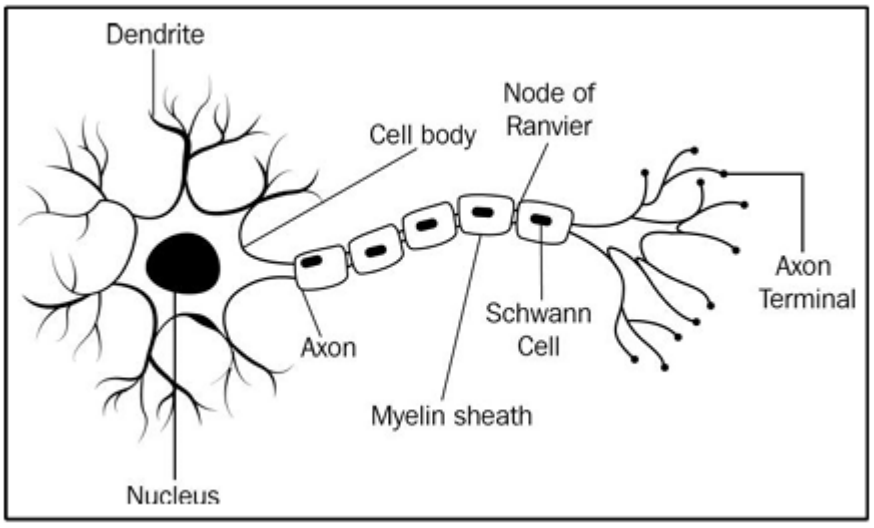
DETECTING CYBERSECURITY THREATS WITH AI

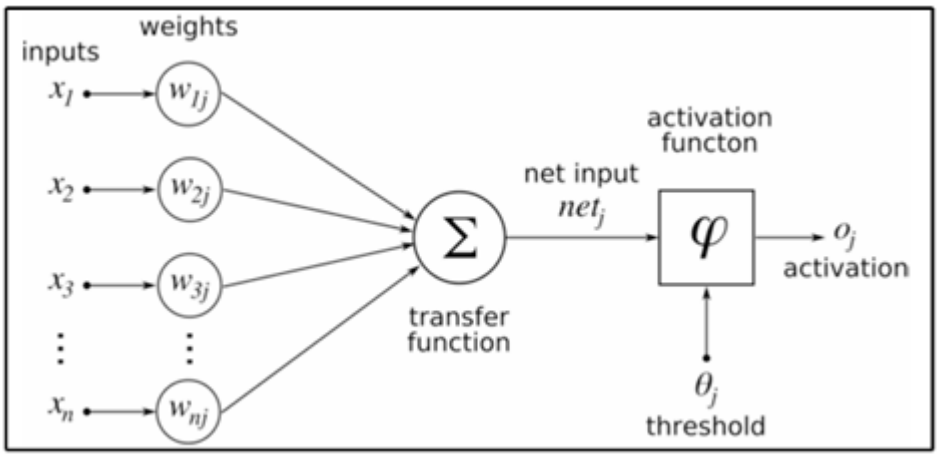
SYLLABUS

Detecting Spam with Perceptron - Perceptron Learning, Detecting Strategies, Pros and Cons of Perceptron. SVM Model, Spam Detection with Naive Bayes. Malware Detection-Tools- Malware Detection Strategies. Clustering algorithms-K Means - Depth, Steps, Pros and Cons. Polymorphic Malware Detection strategies - HMM Fundamentals. Network Anomaly Detection Techniques Network Attacks, Detecting Bot-Net Topology.

DETECTING SPAM WITH PERCEPTRON

- Spam detection is one of the **earliest successful applications of Artificial Intelligence (AI)**, particularly in the realm of **cybersecurity**.
- One of the key tools in this space, especially in its early iterations, was **SpamAssassin**, an open-source spam filtering system.
- The process of detecting spam messages has evolved alongside advancements in machine learning (ML) and AI.
- Early approaches in spam detection largely relied on **statistical methods** or simple rules-based systems, but as spammers evolved, more sophisticated techniques like **neural networks (NNs)** became the norm.
- Among the simplest and most foundational neural network models for spam detection is the **Perceptron**.





The Perceptron: A Basic Neural Network Model

- A **Perceptron** is one of the most basic forms of neural networks.
- It is inspired by the **human brain**, specifically how neurons process and transmit information.
- The Perceptron serves as an introductory model to neural networks, showcasing how **input data** can be **weighted** and passed through an **activation function** to produce an **output**.
- The Perceptron is one of the first successful implementations of a neuron in the field of Artificial Intelligence (AI).
- A **neuron** in the brain receives signals from other neurons, and it decides whether to fire (send a signal) based on the total incoming signals.
- The Perceptron mimics this behavior with a structure that associates an **output result** to one or more input levels.
- These inputs are processed through weighted sums that are passed through an activation function to decide whether the neuron "fires" (outputs a value).
- The mechanism that transforms the input data into an output value is implemented by making use of an appropriate weighing of the values of an input, which are synthesized and forwarded to an activation function, which, when exceeding a certain threshold, produces a value of output that is forwarded to the remaining components of the NN.
- A crucial part of the Perceptron's operation is the **weighting mechanism**.
- Each input value is assigned a weight that determines its importance in generating the output.
- These weights are **adjusted through iterative learning** to improve the Perceptron's ability to make accurate predictions.
- This iterative adjustment is done using a **cost function** that the algorithm tries to minimize.
- The weights are updated with the goal of finding an **optimal weight vector** that allows the Perceptron to correctly classify new, unseen data.

Spam Filters

- Let us imagine that we separate the emails we receive based on the presence or absence of a particular keyword in the text with a certain frequency.
- The number of occurrences of a suspicious keyword within the text of an email message is assigned a score if found to be spam.
- This score will help to identify subsequent messages.
- We will have to identify a threshold value that allows us to separate spam messages.
- If the calculated score exceeds the threshold value, the email will automatically be classified as spam otherwise it is classified as ham
- The threshold value will be constantly redetermined to take into account the new series of spam messages that we will meet in the future.
- An **anti-spam algorithm** classifies email based on suspicious keywords.
- For example, let us consider two words: Buy and Sell.
- A tabular representation is created showing the number of occurrences of individual keywords identified within the text of emails indicating the message as Spam or Ham.

Email	Buy	Sell	Spam or Ham
1	1	0	H
2	0	1	H
3	0	0	H
4	1	1	S

- We will assign a score to every single email message
- The score is calculated using a scoring function that counts the number of occurrences of suspicious keywords contained within the text.
- A possible scoring function could be the sum of the occurrence of two keywords represented by the B variable instead of Buy and S variable instead of Sell.
- Therefore, scoring function y= B+S;
- If both words are present in the text, the probability of spam increases
- Let us assign a lower weight of 2 to B and a greater weight of 3 to S then Y=2B+3S
- Choose the threshold value such that it will classify spam and ham correctly.

Email	B	S	2B + 3S	Spam or Ham?
1	1	0	2	H
2	0	1	3	H
3	0	0	0	H
4	1	1	5	S

- Using 4B+ 5S

Email		Buy	Sell	4B+5S	Spam or Ham
1		1	0	4	S
2		0	1	5	S
3		0	0	0	H
4		1	1	9	S

- At this point let us identify a threshold value which separates spam from ham. A value between 4 and 5 allows us to properly separate spam from ham. If new email score is greater than or equal to 4, then it is spam

Detecting spam with Linear Classifier:

- The equation that represents the function used to determine the score associated to every email message is

  y= 2B+ 3S
- This identifies a straight line in the Cartesian plane.
- The classifier used by our spam filter to classify email is called **linear classifier**
- Y = 2B+3S can be replaced with **y=∑WiXi**
  Wi is a vector of weight and Xi is matrix of indexed value
  i takes the value from 1 to n

- We can generalize the linear classifier to an unspecified no of variable n,

  $Y=W_1X_1+W_2X_2+\ldots\ldots\ldots\ldots+W_nX_n$
- Our function translates into a sum of products (between individual weight and variable) that can be easily represented as the product of matrices and vectors

  Y=wTx, wT represents the transposed weight carrier.
- An appropriate threshold value that correctly splits spam messages from ham messages needs to be identified.
- If the score associated with a single email message is equal to or higher than a threshold value, the email will be classified as spam, otherwise as ham
- In formal term we represent condition as follows

  If Wx >= 0 → f(y) = +1

  If Wx < 0 → f(y) = -1

  Therefore $Y= W_0X_0+ W_1X_1+ \ldots\ldots\ldots WnXn = \sum W_iX_i = wTx$
- Index I assumes the values from 0 to n.

PERCEPTRON LEARNING

The Perceptron learns by iteratively adjusting its weights. The learning process consists of three key steps:

1. **Initial Weight Assignment**: Initially, the weights are set to zero or small random values. The weights for each feature are initialized to small random values (or zero). These weights represent the importance of each feature (keyword) in determining the classification.

2. **Output Calculation**: For each input email, the Perceptron calculates an output value based on the weighted sum of the features (keyword occurrences).

3. **Weight Update**: If the output does not match the expected output (the true label: spam or ham), the weights are adjusted based on the error (difference between expected and predicted outputs). The weight update rule is:

$$W_i = W_i + \Delta W_i$$

Where: :

$$\Delta W_i = \lambda(y - y_i)x_i$$

- Wi is the weight for feature i.
- λ is the **learning rate** (a small constant).
- y is the actual label (spam or ham).
- yi is the predicted output.
- xi is the input feature value.

