FITPED | AI

Advanced Artificial Intelligence Techniques in Cyber Threat Detection

Cyril Klimeš
Ján Skalka
Peter Švec
Tomáš Sochor
Jiří Balej
Jan Francisti

www.fitped.eu

2024

Erasmus+ FITPED-AI
Future IT Professionals Education in Artificial Intelligence
(Project 2021-1-SK01-KA220-HED-000032095)

FITPED | AI

Advanced Artificial Intelligence Techniques in Cyber Threat Detection

Published on

November 2024

Authors

Cyril Klimeš | Mendel University in Brno, Czech Republic

Ján Skalka | Constantine the Philosopher University in Nitra, Slovakia

Peter Švec | Teacher.sk, Slovakia

Tomáš Sochor | Mendel University in Brno, Czech Republic

Jiří Balej | Mendel University in Brno, Czech Republic

Jan Francisti | Constantine the Philosopher University in Nitra, Slovakia

Reviewers

Piet Kommers | Helix5, Netherland

Małgorzata Przybyła-Kasperek | University of Silesia in Katowice, Poland

Vladimiras Dolgopolovas | Vilnius University, Lithuania

Erasmus+ FITPED-AI

Future IT Professionals Education in Artificial Intelligence

Project 2021-1-SK01-KA220-HED-000032095

Funded by
the European Union

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or Slovak Academic Association for International Cooperation. Neither the European Union nor the granting authority can be held responsible for them.

ISBN 978-80-558-2234-1

TABLE OF CONTENTS

Spam Detection

Chapter 1

1.1 Spam

🕮 1.1.1

Spam, or junk mail, refers to unsolicited communication sent in bulk, primarily via email but also through text messages, social media comments, or phone calls. Understanding spam is crucial in today's digital landscape, where the prevalence of unwanted messages can hinder effective communication.

Spam messages are typically characterized by their unsolicited nature; recipients do not anticipate receiving these messages and have not requested them. Additionally, spam is marked by mass distribution, often reaching a vast audience simultaneously. Many spam messages contain manipulative content designed to prompt the recipient to take action, such as clicking on a link or providing personal information.

1.1.2

What is a characteristic of spam?

- It is always welcomed by the recipient.
- It is sent in bulk to many recipients.
- It is personalized for each recipient.
- It is typically sent by friends or family.

🕮 1.1.3

Spam can be categorized into various types, each with distinct characteristics and purposes.

The most common type is **advertising spam**, which promotes products or services without the recipient's consent.

Phishing spam, on the other hand, aims to deceive recipients into divulging sensitive information like bank account details or credit card numbers.

Lastly, **malware spam** includes harmful software, such as viruses or spyware, often disguised as legitimate messages to trick recipients into clicking on links.

Understanding these types helps users recognize and avoid potential threats associated with spam.

1.1.4

Select the types of spam from the options below:

- Advertising spam
- Phishing spam

- Malware spam
- Genuine customer feedback

📖 1.1.5

Engaging with spam can lead to significant risks for individuals and organizations.

One major risk is the **spread of malware**, which can compromise a recipient's computer system, leading to data loss or theft.

Phishing attacks, another serious concern, **manipulate recipients** into sharing sensitive information, which can result in identity theft and financial loss.

Additionally, spam can **consume valuable time** and reduce productivity. The process of sifting through and deleting spam messages can distract users from more critical tasks. Furthermore, a high volume of spam can erode trust in email as a reliable communication method, making individuals wary of legitimate messages.

📝 1.1.6

Spam can lead to the spread of _____ and increase the risk of _____. Additionally, it can cause a loss of _____ as users spend time deleting unwanted messages.

- malware
- phishing attacks
- productivity

📖 1.1.7

Recognizing spam is essential for maintaining a secure online environment. Common indicators of spam include generic greetings, unsolicited offers, and urgent language urging immediate action. Users should be cautious of messages that contain suspicious links or attachments, as these are often tools used by spammers to compromise personal information or install malware.

Being able to differentiate between legitimate communications and spam can help users protect their personal information and maintain their digital security. It’s also vital to report spam to help improve filtering systems and reduce its prevalence in digital communication channels.

📝 1.1.8

Which of the following is an indicator of spam?

- Generic greetings and unsolicited offers.
- Personalized greetings.
- Messages from known contacts.
- Requests for feedback.

📖 1.1.9

Ignoring spam can lead to a range of negative consequences. For individuals, the most immediate impact is the potential compromise of personal information. Engaging with spam can inadvertently provide spammers with access to sensitive data, leading to identity theft or financial fraud. For organizations, a spam attack can result in data breaches, loss of customer trust, and legal repercussions.

Moreover, the accumulation of spam can lead to system slowdowns, as excess messages clog email servers and disrupt normal operations. It is essential for users to be proactive in managing spam to mitigate these risks effectively.

📝 1.1.10

Which of the following are potential consequences of ignoring spam?

- Compromised personal information
- Data breaches in organizations
- Increased productivity
- Enhanced system performance

📖 1.1.11

To combat spam effectively, users should implement best practices for managing their digital communications. One essential practice is to use spam filters, which automatically detect and move suspicious messages to a separate folder. Regularly updating email settings to enhance privacy and security can also help reduce the influx of spam.

Users should also be cautious about sharing their email addresses online and consider using secondary email accounts for subscriptions or less important communications. Educating oneself about the latest spam techniques and remaining vigilant can significantly decrease the likelihood of falling victim to spam-related threats.

📝 1.1.12

To manage spam effectively, users should use _____ and regularly update their _____. It's also advisable to limit sharing of email addresses and stay informed about the latest _____.

- privacy settings
- spam techniques
- spam filters

📖 1.1.13

As technology evolves, so do the tactics used by spammers. Emerging technologies, such as AI, are being utilized to create more sophisticated spam that can bypass traditional filters. This ongoing battle between spammers and cybersecurity professionals means that awareness and education are more critical than ever.

In the future, users may need to adopt new tools and strategies to combat spam effectively. Keeping abreast of the latest developments in cybersecurity and participating in training programs can empower individuals and organizations to stay ahead of potential threats posed by spam.

📝 1.1.14

What is a potential future challenge in combating spam?

- Increased use of AI by spammers.
- The decline of email usage.
- The complete eradication of spam.
- Simpler spam detection methods.

1.2 Spam detection

📖 1.2.1

Detecting spam is a fundamental task in digital security, with implications for both users and email providers. Spam detection involves an imbalance in potential costs. For example, when a spam email is misclassified as legitimate, the user only needs to delete it. However, when a legitimate email is classified as spam, the user may lose important information or waste time checking the spam folder for valid messages.

The approach to spam detection often begins with simple criteria. If a message matches known spam indicators, it is flagged as spam. However, defining these criteria is complex due to evolving spam tactics. Unlike early approaches that flagged basic phrases like “cheap products,” modern spam detection considers factors like message structure, attachments, and sender address.

📝 1.2.2

Which of the following best describes an "unbalanced cost" in spam detection?

- The higher cost of recovering important emails marked as spam.
- The inconvenience of manually deleting a legitimate message.
- The preference to flag emails rather than delete them immediately.
- The decision to use machine learning algorithms for spam detection.

📖 1.2.3

Although defining spam might seem straightforward, the process is highly challenging. Spam content changes frequently, adapting to avoid detection. Spammers use techniques like replacing characters in words (e.g., “V1agra” instead of “Viagra”) or adding invisible characters to hide their messages.

Today, spam detection uses multicriterial classification, which considers various message aspects such as the presence of attachments, risk scores, and overall structure. Messages often receive a cumulative score, and if they exceed a specific threshold, they are classified as spam. This score-based approach helps adapt to evolving spam techniques and reduces the likelihood of misclassification.

📝 1.2.4

Which of the following tactics are commonly used by spammers to avoid detection?

- Adding invisible characters in words
- Frequently changing sender addresses
- Using exact phrases like "cheap V1agra"
- Using verified sender authentication

📖 1.2.5

The Simple Mail Transfer Protocol (SMTP) dialog plays a central role in email transmission. During this process, the sender’s and recipient's addresses are exchanged, which the receiving server can use for preliminary spam filtering. Blocking messages based on the sender’s address is possible, but ineffective alone because spammers can easily alter the sender address, bypassing basic filters.

To improve detection, advanced techniques like greylisting are used. **Greylisting** temporarily rejects messages from unfamiliar senders. If the sender attempts to resend after a delay, the server considers the message as more legitimate, reducing the likelihood of it being spam. Simple spam systems often don’t retry, allowing greylisting to filter them out effectively.

📝 1.2.6

During an SMTP dialog, the _____ and recipient email addresses are exchanged. Techniques like _____ reject emails from unknown addresses temporarily.

- sender
- greylisting

🕮 1.2.7

Verification Techniques

Sender verification techniques such as DKIM, SPF, and DMARC help to ensure the authenticity of email messages.

DomainKeys Identified Mail (DKIM) is an email authentication method that allows the receiver to check that an email was actually sent and authorized by the owner of that domain. It uses a digital signature based on public-key cryptography, which is added to the email's header. Here's how it works:

- When an email is sent, DKIM signs specific parts of the message with a unique hash (signature) that's encrypted with a private key held by the sending domain.
- The receiving server can use the sender's public key, stored in the DNS record, to verify the integrity of the email.
- If the signature matches the email content, the email is deemed authentic. Any modifications to the email in transit would invalidate the signature, signaling that the email might have been tampered with.

