshold. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network. Otherwise, no data is passed along to the next layer of the network.



Neural networks rely on training data to learn and improve their accuracy over time. However, once these learning algorithms are fine-tuned for accuracy, they are powerful tools in computer science and artificial intelligence, allowing us to classify and cluster data at a high velocity. Tasks in speech recognition or image recognition can take minutes versus hours when compared to the manual identification by human experts. One of the most well-known neural networks is Google’s search algorithm.

**Remedial Measure**

Remedial methods for intrusion detection systems (IDS) are strategies and techniques employed to respond to detected intrusions or security breaches. These methods aim to mitigate the impact of intrusions, prevent further damage, and restore the integrity and security of the system. Here are some common remedial methods used in IDS:

1. **Alert Notification**: IDS systems generate alerts when suspicious or malicious activity is detected. These alerts should be promptly reviewed by security personnel to assess the severity of the intrusion and initiate appropriate remedial actions.
2. **Incident Response Plan**: Organizations should have a well-defined incident response plan in place to guide actions in the event of a security breach. This plan outlines roles and responsibilities, escalation procedures, communication protocols, and steps for containing and mitigating the intrusion.
3. **Quarantine and Isolation**: When an intrusion is detected, affected systems or networks may be quarantined or isolated to prevent the spread of malware or unauthorized access. This helps contain the intrusion and minimize further damage to the infrastructure.
4. **Patch Management**: Vulnerabilities discovered during intrusion detection should be addressed promptly through patch management processes. Software patches and updates should be applied to fix known vulnerabilities and reduce the risk of future intrusions exploiting the same weaknesses.
5. **Network Segmentation**: Segregating network resources into separate segments or zones can limit the impact of intrusions by containing them within specific areas of the network. Network segmentation helps prevent lateral movement by attackers and reduces the scope of potential damage.
6. **User Account Management**: In response to intrusions involving compromised user accounts, organizations should implement user account management procedures. This may involve resetting passwords, disabling compromised accounts, and conducting user awareness training to prevent account misuse.
7. **Data Backup and Recovery**: Regular data backups are essential for restoring systems and data in the event of a successful intrusion or data breach. Organizations should maintain comprehensive backup procedures and verify the integrity of backups to ensure reliable recovery in case of emergencies.
8. **Forensic Analysis**: Conducting forensic analysis of intrusion incidents helps determine the cause, extent, and impact of the intrusion. Forensic techniques such as log analysis, memory forensics, and disk imaging can provide valuable insights for investigation and attribution.
9. **Continuous Monitoring and Retrospective Analysis**: IDS systems should continuously monitor network traffic and system activity to detect intrusions in real-time. Additionally, retrospective analysis of historical data can uncover previously undetected intrusions and identify trends or patterns indicative of malicious activity.
10. **Security Awareness Training**: Educating employees and users about security best practices and common attack vectors can help prevent future intrusions. Security awareness training should cover topics such as phishing awareness, password hygiene, and safe browsing habits.

**Top of Form**

**Conclusion**

Cyber-attacks are becoming more damaging and sophisticated. Detecting different types of attacks and understanding their patterns are crucial procedures in network security frameworks. The IDS-ML code repository provides easy-to-use IDS frameworks to apply traditional and advanced ML techniques to the state-of-the-art network traffic dataset for intrusion detection in modern networks. Network and cybersecurity researchers can take advantage of this code due to its easy implementation and clear explanation.

This research project can be extended and improved in two primary research directions. Firstly, the zero-day attack detection performance still has much room for improvement, as it is still an unsolved issue. Advanced unsupervised anomaly detection techniques and online adaptive approaches, such as Extreme Gradient Boosting Outlier Detection (XGBOD) and Performance Weighted Probability Averaging Ensemble (PWPAE), are promising solutions to improve zero-day attack detection performance. Secondly, as 6G networks are expected to be zero-touch networks that enable fully autonomous attack detection and recovery, Automated ML (AutoML) techniques should be deployed to realize automated intrusion detection. Although in IDS-ML, we have used HPO, an important procedure of AutoML, to automatically optimize ML models, there are still many other AutoML procedures that are worth exploring, such as automated data collection, automated data pre-processing, automated feature engineering, automated model selection, and automated model updating/concept drift adaptation.