The **learning rate** determines how much the weights are adjusted after each iteration. If the learning rate is too high, the weights may oscillate and fail to converge; if it's too low, the learning process may be very slow.

This process is repeated for several iterations (or until convergence) across all training data. The Perceptron continues to adjust its weights to reduce the error in predictions, which improves its ability to classify future emails accurately.

If the data is **linearly separable** (i.e., a straight line can separate spam from ham), the Perceptron will converge to an optimal set of weights. However, if the data is not linearly separable, convergence may not be achieved.

DETECTING STRATEGIES

The effectiveness of spam detection with the Perceptron depends on the **features** (i.e., the keywords or characteristics) used to train the model, the **learning rate**, and the **threshold for classification**. Here are some key strategies used in spam detection:

1. **Keyword-Based Feature Extraction**:
    - A simple method is to use specific keywords in the email body to identify spam. For example, words like “free,” “buy now,” and “limited offer” are common indicators of spam.
    - The frequency of these keywords in the email is counted and used as input features for the Perceptron.
2. **TF-IDF (Term Frequency-Inverse Document Frequency)**:
    - A more sophisticated approach involves using **TF-IDF** to represent email content.

    - TF-IDF assigns higher weights to terms that are unique to the email (high term frequency) but not common across all emails (low document frequency).
    - This helps in identifying not just obvious spam words but also less frequent, more subtle indicators of spam.
3. **Training with Labeled Data**:
    - The Perceptron requires labeled data (spam or ham) to learn the appropriate weights.
    - A **training dataset** with thousands of labelled emails is used to train the model.
    - A well-labeled training set improves the model's accuracy in distinguishing spam from ham.
4. **Threshold Adjustment**:
    - After calculating the score for each email, a **threshold value** is set.
    - If the score exceeds the threshold, the email is classified as spam.
    - This threshold may be adjusted over time to improve accuracy, especially as new spam tactics emerge.
5. **Continuous Learning**:
    - As spammers evolve, new words and strategies emerge.
    - A key strategy in modern spam detection is the ability to **continually retrain** the Perceptron with new data to adapt to these changing tactics.

PROS AND CONS OF PERCEPTRON

Being essentially a **binary linear classifier**, the Perceptron is able to offer accurate results only if the analyzed data can be **linearly separable**; that is, it is possible to identify a straight line (or a hyperplane, in case of multidimensional data) that completely bisects the data in the Cartesian plane.

If instead the analyzed data was not linearly separable, the Perceptron learning algorithm would oscillate indefinitely around the data, looking for a possible vector of weights that can linearly separate the data without, however, being able to find it.

Therefore, the convergence of the Perceptron is only possible in the presence of linearly separable data, and in the case of a **small learning rate**. If the classes of data are not linearly separable, it is of great importance to set a maximum number of iterations in order to prevent the algorithm from oscillating indefinitely in search of an nonexistent optimal solution.

Despite its simplicity and effectiveness for certain tasks, the **Perceptron** has its own set of advantages and limitations.

Pros:

1. **Simplicity**:
    - The Perceptron is easy to understand and implement. It's often used as an introductory model for learning about neural networks and machine learning.
2. **Efficiency**:
    - The Perceptron algorithm can quickly classify data based on a small number of features, making it computationally efficient for simpler classification tasks like spam detection.
3. **Binary Classification**:
    - The Perceptron works well for binary classification tasks, making it ideal for applications like email filtering.
4. **Fast Convergence for Linearly Separable Data**
    - If the data is linearly separable, the Perceptron converges quickly to an optimal solution, meaning it can effectively separate spam from ham with minimal training time.

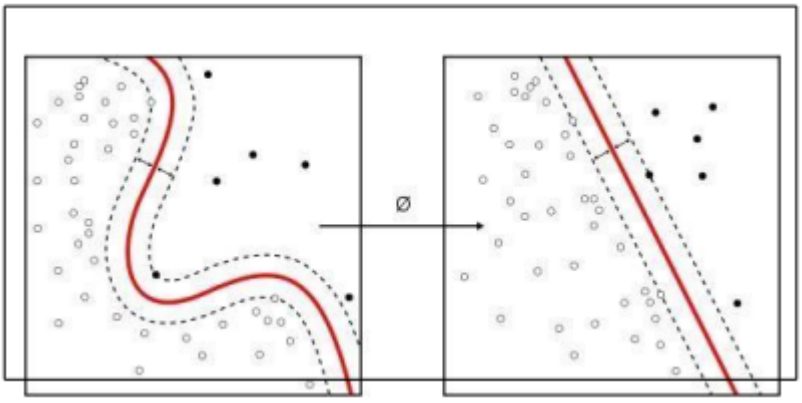
Cons:

1. **Limited to Linearly Separable Data**:
    - One of the main drawbacks of the Perceptron is that it can only **classify linearly separable data**. If spam and ham messages cannot be separated by a straight line, the Perceptron will fail to converge.
2. **Oscillation with Non-Linearly Separable Data**:
    - For data that is not linearly separable, the Perceptron's learning algorithm will **oscillate indefinitely**, never reaching an optimal solution. This makes it unsuitable for complex datasets without further adjustments.
3. **Inability to Handle Complex Patterns**:
    - The Perceptron is a **single-layer neural network** and, therefore, cannot capture more complex patterns or relationships within the data. More advanced models, like multi-layer neural networks or Support Vector Machines (SVMs), are better suited for complex classification tasks.
4. **Sensitivity to Initial Weights and Learning Rate**:
    - The Perceptron's performance can be sensitive to the choice of **initial weights** and the **learning rate**. If the learning rate is too high or too low, the model may either fail to converge or converge very slowly.

Overcoming Perceptron Limitations with SVMs: One way to address the Perceptron's limitations is to allow for a wider margin of data separation between classes

SVM MODEL

- Support Vector Machines (SVMs) are a powerful class of supervised learning algorithms used for classification tasks, including **spam detection**.
- SVMs are particularly well-suited for problems where the goal is to classify data into one of two categories, such as **spam** or **ham** (non-spam) in the context of email or message filtering.
- Let's dive deeper into how SVMs can be effectively used for spam detection.
- SVMs are supervised learning algorithms, meaning they require a **labeled dataset** for training.
- For spam detection, a dataset containing emails or messages that are labeled as **spam** or **ham** (genuine messages) is used.
- The SVM learns from this labeled data and constructs a model capable of classifying new, unseen emails/messages as spam or ham.
- The task of SVMs is to identify the hyperplane that best separates classes of data that can be represented in a multidimensional space.
- It is possible, however, to identify different hyperplanes that correctly separate the data from each other; in this case, the choice falls on the hyperplane that optimizes the prefixed margin, that is, the distance between the hyperplane and the data.
- **Training Data**: The training dataset includes both positive (spam) and negative (ham) samples. Each sample is represented by a vector of features derived from the email content (e.g., word frequencies, presence of certain keywords, etc.).
- **Classification**: Once the model is trained, the SVM uses the decision boundary (or hyperplane) it has learned to classify future emails/messages. If the new email lies on one side of the hyperplane, it is classified as **spam**, and if it lies on the other side, it is classified as **ham**.



SVMs are often seen as an **extension of the Perceptron**. Both are **linear classifiers**, but SVMs offer distinct advantages over the Perceptron, particularly when dealing with non-linearly separable data.

- **Perceptron**: The goal of a Perceptron is to minimize classification errors by adjusting weights during training. However, the Perceptron is limited because it only works well for linearly separable data. If the data cannot be separated by a straight line (or hyperplane in higher dimensions), the Perceptron may fail to converge, resulting in oscillation during training.

- **SVM**: Unlike the Perceptron, the SVM aims to **maximize the margin** between classes, not just minimize classification errors. The **margin** is the distance between the decision boundary (hyperplane) and the closest training samples, which are called **support vectors**. A larger margin leads to better generalization and fewer classification errors.

    - In simple terms, SVM tries to place the decision boundary where the **distance between it and the closest points (support vectors) is as large as possible**.

This emphasis on margin maximization allows SVMs to handle cases where the data is **not linearly separable** more effectively, using various strategies like **kernel methods**.

At the core of the SVM algorithm is the concept of a **hyperplane**, which separates the two classes (spam and ham). The primary objective of SVM is to find the hyperplane that **maximizes the margin**, which is defined as the distance between the hyperplane and the nearest data points from both classes.

- **Support Vectors**: The data points that are closest to the decision boundary are called **support vectors**. These points are critical to defining the position of the hyperplane. The SVM relies on them to determine the optimal separation between the two classes.

- **Margin Maximization**: Maximizing the margin between the classes is beneficial because wider margins tend to improve the **generalization** of the classifier. A classifier with a narrow margin might overfit the training data and perform poorly on unseen data.

Mathematically, the optimization problem in SVM seeks to maximize the **margin (μ)** subject to the constraint that all correctly classified data points satisfy:

$$y_i(w^T x_i + b) \geq 1$$

where:

- yi is the true label of the sample (+1 for spam, −1 for ham),

- w is the weight vector,
- xi is the feature vector for the data point,
- b is the bias term.

This optimization ensures that the margin between the hyperplane and the nearest points is as large as possible, which helps the model to generalize better on unseen data.