Sender Policy Framework (SPF) helps prevent spammers from sending unauthorized emails on behalf of your domain. It does this by defining which IP addresses are allowed to send emails from your domain. Here's how SPF works:

- A domain's DNS records contain a list of authorized mail servers allowed to send emails on its behalf.
- When an email is received, the recipient's server checks the DNS SPF record to verify that the sending IP address is authorized.
- If the IP address matches the SPF record, the email passes the SPF check. If it doesn't, the email may be flagged as suspicious or rejected

Domain-based Message Authentication, Reporting, and Conformance (DMARC) builds on both DKIM and SPF, giving domain owners control over what happens to emails that fail these checks. DMARC provides a policy framework that defines how receiving servers should handle emails that fail SPF or DKIM validation, and it offers a reporting feature for visibility. Here's how DMARC works:

- The domain owner sets a DMARC policy in their DNS record that instructs receiving servers on what to do if an email fails SPF or DKIM validation (e.g., reject, quarantine, or allow but mark as suspicious).
- DMARC records also specify where to send reports, allowing the domain owner to monitor any unauthorized email activity.
- By combining SPF and DKIM with a specified policy, DMARC provides a more robust defense against email spoofing and phishing, offering greater security and accountability.

1.2.8

Which of the following sender verification techniques uses cryptographic keys to confirm that the message content hasn't been altered?

- DKIM
- SMTP
- SPF
- DMARC

1.2.9

After message delivery, the receiving server assesses whether the email should go to the inbox or the spam folder. In some cases, messages flagged as high-risk (like those containing malware) are quarantined. In quarantine, emails may be kept for further inspection or placed in the spam folder with warnings. This delay gives time to evaluate the email's legitimacy and, if necessary, allow retrieval if the message was wrongly flagged.

Post-Delivery Filtering

Post-delivery filtering is a process applied to emails after they've been accepted and initially delivered to the recipient's inbox. Here's how it works:

- **Behavioral Analysis**: Even after delivery, emails can be analyzed for suspicious behavior based on user interaction. For instance, if users frequently mark a particular email as spam or a phishing attempt, the system can flag future emails from this source or with similar content.
- **Content and URL Re-scanning**: Some emails may include links or attachments that initially seem safe but are later identified as malicious. Post-delivery filtering continually re-scans these elements against updated threat intelligence databases, blocking access to newly detected harmful links or attachments.
- **Machine Learning & AI Detection**: Using machine learning algorithms, post-delivery filtering systems can detect unusual patterns in email content, structure, or sender behavior, refining their detection based on the collective interactions of all users and patterns of known spam emails.
- **Flagging & Moving**: If an email is detected as spam or risky post-delivery, it can be automatically flagged and moved from the inbox to a spam or quarantine folder. This minimizes potential exposure without needing user intervention.

Quarantine Methods

Quarantine methods involve holding suspicious emails in a secure area rather than delivering them directly to the recipient's inbox. This adds an additional layer of

security by isolating potentially harmful emails until they can be thoroughly evaluated. Quarantine typically offers the following processes:

- **Initial Isolation**: Instead of immediately delivering all emails to user inboxes, emails flagged with a high-risk score or unknown sender details are directed to a quarantine area. This isolates potentially dangerous content, such as malware or phishing links, from users while allowing security teams to investigate.
- **Review and Release**: The email remains in quarantine for a certain period (often set by system administrators). During this time, users or administrators can review the quarantined emails and decide whether to allow delivery, block the sender, or permanently delete the email. In some cases, users may receive daily or weekly summaries of quarantined emails, allowing them to request specific emails if needed.
- **Automated Re-assessment**: Quarantined emails are periodically re-assessed based on updated threat intelligence (e.g., new malware signatures). If an email is cleared of suspicion, it may be automatically delivered to the user's inbox with a "Spam" label or similar warning.
- **Notifications and Reporting**: Many quarantine systems notify users or administrators if an email they expect was quarantined, especially if it has a high potential risk. Quarantine methods also provide reports on blocked, delivered, and flagged emails, helping security teams monitor and refine their filtering processes.

These post-delivery and quarantine methods enhance security by providing continuous monitoring, minimizing the chance of false positives, and giving recipients a way to recover legitimate emails that may have been incorrectly flagged.

1.2.10

Which of the following actions might a receiving server take with a message detected as high-risk?

- Quarantine the message for review
- Deliver it to the inbox with a warning
- Delete it immediately
- Block the sender permanently without notice

Spam Detection Projects

Chapter 2

2.1 Simple methods

🕮 2.1.1

Simple methods

- Keyword filtering: This method blocks emails containing specific words or phrases typical of spam messages. Although simple to implement, it is not very effective because spammers easily adapt and use different variations of words.
- Blacklisting: Sending emails from known spam addresses or domains is blocked. However, maintaining an up-to-date list is difficult and spammers can easily change addresses.
- Header Filtering: Email headers are analyzed and look for discrepancies such as wrong sender domain or invalid IP address. This method is effective in blocking some types of spam, but is not reliable in detecting more sophisticated techniques.

Statistical methods

- Naive Bayes classifier: It is a probabilistic model that calculates the probability that an email is spam based on the frequency of occurrence of individual words in spam and legitimate messages. This method is simple and relatively effective, but it does not take into account context and word order.
- TF-IDF: This method assigns a weight to each word according to its frequency in the email and the inverse frequency in the entire dataset. Words that occur frequently in spam emails and rarely in legitimate emails will be given a higher weight. Subsequently, these weights are used to train the classification model.

Methods based on AI

- Machine Learning: Various machine learning algorithms such as Support Vector Machines, Random Forest, Naive Bayes and Neural Networks are used to train models that can recognize spam. These models learn from large datasets of emails labeled "spam" or "ham" and can identify complex patterns and characteristics of spam. Procedure: To implement machine learning for spam detection, you need:
- Get a dataset of emails labeled "spam" and "ham".
- Clean and preprocess the data (removal of punctuation, stop words, lemmatization, etc.).
- Extract numeric attributes from emails (eg TF-IDF, word count, etc.).
- Train a machine learning model on prepared data.
- Evaluate the performance of the model on the test dataset.
- Implement the model in the spam filter.
- Natural Language Processing (NLP): NLP techniques are used to analyze the text of emails and identify semantic meaning. This makes it possible to more

accurately distinguish spam from legitimate emails, which may contain similar words or phrases.
- Anomaly detection: This method identifies emails that deviate from normal behavior, such as having unusual word frequency, structure or sender.

🕮 2.1.2

Keyword filtering

Keyword filtering is one of the most basic and simple approaches to identifying spam. This method works on the principle of blocking emails that contain specific words or phrases typically associated with spam messages.

How keyword filtering works

- Creating a list of keywords: First, you need to create a list of words and phrases that are typical for spam. This list may contain words such as "free", "money", "win", "medicine" and the like.
- Email scanning: Received emails are scanned and searched for keywords from the list.
- Email blocking: If the email contains one or more keywords, the system will automatically block it and move it to the spam folder or delete it completely.

Advantages

- Simplicity: The implementation of keyword filtering is relatively simple and does not require complex algorithms.
- Speed: Scanning emails and identifying keywords is a fast process.

Disadvantages

- Low accuracy: Spammers adapt easily and use different variations of words to avoid detection. Therefore, this method can be ineffective and block even legitimate emails that contain some of the keywords.
- False alarms: Keyword filtering can lead to false alarms when legitimate emails containing some of the keywords are incorrectly classified as spam.
- Static list: The list of keywords needs to be constantly updated to catch new spam techniques. This requires manual work and is not very efficient.

Specific examples of "spam" keywords are often given, such as "free", "money", "product" and the like. These words often appear in spam emails, but they can also appear in legitimate emails, which reduces the effectiveness of keyword filtering.

Keyword filtering is a basic spam detection method that has its limitations. Currently, it is not used on its own, but rather as part of more complex antispam systems that combine various methods, including statistical methods and artificial intelligence techniques.

2.1.3

Project: Keyword filtering I.

Build a spam filter that can identify spam messages based on specific words or phrases.

Follow these steps:

Import libraries

```
import pandas as pd
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
```

Data Collection:

- Find a dataset that includes examples of spam and non-spam messages. You can use online databases or create your own dataset.
- **Load data** - load the dataset into the DataFrame. You can use https://www.kaggle.com/datasets/balaka18/email-spam-classification-dataset-csv

```
# load data
```

Data Preprocessing:

Clean the data by:

- Removing any accents or special characters.
- Converting all text to lowercase.
- Breaking the text into individual words (this process is called tokenization).

```
# use separate functions
```

Create a List of Keywords:

- Analyze the spam messages to identify common words or phrases.
- Make a list of these keywords (e.g., "free," "win," "click here," "increase your sales").

```
# List of keywords
keywords = ['free', 'click', 'win', 'money', 'guarantee']
```

Implement the Filter:

- Write a function that checks each message against your list of keywords.
- If a message contains at least one keyword from your list, mark it as spam.

```
def isSpam:

```

Evaluation:

- Test how well your spam filter works by calculating metrics like accuracy, recall, and F1 score.
- Analyze the results to find ways to improve your filter.