**References**

<https://www.sciencedirect.com/science/article/pii/S2665963822001300>

[www.kaggle.com/code/essammohamed4320/intrusion-detection-system-with-ml-dl](http://www.kaggle.com/code/essammohamed4320/intrusion-detection-system-with-ml-dl)

<https://github.com/SoftwareImpacts/SIMPAC-2022-260/blob/main/Figures/Tree-based_IDS_Overview.jpg>

<https://github.com/rahulvigneswaran/Intrusion-Detection-Systems/blob/master/all.py>

**Appendix 1.0**

**Code**

**Importing necessary libraries**

In [2]:

**import** numpy **as** np

**import** pandas **as** pd

**import** warnings

**import** matplotlib.pyplot **as** plt

**import** seaborn **as** sns

**import** tensorflow **as** tf

**from** tensorflow.keras **import** regularizers

**import** xgboost **as** xgb

**from** sklearn.decomposition **import** PCA

**from** sklearn **import** tree

**from** sklearn.naive\_bayes **import** GaussianNB

**from** sklearn.linear\_model **import** LogisticRegression

**from** sklearn.neighbors **import** KNeighborsClassifier

**from** sklearn.tree **import** DecisionTreeClassifier

**from** sklearn.preprocessing **import** RobustScaler

**from** sklearn.ensemble **import** RandomForestClassifier, RandomForestRegressor

**from** sklearn.model\_selection **import** train\_test\_split

**from** sklearn **import** svm

**from** sklearn **import** metrics

pd.set\_option('display.max\_columns',**None**)

warnings.filterwarnings('ignore')

**%**matplotlib inline

**Exploring the dataset**

In [3]:

*# Read Train and Test dataset*

data\_train **=** pd.read\_csv("D:\sem 6\ISM\Mini-project/KDDTrain+.txt")

In [4]:

*# Check data*

data\_train.head()

columns = (['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','outcome','level'])

# Assign name for columns

data\_train.columns = columns

data\_train.head()

data\_train.info()

data\_train.describe().style.background\_gradient(cmap='Blues').set\_properties(\*\*{'font-family':'Segoe UI'})

data\_train.loc[data\_train['outcome'] == "normal", "outcome"] = 'normal'

data\_train.loc[data\_train['outcome'] != 'normal', "outcome"] = 'attack'

def pie\_plot(df, cols\_list, rows, cols):

fig, axes = plt.subplots(rows, cols)

for ax, col in zip(axes.ravel(), cols\_list):

df[col].value\_counts().plot(ax=ax, kind='pie', figsize=(15, 15), fontsize=10, autopct='%1.0f%%')

ax.set\_title(str(col), fontsize = 12)

plt.show()

pie\_plot(data\_train, ['protocol\_type', 'outcome'], 1, 2)

**Data preprocessing**

def Scaling(df\_num, cols):

std\_scaler = RobustScaler()

std\_scaler\_temp = std\_scaler.fit\_transform(df\_num)

std\_df = pd.DataFrame(std\_scaler\_temp, columns =cols)

return std\_df

cat\_cols = ['is\_host\_login','protocol\_type','service','flag','land', 'logged\_in','is\_guest\_login', 'level', 'outcome']

def preprocess(dataframe):

df\_num = dataframe.drop(cat\_cols, axis=1)

num\_cols = df\_num.columns

scaled\_df = Scaling(df\_num, num\_cols)

dataframe.drop(labels=num\_cols, axis="columns", inplace=True)

dataframe[num\_cols] = scaled\_df[num\_cols]

dataframe.loc[dataframe['outcome'] == "normal", "outcome"] = 0

dataframe.loc[dataframe['outcome'] != 0, "outcome"] = 1

dataframe = pd.get\_dummies(dataframe, columns = ['protocol\_type', 'service', 'flag'])

return dataframe

scaled\_train = preprocess(data\_train)

x = scaled\_train.drop(['outcome', 'level'] , axis = 1).values

y = scaled\_train['outcome'].values

y\_reg = scaled\_train['level'].values

​

pca = PCA(n\_components=20)

pca = pca.fit(x)

x\_reduced = pca.transform(x)

print("Number of original features is {} and of reduced features is {}".format(x.shape[1], x\_reduced.shape[1]))

​

y = y.astype('int')

x\_train, x\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size=0.2, random\_state=42)

x\_train\_reduced, x\_test\_reduced, y\_train\_reduced, y\_test\_reduced = train\_test\_split(x\_reduced, y, test\_size=0.2, random\_state=42)

x\_train\_reg, x\_test\_reg, y\_train\_reg, y\_test\_reg = train\_test\_split(x, y\_reg, test\_size=0.2, random\_state=42)