Linear vs. Non-Linear Classifiers in SVMs

One of the key strengths of SVMs is their ability to handle **non-linearly separable data**. While the Perceptron is limited to linear classification, SVMs can use different types of **kernels** to map the input data to a higher-dimensional space where a linear hyperplane can separate the data.

- **Linear SVM**: When the data is linearly separable or approximately linearly separable, a **linear kernel** can be used. In this case, SVM finds a straight-line hyperplane to separate the classes.
- **Non-Linear SVM**: When the data is not linearly separable, SVM can use **kernel functions** such as the **Radial Basis Function (RBF)** kernel, **polynomial kernel**, or **sigmoid kernel** to map the data into a higher-dimensional feature space. In this new space, the data becomes linearly separable, and the SVM can find an optimal hyperplane to classify the data.

Using a **linear SVM** is still effective in many real-world spam detection cases, especially when feature engineering is carefully done to ensure good separability.

Example of a Spam Filter Using a Linear SVM

Let's walk through a simple example using a linear SVM to detect spam based on text data, such as an SMS Spam Message dataset.

- **Dataset**: The dataset contains SMS messages that are labeled as either spam or ham.
- **Feature Extraction**: Each SMS is converted into a **feature vector** (e.g., word counts or TF-IDF features).
- **Training**: The linear SVM is trained on the feature vectors of the SMS messages. The model tries to find the hyperplane that best separates the spam and ham messages.

- **Testing and Evaluation**: After training, the SVM is tested on a separate test set. The model's accuracy can be evaluated by comparing its predicted classifications with the actual labels.

Image Spam Detection with SVMs

SVMs are also effective for **image spam detection**, where spammers often use images to evade text-based filters. The SVM algorithm allows us to deal with even more complex real-world classification cases, such as in the case of spam messages represented by images, instead of simple text. Detecting spam in images typically involves two main approaches:

- **Content-based Filtering**: This method extracts text from images using **Optical Character Recognition (OCR)** and applies traditional text-based spam detection methods to the extracted text. The approach consists of trying to identify the suspect keywords that are most commonly used in textual spam messages even within images; to this end, pattern recognition techniques leveraging optical character recognition (OCR) technology are implemented in order to extract text from images.
- **Non-content-based Filtering**: In this approach, features of the image itself, such as color, texture, or statistical features, are extracted to distinguish spam images from non-spam images. **Deep learning techniques** can be used for feature extraction, but traditional machine learning models like SVMs can still perform well with carefully crafted features.

In a typical image spam detection scenario:

1. The image features are extracted (e.g., using OCR or feature engineering techniques like color histograms).
2. These features are used to train an SVM classifier.
3. The trained SVM can then classify whether an image is spam or not.

SPAM DETECTION WITH NAIVE BAYES

One of the advantages associated with the use of Naive Bayes is the fact that it requires little starting data to begin classifying input data.

Naive Bayes is a family of **probabilistic** algorithms based on **Bayes' Theorem** and is particularly well-suited for **text classification tasks**, such as spam detection.
The core idea behind Naive Bayes classifiers is that features are **independent** given the class (spam or ham). This "naive" assumption simplifies the computation of probabilities and makes the algorithm computationally efficient.

Bayes' Theorem provides a way to calculate the **posterior probability** of a class, given some observed evidence (features). For a given email or message x, Bayes' Theorem states:

$$P(C|x) = \frac{P(x|C)P(C)}{P(x)}$$

Where:

- $P(C|x)$ is the **posterior probability** of the class C (spam or ham) given the observed features x (words or terms in the message).
- $P(x|C)$ is the **likelihood** of observing the features x given the class C.
- $P(C)$ is the **prior probability** of the class C, i.e., how likely the class is in the general population (spam or ham).
- $P(x)$ is the **evidence** or the total probability of the observed features, which is typically a normalizing constant.

In the **Naive Bayes** classifier, we assume that the features (words) are **conditionally independent** given the class. This means that the probability of the features occurring together is simply the product of their individual probabilities:

$$P(x|C) = P(x_1, x_2, ..., x_n|C) = P(x_1|C)P(x_2|C)...P(x_n|C)$$

Thus, the Naive Bayes classifier simplifies the posterior probability calculation to:

$$P(C|x) \propto P(C) \prod_{i=1}^{n} P(x_i|C)$$

Where:

- $x_1, x_2, ..., x_n$ are the features (words in the message).
- $P(x_i|C)$ is the likelihood of observing the word x_i given the class C.

To train a Naive Bayes classifier for spam detection, we need to estimate the **prior** and **likelihood** probabilities for each class (spam or ham). This is done using the training data, which consists of labeled samples of messages.

Step 1: Calculate Prior Probabilities

The prior probabilities are the proportion of spam and ham messages in the training dataset. They are calculated as:

$$P(\text{Spam}) = \frac{\text{Number of spam messages}}{\text{Total number of messages}}$$

$$P(\text{Ham}) = \frac{\text{Number of ham messages}}{\text{Total number of messages}}$$

Step 2: Calculate Likelihood Probabilities

Next, we calculate the likelihood of each word occurring in spam and ham messages. This is done by counting the occurrences of each word in both spam and ham messages and normalizing by the total number of words in each class.

For a word w, the likelihood in spam is:

$$P(w|\text{Spam}) = \frac{\text{Count of } w \text{ in spam messages}}{\text{Total number of words in spam messages}}$$

Similarly, for ham:

$$P(w|\text{Ham}) = \frac{\text{Count of } w \text{ in ham messages}}{\text{Total number of words in ham messages}}$$

Step 3: Apply the Naive Bayes Formula

Once the prior and likelihood probabilities are estimated, we can classify new messages. Given a new email with features x=(x1,x2,...,xn), the classifier computes:

$$P(\text{Spam}|x) \propto P(\text{Spam}) \prod_{i=1}^{n} P(x_i|\text{Spam})$$

$$P(\text{Ham}|x) \propto P(\text{Ham}) \prod_{i=1}^{n} P(x_i|\text{Ham})$$

The classifier assigns the class with the highest posterior probability to the new message.

Advantages of Naive Bayes for Spam Detection

- **Simplicity**: Naive Bayes is simple to understand and implement. It is computationally efficient and works well even with large datasets.
- **Fast**: Due to the independence assumption, Naive Bayes requires fewer computations compared to more complex models like SVMs or neural networks.
- **Works Well for Text Data**: Naive Bayes performs well on text classification problems because it handles high-dimensional feature spaces (i.e., the large number of unique words) efficiently.
- **Good Performance with Small Datasets**: Even with limited labeled data, Naive Bayes can provide good classification results, which is helpful in cases where labeled data is scarce.

Limitations of Naive Bayes for Spam Detection

- **Independence Assumption**: Naive Bayes assumes that all features (words) are independent, which is not true in real-world data. For example, the presence of certain words might be correlated, which Naive Bayes does not account for.
- **Poor Performance on Highly Complex Texts**: For more complex classification tasks with subtle distinctions between classes (e.g., distinguishing very similar types of spam), Naive Bayes may not perform as well as more sophisticated models.
- **Sensitivity to Noisy Data**: Naive Bayes can be sensitive to irrelevant or highly frequent words, which might dominate the likelihood calculations.

MALWARE DETECTION

Malware (malicious software) refers to any program or code designed to harm, exploit, or otherwise compromise a computer system.
Malware detection involves identifying and mitigating the effects of such software, which includes viruses, worms, trojans, ransomware, spyware, etc.

Detection aims to **identify malware before it can cause damage** or steal information.

- Malware analysis is learning to distinguish legitimate binary files from those that are potentially dangerous for the integrity of the machines and the data they contain.
- We refer generically to binary files rather than to executable files (that is, files with extensions such as .exe or .dll), since malware can even hide in apparently innocuous files such as image files (files with extensions such as .jpg or .png).
- The first stage of the spread of a malware often happens by compromising the integrity of the files residing within the machines being attacked.

Artificial Intelligence for malware detection:

- It is practically impossible to think of dealing with these threats effectively using only the analysis conducted by human operators.
- Therefore, it is necessary to introduce algorithms that allow us to at least automate the preparatory phase of malware analysis
- It is necessary that the machines can respond effectively, adapting themselves to the contextual changes related to the spread of unprecedented threats

It is possible to compile a classification of the most common types of malware, in order to understand which are the most effective measures of prevention, and contrast their effectiveness for dealing with each malware species:

- **Trojans**: Executables that appear as legitimate and harmless, but once they are launched, they execute malicious instructions in the background
- **Botnets**: Malware that has the goal of compromising as many possible hosts of a network, in order to put their computational capacity at the service of the attacker
- **Downloaders**: Malware that downloads malicious libraries or portions of code from the network and executes them on victim hosts
- **Rootkits**: Malware that compromises the hosts at the operating system level and, therefore, often come in the form of device drivers, making the various countermeasures (such as antiviruses installed on the endpoints) ineffective
- **APTs**: APTs are forms of tailored attacks that exploit specific vulnerabilities on the victimized hosts
- **Ransomwares**: Malware that proceeds to encrypt files stored inside the host machines, asking for a ransom from the victim (often to be paid in Bitcoin) to obtain the decryption key which is used for recovering the original files
- **Zero days** (0 days): Malware that exploits vulnerabilities not yet disclosed to the community of researchers and analysts, whose characteristics and impacts in terms of security are not yet known, and therefore go undetected by antivirus software

The different techniques for detecting malware include Signature based detection, Heuristic based detection, Behaviour based detection, Sandboxing, ML based detection etc.