Visualize results with graphs.

```
import matplotlib.pyplot as plt
```

2.1.4

Project: Keyword filtering II.

Build a more advanced spam filter that can identify spam messages based on specific words or phrases, while also considering the importance of those words and analyzing email headers.

Data Collection:

- Find a dataset that includes examples of spam and non-spam messages. You can use online databases or create your own dataset.

Data Preprocessing:

Clean the data by:

- Removing any accents or special characters.
- Converting all text to lowercase.
- Breaking the text into individual words (this process is called tokenization).

Create a List of Keywords:

- Analyze the spam messages to identify common words or phrases.
- Make a list of these keywords (e.g., "free," "win," "click here," "increase your sales").

Weighting Keywords:

- Assign a weight to each keyword based on how likely it is to indicate spam. For example, the word "million" might have a higher weight than "free."

Use techniques like TF-IDF (Term Frequency-Inverse Document Frequency):

- Implement TF-IDF to automatically calculate the weight of each keyword based on its frequency in spam messages relative to its occurrence in the entire dataset. This helps highlight more significant terms.

Implement the Filter:

- Write a function that checks each message against your list of keywords.
- For each keyword found, add its corresponding weight to a total spam score for that message.
- If the total spam score exceeds a certain threshold, mark the message as spam.

Evaluation:

- Test how well your spam filter works by calculating metrics like accuracy, recall, and F1 score.
- Analyze the results to find ways to improve your filter.

```
from sklearn.feature_extraction.text import TfidfVectorizer
# ...
```

🕮 2.1.5

Blacklisting

Blacklisting is a simple and straightforward method to identify and filter spam. It works on the principle of creating and maintaining a list (blacklist) of known spam addresses and domains. Received emails are compared to this list and if a match is found, the email is marked as spam and blocked.

Blacklisting process:

1. Creating a blacklist: A blacklist can be created manually, automatically or by a combination of both methods.

- Manual blacklists are managed by humans and contain addresses and domains that have been identified as sources of spam.
- Automatic blacklists are generated by algorithms that analyze emails and identify characteristics typical of spam.

- Publicly available blacklists are managed by organizations that specialize in spam detection.

2. Comparison of emails with the blacklist: Received emails are compared with the blacklist. This comparison is usually done based on the sender's email address or their domain.

3. Spam blocking: If a match is found, the email is marked as spam and blocked. Blocking can mean moving an email to a spam folder, deleting it or rejecting delivery.

Advantages

- Ease of implementation: Blacklisting is a relatively simple method to implement and does not require complex algorithms or extensive computing resources.
- Effectiveness: Blacklisting is an effective way to block spam from known sources.
- Speed: Checking emails against the blacklist is a fast process.

Disadvantages

- Limited effectiveness: Blacklisting is only effective against spam from known sources. New addresses and domains that are not yet in the blacklist are not blocked.
- False alarms: Blacklisting can lead to false alarms if a legitimate sender is accidentally blacklisted.
- Need to be updated: The blacklist must be updated regularly to be effective.
- Bypassing blacklists: Spammers are constantly trying to bypass blacklists and are developing new techniques to mask their addresses and domains.

2.1.6

Project: Blacklist

Develop a Python program that identifies spam emails by comparing the sender's email address with a predefined list (blacklist) of known spam sources.

Follow these steps:

Creating a blacklist

- Create or download a text file or database that contains a list of email addresses and domains that are known to be sources of spam.
- Use publicly available blacklists or create your own based on spam email analysis.
- Make sure the blacklist is structured clearly, with one entry per line for easy reading and processing.

```
# you can find black-list at
http://www.joewein.de/sw/blacklist.htm
# you can find emails with address at
https://www.kaggle.com/datasets/nanditapore/spam-email-dataset
```

Load the blacklist:

- Write a Python function that retrieves a blacklist from a text file or database.
- Store the retrieved blacklist in a suitable data structure (eg list or set) for efficient searching.

```
def extract:
```

Retrieve and analyze emails:

- Create a function to retrieve emails from a text file, email server or database.
- Extract sender information (email address or domain) from each email.

Blacklist comparison:

- Compare the extracted email address or domain with the loaded blacklist.
- If the address or domain is blacklisted, mark the email as spam.

Evaluation of the results:

- Test the program using sample data that contains both spam and legitimate emails.
- Calculate precision, recall, and F1 scores to evaluate program performance.

2.2 Naive Bayes classifier

🕮 2.2.1

The Naive Bayes classifier

The Naive Bayes classifier is a probabilistic machine learning algorithm often used for text classification, such as spam detection. It is based on Bayes' theorem, which describes the probability of an event occurring based on prior knowledge of the conditions associated with that event.

Principle of operation

The algorithm assumes that individual attributes (in this case, words in the text) are independent. This means that the occurrence of one word does not affect the probability of occurrence of another word. Although this assumption is often not met in the real world, the Naive Bayes classifier achieves surprisingly good results in practice.

The Naive Bayes classifier works in two phases:

1. Learning phase: The algorithm analyzes the training data, which contains emails marked as spam or ham (legitimate emails). Based on this analysis, it calculates the probability of occurrence of individual words in spam and ham emails.
2. Classification phase: When classifying a new email, the algorithm calculates the probability that the email belongs to the spam and ham categories, based on the probabilities of occurrence of individual words in the training data. The email is then assigned to a category with a higher probability.

An example

Let's imagine that we have the following training data:

Email	Category
Get a million dollars for free!	Spam
Meeting tomorrow at 2pm	Ham
You won the lottery!	Spam
Important message from your bank	Ham

Based on this data, the algorithm calculates the probability of occurrence of individual words in spam and ham emails. For example, the word "free" occurs only in spam emails, while the word "meeting" occurs only in ham emails.

If we receive a new email with the text "Get a free gift!", the algorithm calculates the probability that the email belongs to the spam and ham category. Since the word "free" only occurs in spam emails, the probability that the email is spam will be higher and the email will be classified as spam.

Advantages

● Ease of implementation and speed

● Good results in practice, although the assumption of independence of attributes is not always fulfilled

Disadvantages

● Sensitivity to "zero" probabilities - if a word in the training data does not appear in any category, the algorithm assigns it a zero probability, which can distort the results

● Limited accuracy - Naive Bayes classifier does not achieve as much accuracy as more complex machine learning algorithms

Naive Bayes classifier is a simple and efficient algorithm for text classification. Although it has its limitations, it achieves surprisingly good results in practice and is often used for spam detection.

2.2.2

The Naive Bayes classifier - more sophisticated example

Let's expand the dataset to include more varied emails:

Email	Category
Get a million dollars for free!	Spam
Meeting tomorrow at 2pm	Ham
You won the lottery!	Spam
Important message from your bank	Ham
Claim your free gift now!	Spam
Project update for next week	Ham
Congratulations! You've won a prize!	Spam
Reminder: Doctor's appointment tomorrow	Ham
Get paid for your opinions!	Spam
Can we reschedule our meeting?	Ham

Analyzing the Training Data

- **Word Occurrence**: Count the occurrence of each word in both spam and ham emails.

Word	Spam Occurrences	Ham Occurrences
get	2	1
million	1	0
dollars	1	0
for	2	1
free	3	0
meeting	1	2
tomorrow	1	2
won	2	0
lottery	1	0
important	1	1
message	1	1
bank	1	1
project	0	1
update	0	1
congratulations	1	0
prize	1	0
paid	1	0
opinions	1	0
reschedule	0	1

Probability Calculation

Calculate the probability of a word given the category (spam or ham). This can be calculated as follows:

```
P(word | spam) = Count of word in spam / Total words in spam
P(word | ham) = Count of word in ham / Total words in ham
```

Total Words

- Total words in spam: 15
- Total words in ham: 14

Example probabilities

for the word "free":

```
P(free | spam) = 3 / 15 = 0.2P
P(free | ham) = 0 / 14 = 0.0
```

For the word "meeting":

```
P(meeting | spam) = 1 / 15 = 0.067
P(meeting | ham) = 2 / 14 = 0.143
```

Classifying a New Email

Now, let's classify a new email: **"Get a free gift!"**

1. Extract Words: The words are "get", "a", "free", "gift".

2. Calculate Probabilities:

- Using Bayes' Theorem, we can compute the probabilities:

```
P(spam | email) = P(get | spam) × P(free | spam) × P(gift | spam)
P(ham | email) = P(get | ham) × P(free | ham) × P(gift | ham)
```

3. Calculate Individual Probabilities:

For "get":

```
P(get | spam) = 2 / 15 = 0.133
P(get | ham) = 1 / 14 = 0.071
```

for the word "free":

```
P(free | spam) = 3 / 15 = 0.2P
P(free | ham) = 0 / 14 = 0.0
```

For "gift":

- Since "gift" is not in the training set, we can apply Laplace smoothing. Assuming a vocabulary size of V=20:

```
P(gift | spam) = (0 + 1) / (15 + 20) = 1 / 35 = 0.029
P(gift | ham) = (0 + 1) / (14 + 20) = 1/34 = 0.029≈0.029
```

Final Probability Calculation:

```
P(spam | email) = (0.133 × 0.2 × 0.029) = 0.000771
P(ham | email) = (0.071 × 0.0 × 0.029) = 0
```

Classification:

Since **P(spam | email)** is greater than **P(ham | email)**, the email "Get a free gift!" is classified as **SPAM**.