Number of original features is 122 and of reduced features is 20

kernal\_evals = dict()

def evaluate\_classification(model, name, X\_train, X\_test, y\_train, y\_test):

train\_accuracy = metrics.accuracy\_score(y\_train, model.predict(X\_train))

test\_accuracy = metrics.accuracy\_score(y\_test, model.predict(X\_test))

train\_precision = metrics.precision\_score(y\_train, model.predict(X\_train))

test\_precision = metrics.precision\_score(y\_test, model.predict(X\_test))

train\_recall = metrics.recall\_score(y\_train, model.predict(X\_train))

test\_recall = metrics.recall\_score(y\_test, model.predict(X\_test))

kernal\_evals[str(name)] = [train\_accuracy, test\_accuracy, train\_precision, test\_precision, train\_recall, test\_recall]

print("Training Accuracy " + str(name) + " {} Test Accuracy ".format(train\_accuracy\*100) + str(name) + " {}".format(test\_accuracy\*100))

print("Training Precesion " + str(name) + " {} Test Precesion ".format(train\_precision\*100) + str(name) + " {}".format(test\_precision\*100))

print("Training Recall " + str(name) + " {} Test Recall ".format(train\_recall\*100) + str(name) + " {}".format(test\_recall\*100))

actual = y\_test

predicted = model.predict(X\_test)

confusion\_matrix = metrics.confusion\_matrix(actual, predicted)

cm\_display = metrics.ConfusionMatrixDisplay(confusion\_matrix = confusion\_matrix, display\_labels = ['normal', 'attack'])

​

fig, ax = plt.subplots(figsize=(10,10))

ax.grid(False)

cm\_display.plot(ax=ax)

**Logistic Regression**

lr = LogisticRegression().fit(x\_train, y\_train)

evaluate\_classification(lr, "Logistic Regression", x\_train, x\_test, y\_train, y\_test)

**KNN**

evaluate\_classification(knn, "KNeighborsClassifier", x\_train, x\_test, y\_train, y\_test)

knn = KNeighborsClassifier(n\_neighbors=20).fit(x\_train, y\_train)

evaluate\_classification(knn, "KNeighborsClassifier", x\_train, x\_test, y\_train, y\_test)

**Naive Bayes**

gnb = GaussianNB().fit(x\_train, y\_train)

evaluate\_classification(gnb, "GaussianNB", x\_train, x\_test, y\_train, y\_test)

Support Vector Machines

lin\_svc = svm.LinearSVC().fit(x\_train, y\_train)

evaluate\_classification(lin\_svc, "Linear SVC(LBasedImpl)", x\_train, x\_test, y\_train, y\_test)

**Decision Tree**

dt = DecisionTreeClassifier(max\_depth=3).fit(x\_train, y\_train)

tdt = DecisionTreeClassifier().fit(x\_train, y\_train)

evaluate\_classification(tdt, "DecisionTreeClassifier", x\_train, x\_test, y\_train, y\_test)

def f\_importances(coef, names, top=-1):

imp = coef

imp, names = zip(\*sorted(list(zip(imp, names))))

# Show all features

if top == -1:

top = len(names)

plt.figure(figsize=(10,10))

plt.barh(range(top), imp[::-1][0:top], align='center')

plt.yticks(range(top), names[::-1][0:top])

plt.title('feature importances for Decision Tree')

plt.show()

features\_names = data\_train.drop(['outcome', 'level'] , axis = 1)

f\_importances(abs(tdt.feature\_importances\_), features\_names, top=18)

fig = plt.figure(figsize=(15,12))

tree.plot\_tree(dt , filled=True)

**Random forest**

rf = RandomForestClassifier().fit(x\_train, y\_train)

evaluate\_classification(rf, "RandomForestClassifier", x\_train, x\_test, y\_train, y\_test)

f\_importances(abs(rf.feature\_importances\_), features\_names, top=18)

**Building an XGBOOST REgressor regressor to predict threat level**

xg\_r = xgb.XGBRegressor(objective ='reg:linear',n\_estimators = 20).fit(x\_train\_reg, y\_train\_reg)

name = "XGBOOST"

train\_error = metrics.mean\_squared\_error(y\_train\_reg, xg\_r.predict(x\_train\_reg), squared=False)

test\_error = metrics.mean\_squared\_error(y\_test\_reg, xg\_r.predict(x\_test\_reg), squared=False)

print("Training Error " + str(name) + " {} Test error ".format(train\_error) + str(name) + " {}".format(test\_error))