MALWARE DETECTION TOOLS

Malware analysis and detection often rely on a combination of **static** and **dynamic analysis tools**. Below are key tool categories and examples:

1. **Disassemblers:** Convert **binary code (machine code)** into **assembly language**, allowing analysts to inspect and understand the program's behavior without executing it.

   Example: IDA Pro, Disasm

2. **Debuggers:** Used to **run malware step-by-step**, monitor registers, set breakpoints, and inspect memory during execution.

   Example: OllyDbg, WinDbg, IDA

3. **System Monitors:** These tools track system activity such as **file modifications**, **registry access**, **process creation**, and **thread activity**, which is crucial for **behavioral malware detection**.

   Example: Process Monitor, Process Explorer, Autoruns

4. **Network Monitors:** Network monitoring tools are essential for detecting **command-and-control (C&C) communications**, data exfiltration, or suspicious network activity triggered by malware.

   Example: Wireshark, TCPView, tcpdump

5. **Unpacking Tools & Packer Identifiers:** Malware authors often use **packers or obfuscators** to hide the actual code. These tools help in **unpacking** or detecting if a binary is packed.

   Example: PEiD, Unpacker, UPX

6. **Binary and Code Analysis Tools:** These tools help inspect the **structure of executables (PE files)** and their metadata, including headers, imports/exports, and resources.

   Examples: PEView, PE Explorer, LordPE, ImpREC

MALWARE DETECTION STRATEGIES

The most common malware detection activities are:

1. **Hashes file calculation**: To identify known threats already present in the knowledge base
2. **System monitoring**: To identify anomalous behavior of both the hardware and the operating system such as an unusual increase in CPU cycles, a particularly heavy disk writing activity, changes to the registry keys, and the creation of new and unsolicited processes in the system
3. **Network monitoring**: To identify anomalous connections established by host machines to remote destinations

Static Malware Analysis- Inspect Without Execution

- The first step in malware analysis begins with the evaluation of the presence of suspect artifacts in binary files, without actually running (executing) the code.
- The complexity of techniques used in this phase goes under the name of static malware analysis.
- Static malware analysis is the process of examining malicious code **without executing it**.
- It relies on analyzing the **binary structure**, **machine-level instructions**, **embedded resources**, and **metadata** of the file.
- This technique aims to derive intelligence about the malware's functionality, origin, or intention **without triggering its behavior**.
- Static malware analysis consists of the following:

    1. Identifying the objectives considered of interest for the analysis

    2. Understanding the flow of executable instructions

    3. Identifying known patterns and associating them to possible malware

- Analysis tools and procedures are used inorder to perform the following functions:
    - Identifying calls to system APIs
    - Decoding and manipulating string data for obtaining sensitive information (for example, domain names and IP addresses)
    - Detecting the presence and invocation by downloading other malware codes (for example, Command and Control (C2), backdoors, and reverse shells)

Objectives:

- Understand the malware's **purpose and mechanism**.
- Extract **indicators of compromise (IOCs)** such as IPs, domains, or filenames.
- Analyze **control flow and data flow** within the binary.
- **Identify strings** and **API calls** used in malicious functions.

Methodologies:

The methodology used by static malware analysis consists of the **examination of the machine instructions** (assembly instructions) present in the disassembled binary image of the malware (malware disassembly), in order to identify its harmful potentialities and evaluate the external characteristics of the binary code, before proceeding with its execution.

Challenges:

- **Obfuscation and Packing**: Malware may be packed (e.g., with UPX) or encrypted to prevent analysis.
- **Incorrect Disassembly**: Misleading instructions or anti-disassembly tricks can make reverse engineering difficult.
- **Hidden Code**: Code may be embedded within harmless-looking resources like images, icons, or metadata.

How to perform static analysis:

Once you have verified that the disassembled malware is reliable, it is possible to proceed in different ways: each analyst, in fact, follows their own preferred strategy, which is based on the experience and objectives they intend to pursue.

In principle, the adoptable strategies are as follows:

- **Analyze the binary instructions** in a systematic way, without executing them. It is an effective technique for limited portions of code that become complicated in cases of large malware, as the analyst must keep track of the status of the data for each instruction analyzed.
- **Scan the instructions** to look for sequences that are considered to be of interest, **setting breakpoints** and **partially executing the program** up to the breakpoint, and then examining the status of the program at that point. This approach is often used to determine the presence of system calls deemed dangerous, based on the sequence in which these calls are invoked.
- In the same way, it is possible to detect the absence of certain API calls. A code that does not present invocations to the system calls (for example, network- related calls), which is necessary for issuing network connections, cannot obviously represent a backdoor.
- **Search for sensitive information** (such as domain names and IP addresses) in a string format inside the disassembled image. Also, in this case, it is possible to set debugger breakpoints in correspondence with the network calls and detect any domain names or remote IP addresses that get contacted by the malware when connecting to the internet.

Hardware requirements for static analysis

- Unlike dynamic analysis, static analysis usually requires fewer specific resources in terms of hardware, since, in principle, the analyst does not execute the malicious code under analysis.
- In the case of dynamic malware analysis, non-trivial hardware requirements may be required, and in some cases it is not enough to use virtual machines. This is due to the presence of countermeasures (anti-analysis tricks)

implemented by the malware, which prevent the execution of the code if the presence of a virtual machine is detected.

Dynamic Malware Analysis- Behaviour Under Execution

- Dynamic analysis involves executing the malware in a **sandboxed or isolated environment** to monitor its **runtime behavior**. This method provides visibility into how malware interacts with the operating system, network, and system files.
- The distinctive character of the dynamic malware analysis is the fact that, unlike the static malware analysis, the binary file gets executed (often in isolated and protected environments, known as malware analysis labs, which make use of sandboxes and virtual machines to prevent the widespread of malware in the corporate network).
- Therefore, this strategy entails analyzing the dynamic behavior, that is, verifying, for example, that the malicious executable does not download malicious libraries or portions of code (payloads) from the internet, or proceeds to modify its own executable instructions at each execution, thus making the signature-based detection procedures (used by the antiviruses) ineffective.

Objectives:

- Observe **real-time behavior** of malware.
- Identify **network activity**, file system changes, registry modifications, and process injections.
- Track **payload downloads**, **code injection**, and **system compromise attempts**.

Limitations:

- **Sandbox Detection**: Malware may detect it's being executed in a virtualized or sandboxed environment and behave benignly.
- **Time Bombs and Logic Bombs**: Some malware executes only under specific conditions (e.g., date/time, geographic IP).
- **Heavy Resource Use**: Dynamic analysis can be resource-intensive, requiring VMs or dedicated analysis environments.

Anti-Analysis Tricks Used by Malware Authors

The countermeasures usually adopted by malware developers, which prevent malware analysis or make it more difficult, rely on **encryption of the payloads**, the **use of packers, of downloaders**, and others.

These tricks are normally detectable with dynamic malware analysis; however, even dynamic malware analysis suffers from limitations related to the use of virtual

machines—for example—whose presence can be easily detected by malware by exploiting some execution tricks, as follows:

- **Execution of instructions that expect a default behavior**: The malware can calculate the time that elapses in the execution of certain operations, and if these were performed more slowly than expected, it can deduce consequently that the execution takes place on a virtual machine.
- **Hardware-based virtual machine detection**: Through the execution of some specific instructions at the hardware level (for example, the instructions that access CPU-protected registers, such as sldt, sgdt, and sidt).
- **Accessing certain registry keys** such as HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\Disk\Enum.
  When the malware detects the presence of a virtual machine, it stops working in the expected way, evading attempts for it to be detected by analysts.

To evade detection and frustrate analysts, modern malware employs sophisticated anti-analysis techniques:

🛡️ Anti-Static Analysis:

- **Encryption/Obfuscation**: Code or strings are encrypted and decrypted at runtime.
- **Packers**: Compress and encrypt the payload (e.g., UPX, Themida).
- **Code Polymorphism**: Code changes its appearance with each execution.
- **Control Flow Obfuscation**: Makes reverse engineering difficult.

🛡️ Anti-Dynamic Analysis:

- **VM Detection**: Looks for registry keys, MAC addresses, or device names associated with virtual machines.
- **Timing Checks**: Detects if instructions execute slower (a sign of a monitored environment).
- **Hardware Instruction Traps**: Uses `sidt`, `sgdt`, `sldt` to detect virtualized CPUs.
- **Environment Checks**: Detects debugging tools or analysis frameworks and alters behavior.

If malware detects is being analyzed, it may:

- Exit prematurely.
- Delay payload activation.
- Execute a fake benign path.
- Crash intentionally to mislead analysts.

Understanding Clustering Algorithms

The analysis of similarities can be carried out in an automated form, by using clustering algorithms.
Clustering algorithms consist of identifying and exploiting the similarities that characterize certain types of phenomena.
In technical terms, it is a matter of distinguishing and recognizing, within a dataset, the features whose values change with high frequency, from those features whose values are shown to remain systematically stable instead.
We can follow these two types of approaches in identifying similarities:

1. **Supervised**: The similarities are identified on the basis of previously categorized samples (for example, the k-Nearest Neighbors (k-NNs) algorithms).
2. **Unsupervised**: Similarities are identified independently by the algorithm itself (for example, the K-Means algorithm).