2.2.3

Laplace smoothing

Laplace smoothing, also known as add-one smoothing, is a technique used in probabilistic models, especially in natural language processing and machine learning, to handle the problem of zero probabilities in categorical data. It is particularly useful in applications such as language modeling, text classification, and spam detection.

Understanding the need for Laplace smoothing

In many probabilistic models, especially when estimating the probability of events based on training data, it is common to encounter situations where some events (or words, in the case of text data) do not appear at all in the training set. For instance, if you're calculating the probability of a specific word occurring in a document, and that word does not appear in any of your training samples, the estimated probability would be zero. This is problematic because:

1. **Zero probability problem**: If any single event has a probability of zero, it can lead to misleading results when calculating probabilities for combinations of events, particularly in tasks like classification or generating sequences.
2. **Sparsity**: In text data, especially in large vocabularies, many words may not appear in every training sample, leading to a sparse representation of data.

How Laplace smoothing works

Laplace smoothing addresses these issues by adjusting the probability estimates to ensure that no probability is ever zero. Here's how it works mathematically:

1. Basic Formula:

- For a given event (like a word) **w** in a training set, the probability of **w** is estimated as:

```
P(w) = (C(w)+1) / (N + V)
```

Where:

- **C(w)** is the count of occurrences of word www in the training set.
- **N** is the total number of words (tokens) in the training set.
- **V** is the size of the vocabulary (the total number of unique words in the training set).

2. Adjustment

By adding 1 to the count of each word, and adding the vocabulary size **V** to the total count **N**, Laplace smoothing effectively distributes some probability mass to words that were not seen in the training data.

3. Example

Suppose you have the following counts from your training data:

- Word "spam": 3 occurrences
- Word "ham": 2 occurrences
- Vocabulary size **V**: 4 (words: spam, ham, free, click)
- Total count **N**: 5 (3 spam + 2 ham)

The probability for "spam" would be:

```
P(spam) = (3 + 1) / (5 + 4) = 4 / 9
```

The probability for "ham" would be:

```
P(ham) = (2 + 1) / (5 + 4) = 3 / 9
```

For a word like "free", which has zero occurrences:

```
P(free)= (0 + 1) / (5 + 4) = 1 / 9
```

Benefits of Laplace Smoothing

1. **Avoids Zero Probabilities**: Ensures that all possible outcomes have a non-zero probability, which is crucial in many probabilistic models.
2. **Improves Generalization**: By smoothing the probabilities, the model can generalize better to unseen data, especially in sparse datasets.
3. **Simple to Implement**: Laplace smoothing is easy to understand and implement, making it a popular choice for many applications.

Limitations

1. **Assumption of Uniformity**: By adding a constant value (1), Laplace smoothing assumes that all unseen events should be treated equally, which may not always be true.
2. **Bias Towards Rare Events**: It can lead to overestimation of probabilities for rare events, particularly in cases where some events may be significantly less likely than others.
3. **More Advanced Methods Available**: While effective, there are other smoothing techniques (like Lidstone smoothing or Kneser-Ney smoothing) that may yield better results for specific applications.

2.2.4

Project: Naive Bayes classifier

Develop a Naive Bayes classifier that can categorize emails as either "Spam" or "Ham" (non-spam) based on their content. You will utilize a training dataset to compute probabilities and apply the classifier to new emails.

Dataset Preparation

- Collect a dataset of emails labeled as "Spam" or "Ham." You can use publicly available datasets like the Enron Email Dataset or https://www.kaggle.com/datasets/venky73/spam-mails-dataset or create your own.
- Ensure that the dataset contains a diverse set of emails, covering different topics and spam characteristics.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, precision_score,
recall_score

# ready
```

Data Preprocessing

- Clean the email texts by removing any irrelevant information, such as HTML tags, special characters, and stop words.
- Tokenize the email content to extract individual words or phrases.

```
# Function to preprocess and tokenize emails
def preprocess(emails):
  #...

# preprocessing data
spam_tokens = preprocess(spam_emails)
ham_tokens = preprocess(ham_emails)
```

Feature Extraction

- Create a frequency distribution of words in both spam and ham emails. This will help in calculating the probability of each word given the email category.
- Implement Laplace smoothing to handle words that may not appear in one of the categories.

```
# Frequency distributions
spam_freq = defaultdict(int)
ham_freq = defaultdict(int)

for token in spam_tokens:
    spam_freq[token] += 1

for token in ham_tokens:
    ham_freq[token] += 1
```

Probabilistic Model Development

- Calculate the prior probabilities for spam and ham categories based on their occurrence in the dataset.
- For each word, calculate the likelihood probabilities conditioned on each category (i.e., the probability of a word appearing in spam vs. ham).
- Use Laplace smoothing to ensure that words not present in the training set do not lead to a zero probability.

```
def laplace_smoothing(word, category_freq, total_words,
vocabulary_size):
    return (category_freq[word] + 1) / (total_words +
vocabulary_size)

# for each word ...
```

Classifying New Emails

- Create a function that classifies a new email based on calculated probabilities.

```
# new emails
```

Evaluation

- Use test data to evaluate the classifier's performance and compute metrics.