Training Error XGBOOST 0.9286577828406372 Test error XGBOOST 0.9955133892384386

y\_pred = xg\_r.predict(x\_test\_reg)

df = pd.DataFrame({"Y\_test": y\_test\_reg , "Y\_pred" : y\_pred})

plt.figure(figsize=(16,8))

plt.plot(df[:80])

plt.legend(['Actual' , 'Predicted'])

**Measuring effect of PCA**

rrf = RandomForestClassifier().fit(x\_train\_reduced, y\_train\_reduced)

evaluate\_classification(rrf, "PCA RandomForest", x\_train\_reduced, x\_test\_reduced, y\_train\_reduced, y\_test\_reduced)

**Neural networks**

model = tf.keras.Sequential([

tf.keras.layers.Dense(units=64, activation='relu', input\_shape=(x\_train.shape[1:]),

kernel\_regularizer=regularizers.L1L2(l1=1e-5, l2=1e-4),

bias\_regularizer=regularizers.L2(1e-4),

activity\_regularizer=regularizers.L2(1e-5)),

tf.keras.layers.Dropout(0.4),

tf.keras.layers.Dense(units=128, activation='relu',

kernel\_regularizer=regularizers.L1L2(l1=1e-5, l2=1e-4),

bias\_regularizer=regularizers.L2(1e-4),

activity\_regularizer=regularizers.L2(1e-5)),

tf.keras.layers.Dropout(0.4),

tf.keras.layers.Dense(units=512, activation='relu',

kernel\_regularizer=regularizers.L1L2(l1=1e-5, l2=1e-4),

bias\_regularizer=regularizers.L2(1e-4),

activity\_regularizer=regularizers.L2(1e-5)),

tf.keras.layers.Dropout(0.4),

tf.keras.layers.Dense(units=128, activation='relu',

kernel\_regularizer=regularizers.L1L2(l1=1e-5, l2=1e-4),

bias\_regularizer=regularizers.L2(1e-4),

activity\_regularizer=regularizers.L2(1e-5)),

tf.keras.layers.Dropout(0.4),

tf.keras.layers.Dense(units=1, activation='sigmoid'),

])

model.compile(optimizer='adam', loss=tf.keras.losses.BinaryCrossentropy(from\_logits=True), metrics=['accuracy'])

model.summary()

history = model.fit(x\_train, y\_train, validation\_data=(x\_test, y\_test), epochs=10, verbose=1)

plt.plot(history.history['loss'], label='loss')

plt.plot(history.history['val\_loss'], label='val\_loss')

plt.xlabel('Epoch')

plt.ylabel('SCCE Loss')

plt.legend()

plt.grid(True)

plt.plot(history.history['accuracy'], label='accuracy')

plt.plot(history.history['val\_accuracy'], label='val\_accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend()

plt.grid(True)

keys = [key for key in kernal\_evals.keys()]

values = [value for value in kernal\_evals.values()]

fig, ax = plt.subplots(figsize=(20, 6))

ax.bar(np.arange(len(keys)) - 0.2, [value[0] for value in values], color='darkred', width=0.25, align='center')

ax.bar(np.arange(len(keys)) + 0.2, [value[1] for value in values], color='y', width=0.25, align='center')

ax.legend(["Training Accuracy", "Test Accuracy"])

ax.set\_xticklabels(keys)

ax.set\_xticks(np.arange(len(keys)))

plt.ylabel("Accuracy")

plt.show()

keys = [key for key in kernal\_evals.keys()]

values = [value for value in kernal\_evals.values()]

fig, ax = plt.subplots(figsize=(20, 6))

ax.bar(np.arange(len(keys)) - 0.2, [value[2] for value in values], color='g', width=0.25, align='center')

ax.bar(np.arange(len(keys)) + 0.2, [value[3] for value in values], color='b', width=0.25, align='center')

ax.legend(["Training Precesion", "Test Presision"])

ax.set\_xticklabels(keys)

ax.set\_xticks(np.arange(len(keys)))

plt.ylabel("Precesion")

plt.show()

keys = [key for key in kernal\_evals.keys()]

values = [value for value in kernal\_evals.values()]

fig, ax = plt.subplots(figsize=(20, 6))

ax.bar(np.arange(len(keys)) - 0.2, [value[2] for value in values], color='g', width=0.25, align='center')

ax.bar(np.arange(len(keys)) + 0.2, [value[3] for value in values], color='b', width=0.25, align='center')

ax.legend(["Training Recall", "Test Recall"])

ax.set\_xticklabels(keys)

ax.set\_xticks(np.arange(len(keys)))

plt.ylabel("Recall")

plt.show()

**Appendix 2.0**