The estimate of the similarity between the features is carried out by associating them with a definition of **distance**.
Some of the measures that can be selected to identify the distances between numerical vectors are as follows:

1. **Euclidean distance**: This feature identifies the shortest path (the straight line) that unites two points in the Cartesian space, and is calculated with the following mathematical formula:

   $$E(x,y) = \sqrt{\sum (x-y)^2}$$

2. **Manhattan distance**: This feature is obtained from the sum of the absolute values of the differences calculated on the elements of the vectors. Unlike the Euclidean distance, the Manhattan distance identifies the longest route that joins the two points; in formulas, it is equivalent to the following:

   $$M(x,y) = \sum |x-y|$$

3. **Chebyshev distance**: This is obtained by calculating the maximum value of the absolute differences between the elements of the vectors; in formulas, it is equivalent to the following:

   $$C(x,y) = max|x-y|$$

   The use of the Chebyshev distance is particularly useful if the number of dimensions to be taken into account is particularly high, although most of them are irrelevant for analysis purposes.

The clustering process therefore consists of classifying together elements that show certain similarities between them.

CLUSTERING ALGORITHMS

Clustering is an **unsupervised machine learning** technique used to **group similar data points** together based on certain features or patterns.

Unlike classification (supervised learning), clustering **does not require labeled data**.

The primary goal is to **maximize intra-cluster similarity** (data points within a cluster are similar) and **minimize inter-cluster similarity** (data points in different clusters are dissimilar).

Clustering is widely used when:

- Labels (ground truth) are **not available**.
- You want to **explore structure** or **hidden patterns** in data.
- You need to **detect anomalies** or **group behaviors**.
- You want to **compress and organize** data.

Some of the most commonly used algorithms are listed as follows:

1. **K-Means**: One of the most widespread among the unsupervised clustering algorithms. **Distance based algorithm.** K-Means can enlist among its strengths the simplicity of implementation and the capability of unveiling hidden patterns within the data. This can be achieved by proceeding to the independent identification of possible labels.

2. **K-NNs**: This is an example of a **lazy learning** model. The K-NN algorithm only starts working in the evaluation phase, while in the training phase it simply limits itself to memorizing the observational data. Due to these characteristics, the use of k-NN is inefficient in the presence of large datasets.

3. **Density-Based Spatial Clustering of Applications with Noise** (DBSCAN): Unlike K-Means, which is a distance-based algorithm, DBSCAN is an example of a **density based algorithm**. As such, the algorithm tries to classify data by identifying high density regions.

K MEANS

K-Means is a fundamental and widely-used **unsupervised learning algorithm** used in clustering problems, where the primary goal is to uncover **hidden patterns** or **groupings** in unlabeled data. It operates by organizing data into **‘K’ distinct clusters**, such that **data points within a cluster are more similar to each other than to those in other clusters**.

It achieves this by **minimizing a cost function**, typically based on **Euclidean distance**, between each data point and the centroid (center) of the cluster it belongs to. The **centroid** is the mean of the points in a cluster and serves as a representative feature for that group.

- K-Means is an unsupervised algorithm, that is, it does not presuppose the prior knowledge of the labels associated with the data.
- The algorithm takes its name from the fact that its final purpose is to divide the data into k different subgroups.
- Being a clustering algorithm, it proceeds to the subdivision of the data into different subgroups on the basis of a chosen measure to represent the distance of the single samples (usually, this measure is the Euclidean distance) from the center of the respective cluster (also known as centroid).
- In other words, the K-Means algorithm proceeds to group the data into distinct clusters, minimizing a cost function represented by the Euclidean distance calculated between the data (considered as points in space) and the respective centroids.
- The algorithm returns the individual samples grouped in correspondence of each cluster, whose centroids constitute the set of distinctive features identified by the algorithm as representative of the different categories that can be identified within the dataset.

K-Means is frequently applied in:

- **Customer segmentation** in marketing,
- **Image segmentation** in computer vision,
- **Malware classification and anomaly detection** in cybersecurity,
- **Document clustering** in Natural Language Processing (NLP).

It is especially valuable in scenarios where **labels are unavailable**, and one wishes to **discover intrinsic structure** within the data.

STEPS

The K-Means algorithm follows an **iterative refinement process**. Here are the detailed steps:

1. Initialization of Centroids

- Decide the number of clusters (**'K'**) to divide the dataset into.
- Randomly select K data points from the dataset as the **initial centroids**.
- Choosing K is often the most challenging part as it may not be known a priori and may require **Exploratory Data Analysis (EDA)**, visual inspection, or techniques like the **Elbow Method** to estimate a suitable number.

2. Assign Each Data Point to the Nearest Centroid

- For every point in the dataset, compute the **Euclidean distance** to each of the centroids.
- Assign the data point to the cluster whose centroid is **closest**.

3. Update Centroids

- After all points are assigned, recalculate the centroids by taking the **mean of all points** in each cluster.
- These new centroids represent the updated centers of the clusters.

4. Repeat Until Convergence

- Reassign all data points based on updated centroids and update the centroids again.
- Continue the iteration until:
    - **No changes** occur in data point assignments (i.e., convergence),
    - Or the **maximum number of iterations** is reached,
    - Or the **change in centroids** is below a small threshold (i.e., stable).

This loop ensures that the **intra-cluster variance** is minimized progressively, resulting in optimal cluster formation under the given K.

PROS AND CONS

PROS

- **Simple and easy to implement** – Minimal algorithmic complexity.
- **Fast and efficient** – Especially suitable for large datasets.
- **Scalable** – Performs well with increasing data sizes.
- **Unsupervised learning** – No need for labeled data.
- **Interpretable results** – Centroids are easy to analyze and understand.
- **Reveals natural groupings** in the data that may not be initially visible.
- **Widely supported** – Available in many libraries (e.g., Scikit-learn, TensorFlow, etc.).

CONS

- **Must specify the number of clusters (K)** beforehand.
- **Sensitive to initial centroid selection** – Can lead to different outcomes.
- **Converges to local minima** – Final solution may not be optimal.
- **Assumes spherical clusters of similar size** – Not ideal for irregularly shaped clusters.
- **Affected by outliers** – Extreme values can skew the results.
- **Poor performance with high-dimensional data** due to the curse of dimensionality.

- **Does not handle categorical data** directly – Only suitable for numerical features.

POLYMORPHIC MALWARE DETECTION STRATEGIES

Polymorphic Malware: Complex file infectors that can create modified versions of itself to avoid detection yet retain the same basic routines after every infection.

Metamorphic Malware: Reprograms itself by translating its own code and then rewriting it to ensure that subsequent copies appear different with each iteration.

Polymorphic malware is a type of **self-altering malicious software** that changes its **code signature or appearance** every time it is executed, while retaining its original malicious functionality. This transformation helps the malware **evade signature-based detection systems**, such as traditional antivirus solutions.

Key features of polymorphic malware:

- Changes encryption keys and variable names regularly.
- Uses **code obfuscation** and **code mutation**.
- Employs **self-replication and payload encryption**.
- Common in **viruses, worms, Trojans, and ransomware**.

- Polymorphic malware uses techniques like encryption, obfuscation, and code mutation to change its appearance each time it infects a system, which makes it harder to detect using traditional signature-based methods.
- Signature-based detection is ineffective because the changes in the malware's code (such as altered file checksums) mean that each new variant is technically unique, even though the core malicious functionality remains the same.

- **Code emulation** is an effective strategy for detecting polymorphic malware. By **running the malware in a controlled, virtualized environment** (sandbox), it allows the malware to execute its code, revealing the decrypted payload for further analysis using traditional signature-based detection methods.
- **Behavior-based detection** analyzes the dynamic behavior of the malware during execution. It looks for patterns such as file modifications, registry changes, or network activity that indicate malicious intent, even if the malware's code changes in each iteration.

- **Machine learning (ML)** algorithms, including techniques such as **Hidden Markov Models (HMM)**, are used to detect polymorphic malware by learning from previous observed behaviors of malware variants. These algorithms can track and predict sequences of malicious behavior, even when the malware itself changes form.
- **De-obfuscation** techniques aim to reverse the changes made by polymorphic malware, such as encrypted or obfuscated code, to uncover the original functionality. This step is crucial in analyzing the malware's real behavior and identifying its malicious intent.
- **Code normalization** is a process that simplifies the complex and mutated code produced by polymorphic malware into a more recognizable and standardized format. By removing misleading or superfluous instructions, the malware can be analyzed in its true form.
- **Automated analysis frameworks** such as **Cuckoo Sandbox** and **Virustotal** offer dynamic analysis, enabling the rapid testing of malware samples in a secure environment. They help to identify malicious behavior patterns and provide insights into how polymorphic malware interacts with a system.
- **Heuristic analysis** is another strategy for identifying polymorphic malware by looking for suspicious patterns in the behavior of code that deviate from normal, known software behavior. It does not rely on known signatures, but instead detects anomalies or behaviors typical of malicious activities.
- **Statistical analysis** of system logs and network traffic can also help detect polymorphic malware by identifying unusual activity patterns, such as high network traffic or strange API calls, which might indicate the presence of an infection.
- **Artificial Intelligence (AI)** techniques, particularly **Deep Learning**, have been integrated into malware detection systems to better identify polymorphic variants. These systems can adapt to new threats by learning from large datasets of malware and their behaviors, even when the malware constantly mutates.