```
# evaluation
```

2.3 Bayes theorem

2.3.1

Bayes' theorem is a fundamental principle in probability theory and statistics that calculates the likelihood of an event based on prior knowledge of related conditions. It's especially useful in machine learning for developing probabilistic models, such as the Naive Bayes classifier, to make predictions based on data.

Bayes' theorem can be written as:

$$P(A \mid B) = \frac{P(B \mid A).P(A)}{P(B)}$$

Where:

- P(A|B) is the **posterior probability**: the probability of event A occurring given that B is true.
- P(B|A) is the **likelihood**: the probability of observing B given that A is true.
- P(A) is the **prior probability**: the probability of A occurring independently of B.
- P(B) is the **marginal probability**: the overall probability of B occurring.

2.3.2

Example of Bayes' Theorem in Use

Let's say a bank wants to determine if a transaction is fraudulent based on prior data. Let:

- A be the event "transaction is fraudulent."
- B be the event "transaction is from a foreign country."

Assume:

- The probability that a transaction is fraudulent, P(A), is 1%.
- The probability that a transaction is from a foreign country, P(B), is 10%.
- The probability that a transaction from a foreign country is fraudulent, P(B|A), is 5%.

Using Bayes' theorem, we can find P(A|B), the probability that a foreign transaction is fraudulent:

$$P(A \mid B) = \frac{P(B \mid A).\, P(A)}{P(B)} = \frac{0.05 \times 0.01}{0.1} = 0.005 \text{ or } 0.5\,\%$$

So, given the data, the probability that a foreign transaction is fraudulent is 0.5%.

2.3.3

Bayes Classifier

The Bayes classifier is a probabilistic model based on Bayes' theorem, often used for classification tasks. One of the most common forms is the **Naive Bayes classifier**, which assumes that features are independent of each other (hence "naive"). This assumption makes it computationally efficient and useful for text classification tasks like spam detection.

How the Bayes Classifier Works for Spam Detection

Training Phase:

- Gather data with labeled examples (e.g., spam and ham emails).
- Calculate the probability of each word appearing in spam and ham emails.
- Calculate prior probabilities for each class (spam and ham).

Prediction Phase:

- For a new email, break it down into individual words.
- Using Bayes' theorem, compute the probability of the email being spam or ham based on the frequency of each word.
- Classify the email as spam or ham based on the higher probability.

2.4 TF-IDF

📖 2.4.1

TF-IDF

TF-IDF (Term Frequency-Inverse Document Frequency) is a statistical method used to search and recognize important words in a document with respect to the entire corpus of documents. In the context of spam filtering, it allows us to determine which words are characteristic of spam administrations and which are less important or even typical of legitimate administrations.

How does it work?

Term Frequency (TF):

- It measures how often a given word occurs in a specific document.
- The higher the TF, the more important the word can be for that particular document.
- For example, the word "free" will appear with high frequency in spam messages offering something for free.

Inverse Document Frequency (IDF):

- It measures how unique a given word is in the entire corpus of documents (emails).
- The fewer documents a given word contains, the higher its IDF.
- Words like "and", "that", "is" occur in most documents and thus have a low IDF.

TF-IDF:

- It is the product of TF and IDF.
- A higher TF-IDF value means that the word is important for the given document and at the same time is relatively rare in the entire corpus.
- Words with a high TF-IDF value are often good candidates for characteristic words for a given document category (in our case, spam).

📝 2.4.2

Which of the following best describes the purpose of TF-IDF in spam filtering?

- To identify important words within a document compared to the entire corpus
- To detect grammatical errors in documents
- To highlight frequently used words in the entire corpus
- To measure the length of documents in the corpus

📝 2.4.3

Select statements that are true about Term Frequency (TF) and Inverse Document Frequency (IDF):

- TF calculates word frequency within a specific document.
- IDF gives a higher score to words found in fewer documents.
- IDF assigns a higher value to common words found in most documents.
- TF measures how often a word appears across all documents.

📝 2.4.4

The TF in TF-IDF stands for _____ frequency, which shows the importance of a word in a particular document.

A higher TF indicates that a word is more _____ within a specific document.

Calculating _____ helps to find words that are repeated often in a single document.

- frequency
- frequent
- term

🕮 2.4.5

TF-IDF and Bayes Classifier for spam detection

TF-IDF (Term Frequency-Inverse Document Frequency) is a method to evaluate the importance of a word in a document relative to a collection of documents (corpus). In spam detection, TF-IDF is used to assign weights to words based on how frequently they appear in spam emails versus legitimate (ham) emails. This is how TF-IDF strengthens the Bayes classifier's effectiveness.

Steps to integrate TF-IDF with Bayes Classifier

1. Calculate TF-IDF values:

- For each word in the training dataset, calculate the term frequency (TF) of the word in each email.
- Calculate the inverse document frequency (IDF) across all emails in the dataset.
- Multiply TF and IDF to get the TF-IDF score for each word in both spam and ham categories.
- **Example**: Assume that "win" appears often in spam but rarely in ham emails. TF-IDF will assign it a high value, making it a strong spam indicator.

2. Incorporate TF-IDF into Bayes Classifier:

- Use the TF-IDF score as a weight in the Bayes classifier's calculation. Words with higher TF-IDF values will contribute more to the likelihood of the email being spam.
- For each new email, calculate the probability of it being spam or ham by incorporating the weighted contributions of each word's TF-IDF score.

3. Final classification:

- After calculating the probabilities for both spam and ham categories, classify the email based on the category with the higher weighted probability.

2.4.6

Detailed example of spam detection

Imagine a small dataset with four training emails:

Email	Category
Get a million dollars for free!	Spam
Meeting tomorrow at 2pm	Ham
You won the lottery!	Spam
Important message from your bank	Ham

Step 1: Calculate word frequency for spam and ham

Word	Spam Frequency	Ham Frequency
get	1	0
a	2	2
million	1	0
dollars	1	0
free	1	0
meeting	0	1
tomorrow	0	1

Step 2: Calculate TF-IDF scores

Using the TF-IDF formula, we get:

Word	TF-IDF Score (Spam)	TF-IDF Score (Ham)
get	0.6	0.0
a	0.2	0.2
million	0.5	0.0
dollars	0.5	0.0
free	0.6	0.0
meeting	0.0	0.6
tomorrow	0.0	0.6

Step 3: Apply TF-IDF scores in Bayes calculation

We use the TF-IDF scores to improve the accuracy of the Naive Bayes classifier in identifying spam messages. This involves applying Bayes' theorem and incorporating the TF-IDF scores for words that appear in a new email message to determine the likelihood that the message is spam. Here's a step-by-step breakdown:

1. Identify words in the incoming email:

Suppose we receive a new email message with the text: `"Get a free gift!"`.

2. Extract relevant words from the message:

The main words here are:

- "get"
- "a"
- "free"
- "gift"

3. Retrieve TF-IDF scores:

Using the TF-IDF scores from Table 3 for each word, we get:

- **"get"** - TF-IDF (Spam): 0.6, TF-IDF (Ham): 0.0
- **"a"** - TF-IDF (Spam): 0.2, TF-IDF (Ham): 0.2
- **"free"** - TF-IDF (Spam): 0.6, TF-IDF (Ham): 0.0
- **"gift"** - Since "gift" wasn't in the training data, we can assume an equal likelihood for both Spam and Ham (or use smoothing techniques to avoid zero probabilities).

4. Apply Bayes' theorem for classification:

We'll use Bayes' theorem to determine the probability that the email is spam P(Spam | Email) and the probability that the email is ham P(Ham | Email).

$$P(Spam \mid Email) = \frac{P(Email \mid Spam).\, P(Spam)}{P(Email)}$$

Where:

- P(Email|Spam): The probability of this email being observed given it's spam.
- P(Spam): The prior probability of spam (often taken as the percentage of spam emails in the dataset).
- P(Email): The overall probability of observing this email.

Step 5: Calculate P(Email|Spam) and P(Email|Ham)

Calculate P(Email|Spam)

Using the formula:

```
P(Email|Spam) = TF-IDF("get", Spam) × TF-IDF("a", Spam) × TF-IDF("free", Spam) × TF-IDF("gift", Spam)
```

Substitute the values:

```
P(Email|Spam) = 0.6 × 0.2 × 0.6 × 0.5 = 0.036
```

Calculate P(Email|Ham)

Using the same method:

```
P(Email|Ham) = TF-IDF("get", Ham) × TF-IDF("a", Ham) × TF-IDF("free", Ham) × TF-IDF("gift", Ham)
```

Substitute the values:

```
P(Email|Ham) = 0.0 × 0.2 × 0.0 × 0.5 = 0.0
```

Step 6: Apply Bayes' theorem

Since

```
P(Email|Ham) = 0
```

the probability that this email is ham is essentially zero.

Therefore:

```
P(Spam|Email) > P(Ham|Email)
```

Result

Since **P(Email|Spam) = 0.036** and **P(Email|Ham) = 0.0**, the classifier would label this email as **Spam**.

This result aligns with the intuition: the presence of terms like "get" and "free," which are common in spam messages, indicates that this email is likely spam.

2.4.7

Project: Spam Classifier with TF-IDF and Naive Bayes

(by https://hussnain-akbar.medium.com/understanding-and-implementing-na%C3%AFve-bayes-algorithm-for-email-spam-detection-85a14b330fc6)

Create a spam classifier using a Naive Bayes algorithm in combination with TF-IDF (Term Frequency-Inverse Document Frequency) for feature extraction.

The Naïve Bayes classifier is a supervised machine learning model that predicts the probability of an event by analyzing related features. Here, "Naïve" means that the model assumes that all features are independent, meaning that each feature contributes to the prediction independently. In simpler terms, the model considers each feature separately, without assuming any relationships between them.

For now, we will start with a simple version of the model to make it easier to understand. To do this, we will create a small, sample dataset.

```
#Essential libraries required for this model
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score,
classification_report
```

This code will create a data frame with random emails and their corresponding labels (spam or not spam). Each email will consist of a random selection of words from the **word_list**. However, the above code will have the following output.

```
# Create a random dataset
np.random.seed(42)  # For reproducibility

# Generate random words for features (words in emails)
word_list = ['discount', 'offer', 'sale', 'free', 'click',
'buy', 'win', 'money', 'gift', 'limited']

# Generate random emails
num_emails = 1000
emails = []
labels = []
for _ in range(num_emails):
    email = ' '.join(np.random.choice(word_list,
size=np.random.randint(5, 15)))
    emails.append(email)
    # Assign labels (spam or not spam)
    labels.append(np.random.choice(['spam', 'not spam'],
p=[0.3, 0.7]))

# Create a DataFrame
data = pd.DataFrame({'email': emails, 'label': labels})

# Display the first few rows of the dataset
print(data.head())
```

Program output:

```
                                               email     label
0  free money click win limited sale win money cl...  not spam
1  click offer money buy offer click discount lim...      spam
2  sale win free gift sale click sale win click g...  not spam
3              limited gift limited click offer free      spam
4  money sale discount free offer money free offe...  not spam
```

Let's walk through the steps to build and train a Naïve Bayes classifier using the dataset we created. Here is a breakdown of the four main steps:

1. Data preprocessing

In this step, we will convert the text data to numeric characters. We will use the TF-IDF (Term Frequency-Inverse Document Frequency) technique, which transforms the

text into a format understood by the Naïve Bayes classifier. The TF-IDF approach helps highlight important words in a dataset while reducing the impact of common words that may not provide significant meaning.

Steps:

- Tokenization: Splitting text into individual words or tokens.
- Lowercase: Convert all text to lowercase for consistency.
- Eliminating Stop Words: Eliminate common words (such as "the", "is", "and") that do not add much to the meaning.
- TF-IDF Calculation: Calculate the TF-IDF score for each word in each document.

In following code we apply only conversion of text data into numerical features using techniques like TF-IDF.

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score,
classification_report

# Preprocessing: Convert text data to numerical features
tfidf_vectorizer = TfidfVectorizer(max_features=1000)  # Limit
features to 1000 for simplicity
X = tfidf_vectorizer.fit_transform(data['email'])
y = data['label']
```

2. Splitting the data

Next, we need to split the dataset into two parts: one for training the model and another for testing its performance. A typical split might allocate 70-80% of the data for training and the remaining 20-30% for testing.

We will use a library **sklearn** to split the dataset into training and testing sets, ensuring that both sets contain a representative distribution of classes (e.g., spam and not spam).

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

3. Training the Naïve Bayes Model

Now we can train the Naïve Bayes classifier using the training data. The model will learn from the features extracted in the preprocessing step.