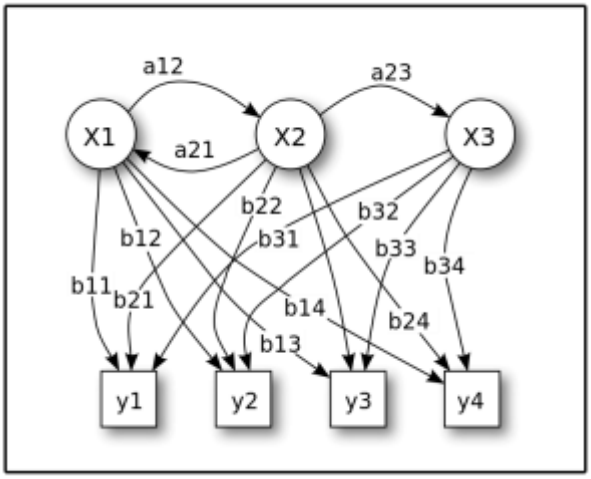
- Tools like **PEiD** are used to identify packers and protectors that polymorphic malware often employs to obscure its true nature. By detecting these tools, analysts can better understand how the malware is hiding its payload.
- Polymorphic malware poses challenges not only to traditional antivirus tools but also to newer detection systems, as it often employs **anti-sandboxing** techniques to detect when it is being analyzed in a controlled environment and alters its behavior to evade detection.
- The use of **hash-based detection** is limited against polymorphic malware, as each variant has a different hash. Instead, detecting malware behavior and network traffic patterns is becoming a more effective approach in identifying threats.
- The **evolution of attack techniques** means that polymorphic malware is continually adapting to bypass existing detection methods, leading to a need

for constantly evolving strategies that incorporate machine learning and behavior-based detection methods.

- **False positives** are a concern in behavior-based detection strategies, as legitimate software might also exhibit behavior similar to malware, leading to misidentification. Thus, refining these detection methods to improve accuracy is an ongoing challenge.

HMM FUNDAMENTALS

- Hidden Markov Models are a type of stochastic model.
- They are used as a strategy for detecting metamorphic malware and zero-day threats.
- An HMM is a **Markov process** where the state of the system is not directly observable.
- To understand what an HMM (Hidden Markov Model) is, we need to introduce Markov processes.
- A Markov process (or Markov chain) is a stochastic model that changes its status based on a predefined set of probabilities.
- One of the assumptions of the Markov process prescribes that the probability distribution of future states depends exclusively on the current state.
- The state of the system: the only observable elements are the events and secondary effects associated with the state of the system; however, the probabilities of the events being determined by each state of the system are fixed.
- Consequently, the observations on each state of the system are made indirectly on the basis of the events determined by such hidden states, to which probability estimates can be associated:



- To intuitively understand how HMMs work, we present the following example:
- Imagine an executable that is launched on a host machine. At a given time, the machine can continue to function properly, or stop working properly; this behavior represents the observable event.

- Let's assume for simplicity that the reasons why the machine stops working regularly can be reduced to the following:
    - The executable executed a malicious instruction
    - The executable executed a legitimate instruction
- The information relating to the specific reason why the machine stops working properly is the entity unknown to us, which we can only infer based on observable events.
- These observable events, in our example, are reduced to the following:
    - The machine works regularly (working)
    - The machine stops working (not working)
- Similarly, the hidden entities of our example are represented by the instructions executed by the program:
  1. Malicious instruction
  2. Legitimate instruction
- Finally, imagine assigning the probability estimates to the various events and states of the system.
- We summarize this in the following table, also known as the **emission matrix**, which summarizes the probabilities that a given observation is associated with a particular observable state (remember that the sum of the probabilities associated to each hidden entity, subdivided according to the possible events, must account to 1):

	Working	Not Working
Malicious	0.2	0.8
Legitimate	0.4	0.6

- At this point, we must estimate the probabilities associated with the next instruction executed by the program, which can be summarized in the **transition matrix**
- Therefore, if the program has previously executed a malicious (instead of a legitimate) instruction, the probability that the next instruction executed is malicious (rather than legitimate) is equal to the following:

Malicious	0.7	0.3
Legitimate	0.1	0.9

- Finally, we must assign the probability associated with the starting state of the HMM; in other words, the probability associated with the first hidden state corresponds to the probability that the first instruction executed by the program is malicious or legitimate:

Malicious	0.1
Legitimate	0.9

At this point, the task of our HMM is to identify hidden entities (in our example, if the instructions executed by the program are malicious or legitimate) based on the observation of the behavior of the machine.

- **Given Data as Matrices:**
- **Emission Matrix:**

	Working	Not Working
Malicious	0.2	0.8
Legitimate	0.4	0.6

- **Transition Matrix:**

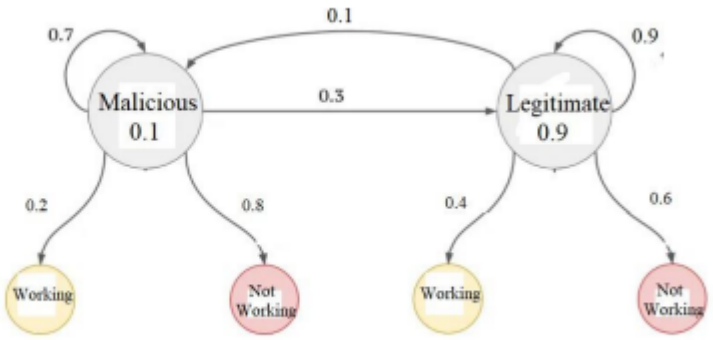
	Malicious	Legitimate
Malicious	0.7	0.3
Legitimate	0.1	0.9

- **Initial State S0 π:**

Malicious	Legitimate
0.1	0.9

- **State Transition Matrix:**

	Malicious	Legitimate
Initial State S0 π	**0.1**	**0.9**
Malicious	**0.7**	**0.3**
Legitimate	**0.1**	**0.9**



Generated Finite State Machine for HMM

- The only elements you can directly observe are the events and secondary effects associated with the system's state.
- The probabilities of these observable events being determined by each system state are fixed.
- Observations about each state are made indirectly based on the events triggered by the hidden states, and you can associate probability estimates with these hidden states.
- In a malware detection context, the hidden entities could represent the instructions being executed by a program, such as a malicious instruction or a legitimate instruction.
- The corresponding observable events might be whether the machine is working regularly or stops working.
- An emission matrix is used to summarise the probabilities that a given observation (e.g., 'Working' or 'Not Working') is associated with a particular hidden state (e.g., 'Malicious' or 'Legitimate'). For example, the probability of observing the machine 'Not Working' given a 'Malicious' instruction might be high.
- The sum of probabilities for each hidden entity, when broken down by the possible events, must equal 1.
- A transition matrix is used to estimate the probabilities associated with the *next* instruction executed by the program, based on the *current* instruction's state. For instance, if the current instruction is malicious, the probability that the next instruction is also malicious is defined in this matrix.
- You must also assign the initial probabilities for the starting state of the HMM, representing the probability that the very first instruction executed is malicious or legitimate.
- The primary task of an HMM is to identify these hidden entities (like whether an instruction was malicious or legitimate) by analysing a sequence of observable events (like the machine's behaviour). For example, observing a sequence where the machine works for a time then stops can help infer the nature of the executed instructions.

NETWORK ANOMALY DETECTION TECHNIQUES

The current level of interconnection between devices has reached such complexity that it challenges the effectiveness of traditional security concepts like perimeter security.

Cyberspace's attack surface is growing exponentially.

Therefore, resorting to **automated tools**, often leveraging Artificial Intelligence (AI), is essential for the effective detection of network anomalies associated with unprecedented cybersecurity threats.

Anomaly detection has always been a research area in cybersecurity, particularly for network security protection. It's also applicable to fraud detection and identifying compromised user profiles.

- **Evolution of Detection Approaches or Anomaly detection rationale:**
    - Historically, network intrusion detection has followed two main approaches: Signature-based detection and Anomaly detection.
    - **Signature-based detection** involves analyzing known attacks to build a knowledge base of attack signatures. An alert system triggers when network traffic matches an archived signature. This is analogous to traditional antivirus software. A key disadvantage is that the knowledge base must be constantly updated to detect new attack types.
    - **Anomaly detection**, conversely, attempts to identify network traffic behaviour defined as "normal" in order to detect differences that deviate from this norm as anomalous. This approach enables the detection of new types of attacks by analysing their characteristics that appear anomalous.
- **Identifying Anomalous Traffic:**
    - To detect anomalous traffic, several elements can be considered suspicious based on the analysis of what is deemed normal network traffic:
        - The number of connections to and from a specific host.
        - Unusual remote communication ports or unexpected traffic patterns.
        - Unusual traffic peaks occurring at particular times of the day (e.g., during the night).
        - Communication bandwidth heavily occupied by particular hosts within the network.

    - Appropriate filters can be defined, and alarm signals (alerts) can be associated with them, potentially leading to the dropping of corresponding network traffic.
    - It's necessary to account for novelties in network traffic that are *not* associated with suspicious behaviour (e.g., adding a new legitimate communication channel). A sensitive aspect is distinguishing between true positives and false positives.

- **Intrusion Detection Systems (IDS):**
    - Traditionally, intrusion detection has been managed by specialized devices called Intrusion Detection Systems (IDS).
    - Two traditional categories are:
        - Host-based IDS (HIDS) and
        - Network-based IDS (NIDS).
    - **HIDS** detect intrusions on host machines by monitoring significant system metrics like: number/type of running processes, user accounts, kernel module loading, file/directory activity, registry key modification, background processes, OS modules loaded at startup, and host network activity. The metrics monitored depend on the adopted threat model and can be collected using OS tools or specialized monitoring tools.
    - **NIDS** identify attack patterns by analysing network traffic (incoming and outgoing packets), detecting known patterns in data flow. Examples of attacks detected include Adware (resulting in unsolicited and malicious advertisements downloads from remote hosts) , Spyware (sensitive information transmitted toward remote hosts) , Advanced Persistent Threats (APTs) (APT-targeted attacks that leverage specific organization flaws or misconfigured services) , and Botnets (a typical Command and Control (C2) attack that leverages organization network resources by transforming hosts into zombie machines, executing remote instructions) . NIDS can use sniffers (like tcpdump, Wireshark) or integrated solutions like Snort for real-time detection. They define rules and triggers, often associated with thresholds, to compare normal and malicious traffic. However, attackers can evade detection by operating below a fixed threshold. Dynamic thresholds can be used.
    - **Stateful inspection** (Packet Filtering) is necessary given the complexity of network data flow. It correlates different packet types to identify connection attempts, Denial of Service (DoS) attempts, or attacks at lower protocol levels. Stateful inspection is associated with more sophisticated forms of anomaly detection.
    - With the introduction of AI, a third type, **anomaly-driven IDS**, has been added.

How to classify network attacks?

- We have seen that it is possible to use all different types of algorithms (such as supervised, unsupervised, and reinforcement learning), even in the implementation of network anomaly detection systems.
- But how can we effectively train these algorithms in order to identify the anomalous traffic?
- It will be necessary to first identify a training dataset that is representative of the traffic considered normal within a given organization.
- To this end, we will have to adequately choose the representative features of our model.
- The choice of features is of particular importance, as they provide a contextual value to the analyzed data, and consequently determine the reliability and accuracy of our detection system.
- In fact, choosing features that are not characterized by high correlation with possible anomalous behaviors translates into high error rates (false positives), which therefore invalidate their usefulness.
- A solution to choosing reliable features could be to evaluate existing anomalies in the use of network protocols.
- Attacks—such as SYN floods—are characterized by the anomalous use of the TCP/IP handshake (in which the packet with the SYN flag set is not followed by packets with the ACK flag set, in order to establish a valid connection).
- A feature can be characterized by one or more attributes related to the protocol or the header of the network packet, just as different types of network attributes constitute a feature represented by the specific network connection being analyzed (that is, a telnet session is characterized by a connection to a remote port 23, which is carried out between two endpoints having their respective IP addresses and IP ports).

NETWORK ATTACKS

Network security threats are a major concern in the current interconnected environment, where the attack surface grows exponentially. Automating detection procedures using Artificial Intelligence (AI) and Machine Learning (ML) is necessary to effectively deal with these threats, as analysis by human operators alone is practically impossible due to the scale and dynamism of attacks.

Identifying the most frequent types of network attacks is useful for determining which features to monitor for anomaly detection and for feeding algorithms with more representative datasets. This analysis should be adapted to the specific context and constantly updated.

Some of the most common network attacks include:

- **Malware-based attacks**: Malware can spread by compromising the integrity of files on machines. Detection strategies often involve evaluating suspect artifacts in binary files (static analysis) or analysing the dynamic behaviour of the code by executing it in a controlled environment (dynamic analysis). AI is necessary to automate the preliminary phase of malware analysis, known as triage. Malware comes in many forms, such as Trojans, Botnets, Downloaders, Rootkits, Ransomware, APTs, and Zero days. Malware can even be hidden in seemingly innocuous files like image files or text documents. Network monitoring can be used to identify anomalous connections established by host machines to remote destinations, which could indicate malware activity.
- **Zero-day exploits:** These exploit vulnerabilities that have not yet been disclosed to the security community, meaning their characteristics and security impacts are unknown, making them undetectable by traditional antivirus software. Detecting zero-days often requires analysing the behaviour of the suspect file or using anomaly detection techniques which can identify new types of attacks.
- **Data exfiltration via network sniffing**: This involves capturing network traffic to steal sensitive information. Network diagnostic tools including sniffers like tcpdump and Wireshark can be leveraged for NIDS implementation. Data leakage can also be a goal of botnets.
- **Saturation of network resources (DoS/DDoS)**: Denial of Service (DoS) or Distributed Denial of Service (DDoS) attacks attempt to saturate network resources. Stateful inspection can detect these attempts by correlating different types of packets. Botnets are often used to perform DDoS attacks towards target sites. Reactive alarm systems can be exploited by attackers simulating DoS attempts to cause legitimate user accounts or IP addresses to be automatically blocked, damaging the organisation's reputation.
- **Session hijacking**: This consists of abusing a user session that was legitimately initiated. Detecting this can be difficult, especially if the attacker has obtained the user's password through other means. Monitoring account activity can help identify such threats.
- **Connection spoofing**: This involves creating network packets with a false source IP address to impersonate another device or user.
- **Port scanning**: This is often a reconnaissance technique where an attacker scans a network or host to find open ports and identify potential vulnerabilities or services running.

Network anomaly detection systems, leveraging AI, aim to identify behaviours that deviate from what is considered normal network traffic, enabling the detection of new attack types that signature-based systems might miss. Elements considered for detecting anomalous traffic include the number of connections, unusual ports or patterns, unusual traffic peaks, and bandwidth usage. This involves distinguishing

between true anomalies (potential threats) and false positives (normal, but unusual, behaviour).

Anomaly Detection strategies

We have therefore seen that the very concept of anomaly detection refers to a behavior that is different from what was expected; this difference, in technical terms, translates into outlier detection.

To identify the outliers, it is possible to follow different strategies:

- **Analyzing a sequence of events within a time series**: The data is collected at regular intervals, evaluating the changes that occur in the series over time. This is a technique widely used in the analysis of financial markets, but it can be also validly used in the cybersecurity context to detect the frequency of characters (or commands) entered by the user in a remote session. Even the simple unnatural increase in the frequency of data entered per unit of time is indicative of an anomaly that can be traced back to the presence of an automated agent (instead of a human user) in the remote endpoint.
- **Using supervised learning algorithms**: This approach makes sense when normal and anomalous behaviors can be reliably distinguished from each other, as in the case of credit card fraud, in which it is possible to detect predefined patterns of suspicious behavior, relying on the fact that future fraud attempts are attributable to a predefined scheme.
- **Using unsupervised learning algorithms**: In this case, it is not possible to trace the anomalies back to predefined behaviors, as it is not possible to identify a reliable and representative training dataset for supervised learning. This scenario is the one that most commonly describes the reality of cybersecurity, characterized by new forms of attack or exploits of new vulnerabilities (zero-day attacks). Similarly, it is often difficult to trace all the theoretically possible intrusions back to a single predefined scheme.

DETECTING BOT-NET TOPOLOGY

Detecting botnet topology is a particularly relevant aspect of network anomaly detection due to the dangers they pose to an organisation. Identifying the presence of a botnet in a timely manner is often a complex operation.

What is a Botnet?

- The term botnet comes from the words **'bot'** and **'net'**. 'Net' relates to networking, while 'bot' is increasingly associated with the spread of automated

agents in cyberspace. This includes chatbots or 'trolls' aimed at spreading false information.
- In the context of botnets, the attacker's goal is to compromise many hosts in a network, often by installing malware, turning them into automated agents (zombie machines) that carry out instructions received from the attacker via a Command and Control (C2) console, typically managed by a centralised server.
- The compromised machine becomes part of a large network of infected machines, contributing its computational and network resources towards common goals set by the attacker.

Activities and Risks Associated with Botnets

Botnets can participate in various malicious activities, including:

- Taking part in email spamming campaigns.
- Performing Distributed Denial of Service (DDoS) attacks towards target sites.
- Bitcoin and Cryptocurrency mining.
- Password cracking.
- Credit card cracking.
- Data leakages and data breaches (dissemination of sensitive information).

For an organisation, dealing with a botnet, even unconsciously, represents a serious risk. This includes the exhaustion of computational and network resources by external attackers and significant legal responsibility towards third parties, not just a waste of company resources.

Botnet Detection Challenges and Techniques

Identifying the presence of a botnet is often difficult. Automated detection procedures are essential due to the complexity and scale of threats. Analysing which types of network attacks are most frequent, such as botnets, can help determine which features to monitor for anomaly detection and provide more representative datasets for algorithms. This analysis should be adapted to the specific context and constantly updated.

In order to promptly identify the possible presence of a botnet, it may be useful to consider its kill chain (the different phases that characterize its realization).

We can, therefore, distinguish the following phases:

- Malicious software installation
- Joining the botnet via C2
- Spreading the botnet to other hosts

Key aspects and techniques for detecting botnet topology mentioned in the sources include:

- **Monitoring the Botnet Kill Chain:** It can be useful to consider the botnet kill chain, which includes phases like malicious software installation, joining the botnet via C2, and spreading the botnet to other hosts.