Steps:

- Create an instance of the Naïve Bayes classifier.
- Fit the model on the training data, allowing it to learn the relationship between the features and the labels (spam or not spam).

```
# Initialize and train the Naive Bayes classifier
naive_bayes = MultinomialNB()
naive_bayes.fit(X_train, y_train)
```

4. Evaluating the Model

After training the model, we'll evaluate its performance on the testing data to see how well it predicts new, unseen data.

Steps:

- Use the trained model to make predictions on the testing set.
- Compare the predicted labels to the actual labels to calculate performance metrics such as:
- Accuracy: The proportion of correctly classified instances.
- Precision: The proportion of true positive predictions to the total positive predictions.
- Recall (Sensitivity): The proportion of true positive predictions to the total actual positives.
- F1 Score: The harmonic mean of precision and recall, providing a balance between the two.

```
y_pred = naive_bayes.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred,
zero_division=0)

print(f'Accuracy: {accuracy}')
print('Classification Report:\n', report)
```

Program output:

```
Accuracy: 0.66
Classification Report:
              precision    recall  f1-score   support

    not spam       0.66      1.00      0.80       132
```

```
        spam       0.00      0.00      0.00        68

    accuracy                           0.66       200
   macro avg       0.33      0.50      0.40       200
weighted avg       0.44      0.66      0.52       200
```

The accuracy of our Naive Bayes classifier on the test data is 66%. This means that the model correctly identified about two-thirds of the emails in our test set. However, when we look closer at the classification report, we notice that the precision, recall, and F1 score for the “spam” class are quite low.

Low precision means that when the model predicts an email is spam, it often turns out to be wrong. Low recall indicates that the model is missing many actual spam emails, failing to identify them correctly. Essentially, this suggests that our model struggles to accurately recognize spam emails, which is a significant concern for applications that rely on effective spam detection.

The final step is to use our trained Naive Bayes model to predict whether new emails are spam or not. To do this, we run the following code, which takes the new email data and applies the model we’ve trained. After running the prediction, we can analyze the output to see how well the model identifies spam in this new data.

```
# Example of a new email to be predicted
new_email = "Limited time offer! Click here to win a free
gift."

# Preprocess the new email using the TF-IDF vectorizer from
the training
new_email_features = tfidf_vectorizer.transform([new_email])

# Make prediction using the trained Naive Bayes classifier
predicted_label = naive_bayes.predict(new_email_features)

# Print the predicted label
print(f"Predicted Label: {predicted_label[0]}")
```

Program output:

```
Predicted Label: not spam
```

Machine Learning in Spam Detection

Chapter 3

3.1 AI spam detection

📖 3.1.1

Spam detection is a dynamic challenge due to the constantly evolving nature of spam. Machine learning (ML) algorithms are well-suited for this task because they can adapt to new spam patterns without human intervention. ML models analyze vast data, including message patterns, user behaviors, and known spam indicators, to detect likely spam.

An ML model calculates a risk score based on these factors, classifying messages accordingly. This approach is highly effective in identifying subtle variations in spam messages, making detection more accurate and reducing the number of legitimate emails marked as spam.

📝 3.1.2

In spam detection, _____ _____ algorithms adapt to evolving spam by analyzing message _____ and user _____ to identify probable spam content.

- machine
- patterns
- learning
- behavior

📖 3.1.3

In today's age of digital communication, spam is a ubiquitous problem. Traditional spam filters, based on rules and keyword detection, are no longer sufficient to combat the constantly evolving techniques of spammers. AI is thus becoming increasingly important in the fight against spam as it improves traditional methods through advanced data analysis and pattern recognition.

Content analysis

AI algorithms excel at content analysis by examining the text in emails to determine their legitimacy. They consider various elements including word choice, sentence structure and semantic meaning. For example, spam emails often contain specific phrases such as "Click here to claim your prize!" or "Act now to secure your offer!" Artificial intelligence can be trained to recognize these linguistic signals and associate them with the characteristics of spam.

In addition, AI can analyze the overall tone and context of messages, distinguishing between promotional content and genuine communication. By leveraging natural language processing (NLP), AI systems can further understand the intent of words and identify manipulative or deceptive language commonly used in spam. In-depth

analysis thus enables a more accurate classification of e-mails, reduces the number of false alarms and improves the user experience.

Additionally, while spammers develop their language to bypass filters, AI constantly updates its understanding and ensures that new spam tactics are effectively identified.

3.1.4

Which of the following elements does AI analyze in email content to determine its legitimacy?

- Text, structure, and semantic meaning
- Subject line
- Recipient’s email address
- Time of day the email was sent

3.1.5

Pattern detection

AI's ability to recognize patterns in spam messages is another critical aspect of its effectiveness. Over time, AI systems learn to identify recurring themes, such as common keywords or phrases that appear frequently in spam. For example, terms like "urgent," "limited time," or "guaranteed" often signal promotional emails intended to elicit an immediate response.

In addition, artificial intelligence can detect suspicious URLs and email templates that are typical of spam campaigns. For example, a URL pointing to a domain with a long string of random characters may indicate phishing attempts. By analyzing large data sets, AI can uncover hidden correlations between different email features, facilitating accurate categorization of messages.

Additionally, machine learning algorithms improve their accuracy as they process more data, allowing them to quickly adapt to emerging spam tactics. Proactive approach to pattern detection not only identifies existing spam, but also predicts future threats based on historical data.

3.1.6

What type of elements does AI recognize in spam messages to improve detection accuracy?

- Images included in the email
- Recurring themes, keywords, and suspicious URLs
- The sender's location
- The length of the email

📖 3.1.7

Identification of anomalies

AI's ability to detect anomalies is critical to identifying unusual behavior that may indicate spam activity. For example, if a user normally receives ten emails a day, but suddenly sees an increase to a hundred, the AI may flag this as suspicious. This increase in message volume could indicate that the user is being targeted by a spam campaign.

Similarly, AI monitors sending patterns to detect anomalies, such as sending multiple emails to new senders in a short time frame. For example, if an email address that has never sent messages before suddenly sends a large number of emails, this may indicate hacked accounts or spam activity.

AI systems use statistical methods to determine underlying behavior, allowing them to identify deviations with high accuracy. This capability is especially valuable in enterprise environments where large volumes of email are exchanged daily.

📝 3.1.8

What does AI monitor to detect anomalies that might indicate spam activity?

- Sending patterns and volume of emails
- The color scheme of the email
- User interaction with emails
- The sender’s font choice

📖 3.1.9

Adaptability

One of the most significant advantages of AI in spam detection is its adaptability. AI algorithms are designed to constantly learn from new data, allowing them to adapt to changing spam tactics.

For example, as spammers develop new techniques to avoid detection—such as using image-based messages or obfuscating links - AI can update its models to recognize these evolving patterns. This ongoing learning process is facilitated by feedback loops where the AI receives information about its classifications, allowing it to refine its accuracy over time.

As a result, AI systems can quickly adapt to new threats without the need for manual updates to filtering criteria. For example, if a new phishing technique emerges that combines social engineering with legitimate-looking websites, the AI can quickly incorporate that information into its detection framework.

📝 3.1.10

How does AI maintain its effectiveness against evolving spam tactics?

- By relying solely on user reports
- By continuously learning from new data
- By using static rules from the past
- By limiting analysis to only the most recent emails

📖 3.1.11

The importance of AI in spam detection lies in:

● Increased accuracy: AI systems can identify spam messages with greater accuracy and reduce the number of false positives and false negatives.

● Efficiency improvements: AI automates the spam detection process and enables more efficient filtering of large volumes of messages.

● Threat protection: AI helps protect users from spam-related threats, such as phishing attacks, the spread of malware, and the loss of sensitive information.

● Keeping the inbox clean: AI helps keep the inbox clean and makes it easier for users to sort through important messages.

📝 3.1.12

The importance of AI in spam detection lies in increased accuracy, efficiency improvements, threat protection, and keeping the inbox _____ and making it easier for users to sort through _____ messages.

Additionally, AI significantly enhances the _____ of spam detection by learning from past data and adjusting to new tactics. This adaptability ensures that users are better protected against _____ that can compromise their sensitive information.

- efficiency
- clean
- important
- threats

3.2 Implementation

📖 3.2.1

Process of implementing AI-based spam detection

Implementing an AI-based spam detection system involves several key steps. First, it's essential to choose the right AI tools and platforms, as there are various options designed for different levels of complexity and technical requirements. After selecting a tool, the next step is to train the AI model on a large dataset that includes both spam and legitimate messages. This allows the model to learn what spam looks like and what differentiates it from legitimate messages. The training phase is crucial, as the quality and diversity of the data directly affect the accuracy of the AI model.

Once the model is trained, the next phase is deployment, where the model is integrated into the desired application, such as an email server or social media platform. After deployment, continuous testing is necessary to ensure the model maintains its accuracy, especially as spam tactics evolve. Regular updates and fine-tuning of the AI model are often required to keep up with these changes. This iterative process ensures that AI-based spam detection remains effective over time, adapting to new challenges and maintaining online security.

📝 3.2.2

Which steps are part of implementing an AI-based spam detection system?