- **Focus on C2 Communication and Beaconing:** A crucial event to monitor for the possible presence of a botnet is connections made at regular intervals to remote hosts. The victim host must constantly communicate ("call home") to the C2 server to receive orders and send gathered information. This phenomenon is known as **beaconing**.

- **Characteristics of Beaconing:** Beaconing is characterised by the presence of regular connections within the network, even during closing hours, between infected hosts and remote destinations (which can include compromised legitimate websites). Typical characteristics include:

    - Long-term user sessions with the exchange of empty packets (keepalive packets) to keep the connection open.
    - Data exchange between hosts on a regular basis.
- **Detection Strategies:**
    - **Network Monitoring:** Monitoring the company network to promptly identify hosts that might be part of a botnet is important.
    - **Investigating Connection Nature:** Rather than just monitoring the traffic quality (botnets often use seemingly harmless protocols like HTTP on port 80 to mask their presence), the real nature of the network connections should be investigated.
    - **Network Intrusion Detection Systems (NIDS):** Botnets are listed as a typical Command and Control (C2) attack detected by NIDS. NIDS work by analysing network traffic and detecting known attacking patterns. NIDS implementation can leverage network diagnostic tools, including sniffers like tcpdump and Wireshark, and integrated software solutions like Snort.
    - **Anomaly Detection:** Identifying network traffic behaviour that deviates from what is considered normal is crucial. Elements considered for detecting anomalous traffic indicative of botnets or other threats include the number of connections to/from a host, unusual remote communication ports or patterns, unusual traffic peaks, and bandwidth usage.
    - **Stateful Inspection:** This technique, also known as Packet Filtering, keeps track of packets and correlates different types to identify connection attempts or attempts to saturate network resources (DoS).

It can be associated with more sophisticated forms of anomaly detection.
    - **Statistical Analysis:** Detecting beaconing requires in-depth network traffic monitoring along with statistical analysis of a time series and calculating position measures (like the median and IQR) to spot communications occurring regularly.
    - **Graphical Visualisation:** Graphically visualising the mapping of local and remote hosts exhibiting regular connections can help identify a possible network topology with stable characteristics that could indicate a botnet.
    - **AI and ML Algorithms:** AI techniques, including ML algorithms like supervised and unsupervised learning, reinforcement learning, and deep learning, can be used to evolve traditional IDS into more advanced anomaly detection solutions. Clustering techniques, which exploit the concept of similarity, can be used for anomaly-based IDS implementation. Identifying similarities in malware behavior, which can be carried out in an automated form using clustering algorithms, is increasingly important. AI is necessary to automate the preliminary phase of malware analysis (triage).
- **Difficulty in Reliable Identification:** The problem with beaconing is that it cannot always be identified reliably as it can resemble legitimate services (e.g., benign SSH or telnet sessions, antivirus updates). This leads to a high risk of false positives, especially with the increasing number of interconnected devices like IoT.

Three Common Types of Malware and Their Applications:

1. **Virus:**
    - **Description**: A virus attaches itself to legitimate programs or files and spreads when the file is run.
    - **Application**:
        - Often used to **corrupt files**, **delete data**, or **slow down system performance**.
        - Example: **ILOVEYOU virus** — it spreads via email and overwrite files, causing system crashes.
2. **Worm:**
    - **Description**: A self-replicating program that spreads across networks without needing to attach to files.
    - **Application**:
        - Used to **consume bandwidth**, **crash networks**, or **install backdoors**.
        - Example: **Blaster worm** exploited vulnerabilities in Windows systems and caused repeated reboots.

3. **Trojan Horse:**
    - **Description**: Malware disguised as legitimate software. It does not replicate but creates a backdoor for attackers.
    - **Application**:
        - Used to **steal sensitive information**, **log keystrokes**, or **gain remote access**.
        - Example: **Zeus Trojan** was used to steal banking credentials and perform financial fraud.

Difference Between Static and Dynamic Malware Analysis:

Feature	Static Malware Analysis	Dynamic Malware Analysis
Definition	Analyzing malware **without executing** it.	Analyzing malware by **executing it in a controlled environment**.
Method	Code inspection, disassembling, decompiling.	Running in sandbox, monitoring behavior & system calls.
Tools Used	Hex editors, disassemblers (e.g., IDA Pro), strings.	Sandboxes, debuggers, monitoring tools (e.g., Cuckoo).
Speed	Generally faster as it doesn't require execution.	Slower due to real-time execution.
Safety	Safer – no execution, hence no infection risk.	Riskier – malware actually runs (in isolated setup).
Limitations	Cannot detect obfuscated or packed malware easily.	Might miss time-triggered or condition-based payloads.
Use Case	Used in initial triage or when code is available.	Used when behavior needs to be observed or verified.

Maximum Margin Hyperplane (MMH)

- The **Maximum Margin Hyperplane** is a key concept in **Support Vector Machines (SVM)** used for classification tasks in machine learning.
- The **maximum margin hyperplane** is the **decision boundary** that **best separates** the data points of different classes **with the widest possible margin**.
- It is the **optimal separating hyperplane** that maximizes the **distance between itself and the nearest data points** from each class. These nearest points are called **support vectors**.

🔹 Mathematical Explanation:

- For a linearly separable dataset, the hyperplane is defined by:
  $w \cdot x+b=0$
  where:
  - w = weight vector (normal to hyperplane)
  - b = bias (offset)
- The margin is:
  $2||w|$
  and MMH is found by minimizing $||w||||\mathbf{w}||||w||$ subject to the constraint that each data point is correctly classified with a margin ≥ 1.

🔹 Why Maximize the Margin?

- A larger margin leads to **better generalization** on unseen data.
- It helps reduce **overfitting**, making the model more robust.

🔹 Visual Understanding:

- Imagine two parallel lines (margins) on either side of the hyperplane. MMH keeps the largest possible distance between these two lines while still separating the classes.

🔹 Support Vectors:

- Only the data points that lie on the margin boundaries affect the position of the hyperplane.
- These critical points are called **support vectors**.

Different Methods to Classify Network Attacks:

Network attacks can be classified based on **various criteria**:

1. Based on Attack Origin:

- **Active Attacks**: Modify, disrupt, or destroy data.
  Example: Man-in-the-middle, Denial of Service (DoS).
- **Passive Attacks**: Only eavesdrop or monitor traffic.
  Example: Packet sniffing, traffic analysis.

2. Based on Attack Type:

- **DoS (Denial of Service)**: Overloads system/network to make it unavailable.
- **DDoS (Distributed DoS)**: Same as DoS but from multiple sources (often botnets).
- **Phishing**: Tricking users to reveal sensitive info using fake websites/emails.
- **Spoofing**: Impersonating another device or user on the network.
- **MITM (Man-in-the-Middle)**: Intercepts communication between two parties.

3. Based on Target Layer (OSI Model):

- **Network Layer Attacks**: IP spoofing, ICMP flooding.
- **Transport Layer Attacks**: SYN flood, port scanning.
- **Application Layer Attacks**: SQL injection, cross-site scripting (XSS).

4. Based on Attacker's Intent:

- **Reconnaissance Attacks**: Gathering information (e.g., port scanning).
- **Access Attacks**: Gaining unauthorized access (e.g., password cracking).
- **Control Attacks**: Taking control of systems (e.g., installing rootkits).

What is a Botnet?

- A **botnet** is a **network of compromised computers (called bots or zombies)** controlled remotely by an attacker (called botmaster).
- Bot + Network
- These bots are often infected via malware and work together to perform:
  - **DDoS attacks**
  - **Spam email campaigns**
  - **Data theft**
  - **Cryptomining**
- **Famous Example**: **Mirai Botnet** – took down major websites by launching DDoS attacks.

What is Metamorphic Malware?

- **Metamorphic malware** is an advanced type of malware that **rewrites its own code** each time it infects a new system, without changing its external behavior or function.

- Unlike polymorphic malware (which only changes its encryption), **metamorphic malware changes its actual code structure**, making it extremely hard to detect using traditional signature-based methods.

Key Characteristics of Metamorphic Malware:

- Uses **code obfuscation**, junk code insertion, register renaming, control flow alteration, etc.
- Every new variant **looks different** in code but **acts the same**.
- **Examples**: Simile, ZMist.

Hidden Markov Model (HMM) for Detecting Metamorphic Malware

What is HMM?

- A **Hidden Markov Model (HMM)** is a **statistical model** where the system being modeled is assumed to be a Markov process with **hidden (unobservable) states**.

- HMM is widely used in **pattern recognition**, including speech, handwriting, and now **malware detection**.

How HMM Helps Detect Metamorphic Malware:

1. **Training Phase:**
    - **Collect known metamorphic malware samples**.
    - Disassemble the malware binaries to get **opcode sequences** (like `MOV`, `ADD`, `JMP`, etc.).
    - Train an HMM on these opcode sequences to **learn the probabilistic transitions** between opcodes.
    - The HMM now represents the "behavior pattern" of metamorphic malware.
2. **Testing Phase:**
    - Take a suspicious program and extract its **opcode sequence**.
    - Calculate the **likelihood (probability)** that this sequence was generated by the trained HMM.
    - If the likelihood is **high**, the program is likely to be metamorphic malware.
3. **Advantage of HMM:**
    - **Does not depend on fixed signatures.**
    - Focuses on **behavioral patterns and statistical similarity**, even if the code looks different.
    - Effective against obfuscated and transformed code.