- Choosing the right AI tools
- Training the AI model
- Deploying and testing the AI model
- Manually filtering spam messages

📖 3.2.3

While AI-based spam detection offers many benefits, it also presents certain challenges. One major consideration is data privacy. AI-based systems require large amounts of data to train effectively, but this data often includes sensitive or private information. Ensuring that data privacy is protected is a critical step, often involving data anonymization or using synthetic data to avoid compromising user information.

Another challenge is managing false positives and false negatives. Even with advanced AI, there can still be mistakes, where legitimate emails are flagged as spam or spam messages slip through. Reducing these errors requires careful tuning of the AI model and ongoing improvements based on feedback. Finally, AI-based spam detection systems need continuous adaptation. Spammers constantly change their tactics, so models must be updated to handle new types of spam and emerging threats. This ongoing process keeps spam detection accurate and reliable.

3.2.4

In AI-based spam detection, data _____ is crucial to protect user information, while constant _____ helps the system adapt to new spam techniques.

- adaptation
- privacy

3.2.5

Data privacy

Data privacy is a critical concern in AI-based spam detection, as these systems analyze large volumes of message data to distinguish spam from legitimate content. Since this data may contain sensitive information, organizations must implement robust measures to protect user privacy. Encryption, access controls, and data anonymization are key strategies that help safeguard this information, ensuring compliance with data protection regulations. Furthermore, transparency in data collection and usage practices is essential. By informing users about how their data is used in spam detection and obtaining consent, organizations can build trust, helping users feel secure in the system's operations.

In addition to these measures, it is essential for organizations to stay informed about updates to data protection laws and implement necessary changes proactively. By prioritizing data privacy, AI-based spam detection can function effectively while respecting user privacy rights, building a safer and more responsible digital environment.

3.2.6

Which is a crucial aspect of AI-based spam detection related to data?

- Data anonymization
- Increasing spam accuracy
- Reducing spam messages
- Preventing phishing attacks

3.2.7

Minimizing false positives and negatives

Balancing false positives and false negatives is a significant challenge in AI-based spam detection. False positives, where legitimate messages are incorrectly flagged as spam, can lead to user frustration and a loss of trust in the spam filter. On the other hand, false negatives allow spam messages to slip through, potentially exposing users to security risks. To address this, AI models must be continuously fine-tuned and trained with diverse datasets to improve accuracy. This process

includes regularly analyzing user feedback and adjusting model parameters to minimize errors, ensuring that legitimate messages reach users while spam is effectively filtered out.

Furthermore, ongoing evaluation of the model's performance is essential, as it helps detect shifts in message patterns over time. A combination of user input and technical adjustments creates a balanced spam detection system, minimizing disruptions while maximizing security for users.

📝 3.2.8

What can help reduce false positives and negatives in AI-based spam detection?

- Regular model updates
- User feedback analysis
- Stopping all spam checks
- Using rule-based filters

🕮 3.2.9

Adapting to evolving spam techniques

Spam techniques evolve constantly, with spammers developing new tactics to bypass detection. For AI-based spam filters to stay effective, they must adapt continuously to these changing methods. This involves regular retraining of the AI model on new data, updating the system to recognize emerging spam patterns, and staying aware of recent trends in spam behavior. By doing so, the model can detect even advanced spam methods, such as zero-day attacks, which may initially evade simpler filters. Proactive adaptation keeps spam detection systems effective, blocking new threats as soon as they appear.

Organizations can enhance this adaptability by collaborating with industry peers and security experts. Sharing insights on recent spam trends enables faster responses to new threats, creating a more resilient spam detection framework. This continuous adaptation keeps online environments safe and spam-free for users.

📝 3.2.10

Why is continuous adaptation necessary in AI-based spam detection?

- Spammers regularly evolve their techniques
- It's cheaper than traditional spam detection
- Users prefer it over other methods
- It ignores user privacy

🕮 3.2.11

Performance and scalability

AI-based spam detection is computationally intensive, requiring substantial resources to process large volumes of messages accurately. Ensuring the system's performance and scalability is essential, particularly for organizations handling high message traffic. Allocating sufficient processing power and memory allows the spam filter to operate smoothly without delays. Optimizing algorithms and using efficient data processing techniques can further enhance the model's performance, allowing it to scale with increasing demand without compromising speed or accuracy.

Scalability also involves planning for future growth, enabling the system to handle larger data volumes as the organization expands. By investing in both performance optimization and scalability, organizations can maintain a reliable and efficient spam detection system that supports the demands of a growing user base.

3.2.12

To handle the large volumes of _____ in spam _____, organizations need to focus on both performance and _____.

- messages
- scalability
- detection

3.3 AI projects

3.3.1

Project: Multinomial Naive Bayes

(by https://github.com/Apaulgithub/oibsip_taskno4/blob/main/Email_Spam_Detection_with_Machine_Learning.ipynb)

In the rapidly developing digital world, the fight against spam emails will become an increasingly important challenge. Spam emailsare expected to continue to flood inboxes.

The main points of the project:

- **Data Preprocessing**: We start by preparing a large email dataset, which includes cleaning the data, handling missing values, and transforming the text data into a machine learning-ready format.
- **Feature Extraction**: We will use a variety of feature extraction techniques to capture the defining characteristics of spam emails, a critical step in making the data interpretable for models.

- **Machine learning models**: A machine learning algorithms will be used to train and evaluate the spam detection model.
- **Evaluation metrics**: We carefully select evaluation metrics such as accuracy, precision, recall, and F1-score to measure model effectiveness and gain valuable performance information.
- **Tuning and Optimization**: Fine-tuning the hyperparameters and optimizing the model will increase the accuracy of the prediction, allowing the model to better adapt to future data.
- **Validation**: Thorough cross-validation and testing on a separate data set will ensure that the model generalizes well to unseen data.

1. Known your data

- import libraries
- upload dataset

```
# Import Libraries
# Importing Numpy & Pandas for data processing & data
wrangling
import numpy as np
import pandas as pd

# Importing  tools for visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Import evaluation metric libraries
from sklearn.metrics import confusion_matrix, accuracy_score,
precision_score, recall_score, f1_score, roc_auc_score,
roc_curve, classification_report

# Word Cloud library
from wordcloud import WordCloud, STOPWORDS

# Library used for data preprocessing
from sklearn.feature_extraction.text import CountVectorizer

# Import model selection libraries
from sklearn.model_selection import train_test_split

# Library used for ML Model implementation
from sklearn.naive_bayes import MultinomialNB

# Importing the Pipeline class from scikit-learn
from sklearn.pipeline import Pipeline
```

```
# Library used for ignore warnings
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline
```

Dataset Loading

- original source: https://raw.githubusercontent.com/Apaulgithub/oibsip_taskno4/main/spam.csv
- system copy: https://priscilla.fitped.eu/data/cybersecurity/spam/spam_908.csv

```
# Load Dataset from  repository
df =
pd.read_csv("https://priscilla.fitped.eu/data/cybersecurity/spam/spam_908.csv", encoding='ISO-8859-1')
```

Dataset First View

- Show 5 lines of data

```
# Dataset First Look
# View top 5 rows of the dataset
print(df.head())
```

Program output:

```
     v1                                                  v2
Unnamed: 2  \
0   ham  Go until jurong point, crazy.. Available only ...
NaN
1   ham                      Ok lar... Joking wif u oni...
NaN
2  spam  Free entry in 2 a wkly comp to win FA Cup fina...
NaN
3   ham  U dun say so early hor... U c already then say...
NaN
4   ham  Nah I don't think he goes to usf, he lives aro...
NaN

  Unnamed: 3 Unnamed: 4
0        NaN        NaN
1        NaN        NaN
2        NaN        NaN
3        NaN        NaN
4        NaN        NaN
```

```
```

Dataset Rows & Columns count

```
# Dataset Rows & Columns count
# Checking number of rows and columns of the dataset using
shape
print("Number of rows are: ",df.shape[0])
print("Number of columns are: ",df.shape[1])
```

Program output:

```
Number of rows are:  5572
Number of columns are:  5
```

Dataset Information

```
# Dataset Info
# Checking information about the dataset using info
df.info()
```

Program output:

```

RangeIndex: 5572 entries, 0 to 5571
Data columns (total 5 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   v1          5572 non-null   object
 1   v2          5572 non-null   object
 2   Unnamed: 2  50 non-null     object
 3   Unnamed: 3  12 non-null     object
 4   Unnamed: 4  6 non-null      object
dtypes: object(5)
memory usage: 217.8+ KB
```

Duplicate values

```
# Dataset Duplicate Value Count
dup = df.duplicated().sum()
print(f'number of duplicated rows are {dup}')
```

Program output:

```
number of duplicated rows are 403
```

Missing Values/Null Values

```
# Missing Values/Null Values Count
print(df.isnull().sum())
```

Program output:

```
v1                0
v2                0
Unnamed: 2     5522
Unnamed: 3     5560
Unnamed: 4     5566
dtype: int64
```

What did i know about the dataset?

- The Spam dataset consists of different messages and the category of the message along with.
- There are 5572 rows and 5 columns provided in the data.
- 403 duplicate rows are present in the dataset.
- No Null values exist in v1 & v2 column, but lots of null values present in unnamed 2,3,4 columns (will drop those 3 columns later).

2. Understanding The Variables

```
# Dataset Columns
print(df.columns)
```

Program output:

```
Index(['v1', 'v2', 'Unnamed: 2', 'Unnamed: 3', 'Unnamed: 4'],
dtype='object')
# Dataset Describe (all columns included)
print(df.describe(include= 'all').round(2))
```

Program output:

```
          v1                      v2  \
count   5572                    5572
unique     2                    5169
top      ham  Sorry, I'll call later
freq    4825                      30

                                                Unnamed: 2  \
count                                                   50
unique                                                  43
top      bt not his girlfrnd... G o o d n i g h t . . .@"
freq                                                     3
```

```
                    Unnamed: 3 Unnamed: 4
count                       12          6
unique                      10          5
top       MK17 92H. 450Ppw 16"    GNT:-)"
freq                         2          2
```

Check unique values for each variable.

```
# Check Unique Values for each variable using a for loop.
for i in df.columns.tolist():
  print("No. of unique values in",i,"is",df[i].nunique())
```

Program output:

```
No. of unique values in v1 is 2
No. of unique values in v2 is 5169
No. of unique values in Unnamed: 2 is 43
No. of unique values in Unnamed: 3 is 10
No. of unique values in Unnamed: 4 is 5
```

3. Data Wrangling

```
# Change the v1 & v2 columns as Category and Message
df.rename(columns={"v1": "Category", "v2": "Message"},
inplace=True)

# Removing the all unnamed columns (its include much number of
missing values)
df.drop(columns={'Unnamed: 2','Unnamed: 3','Unnamed: 4'},
inplace=True)

# Create a binary 'Spam' column: 1 for 'spam' and 0 for 'ham',
based on the 'Category' column.
df['Spam'] = df['Category'].apply(lambda x: 1 if x == 'spam'
else 0)

# Updated new dataset
print(df.head())
```

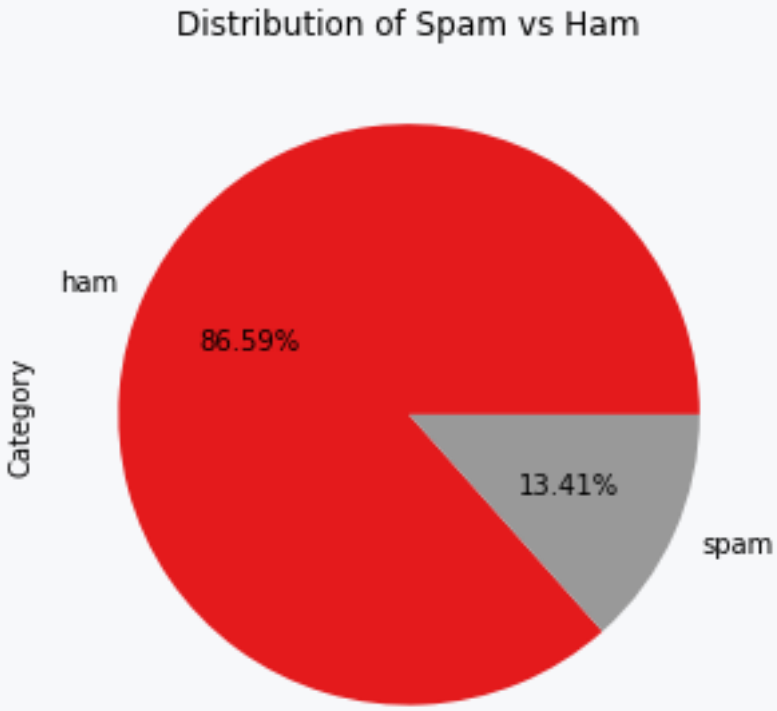
4. Data explanation

Vizualization, Storytelling & Experimenting with charts : Understand the relationships between variables

Chart 1: Distribution of Spam vs Ham

```
# Chart - 1 Pie Chart Visualization Code For Distribution of
Spam vs Ham Messages
spread = df['Category'].value_counts()
plt.rcParams['figure.figsize'] = (5,5)

# Set Labels
spread.plot(kind = 'pie', autopct='%1.2f%%', cmap='Set1')
plt.title(f'Distribution of Spam vs Ham')

# Display the Chart
plt.show()
```

Program output:



What is/are the insight(s) found from the chart?

We got to know that the dataset contain 13.41% of spam messages and 86.59% of ham messages.

Chart 2: Most Used Words in Spam Messages

```
# Filter Spam Messages
df_spam = df[df['Category'] == 'spam'].copy()
```

```
# String to Store All Words
comment_words = '
'.join(df_spam['Message'].astype(str).str.lower())

# Stopwords
stopwords = set(STOPWORDS)

# Create WordCloud Object with Parameters
wordcloud = WordCloud(width=1000, height=500,
                      background_color='white',
                      stopwords=stopwords,
                      min_font_size=10,
                      max_words=1000,

colormap='gist_heat_r').generate(comment_words)

# Plot WordCloud
plt.figure(figsize=(10, 6), facecolor=None)
plt.title('Most Used Words In Spam Messages', fontsize=15,
pad=20)
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")
plt.tight_layout(pad=0)
plt.show()
```

Program output:

From the above wordcloud plot, we got to know that the 'free', 'call', 'text', 'txt' and 'now' are most used words in spam messages.

5. Feature Engineering & Data Pre-processing

Data Splitting

```
# Splitting the data to train and test
X_train,X_test,y_train,y_test=train_test_split(df.Message,df.Spam,test_size=0.25)
```

6. ML Model Implementation

```
def evaluate_model(model, X_train, X_test, y_train, y_test):
    '''The function will take model, x train, x test, y train, y test
    and then it will fit the model, then make predictions on the trained model,
    it will then print roc-auc score of train and test, then plot the roc, auc curve,
    print confusion matrix for train and test, then print classification report for train and test,
    then plot the feature importances if the model has feature importances,
    and finally it will return the following scores as a list:
    recall_train, recall_test, acc_train, acc_test, roc_auc_train, roc_auc_test, F1_train, F1_test
    '''

    # fit the model on the training data
    model.fit(X_train, y_train)

    # make predictions on the test data
    y_pred_train = model.predict(X_train)
    y_pred_test = model.predict(X_test)
    pred_prob_train = model.predict_proba(X_train)[:,1]
    pred_prob_test = model.predict_proba(X_test)[:,1]

    # calculate ROC AUC score
    roc_auc_train = roc_auc_score(y_train, y_pred_train)
    roc_auc_test = roc_auc_score(y_test, y_pred_test)
    print("\nTrain ROC AUC:", roc_auc_train)
    print("Test ROC AUC:", roc_auc_test)

    # plot the ROC curve
```

```
    fpr_train, tpr_train, thresholds_train =
roc_curve(y_train, pred_prob_train)
    fpr_test, tpr_test, thresholds_test = roc_curve(y_test,
pred_prob_test)
    plt.plot([0,1],[0,1],'k--')
    plt.plot(fpr_train, tpr_train, label="Train ROC AUC:
{:.2f}".format(roc_auc_train))
    plt.plot(fpr_test, tpr_test, label="Test ROC AUC:
{:.2f}".format(roc_auc_test))
    plt.legend()
    plt.title("ROC Curve")
    plt.xlabel("False Positive Rate")
    plt.ylabel("True Positive Rate")
    plt.show()

    # calculate confusion matrix
    cm_train = confusion_matrix(y_train, y_pred_train)
    cm_test = confusion_matrix(y_test, y_pred_test)

    fig, ax = plt.subplots(1, 2, figsize=(11,4))

    print("\nConfusion Matrix:")
    sns.heatmap(cm_train, annot=True, xticklabels=['Negative',
'Positive'], yticklabels=['Negative', 'Positive'],
cmap="Oranges", fmt='.4g', ax=ax[0])
    ax[0].set_xlabel("Predicted Label")
    ax[0].set_ylabel("True Label")
    ax[0].set_title("Train Confusion Matrix")

    sns.heatmap(cm_test, annot=True, xticklabels=['Negative',
'Positive'], yticklabels=['Negative', 'Positive'],
cmap="Oranges", fmt='.4g', ax=ax[1])
    ax[1].set_xlabel("Predicted Label")
    ax[1].set_ylabel("True Label")
    ax[1].set_title("Test Confusion Matrix")

    plt.tight_layout()
    plt.show()


    # calculate classification report
    cr_train = classification_report(y_train, y_pred_train,
output_dict=True)
```

```
    cr_test = classification_report(y_test, y_pred_test,
output_dict=True)
    print("\nTrain Classification Report:")
    crt = pd.DataFrame(cr_train).T
    print(crt.to_markdown())
    # sns.heatmap(pd.DataFrame(cr_train).T.iloc[:, :-1],
annot=True, cmap="Blues")
    print("\nTest Classification Report:")
    crt2 = pd.DataFrame(cr_test).T
    print(crt2.to_markdown())
    # sns.heatmap(pd.DataFrame(cr_test).T.iloc[:, :-1],
annot=True, cmap="Blues")


    precision_train = cr_train['weighted avg']['precision']
    precision_test = cr_test['weighted avg']['precision']

    recall_train = cr_train['weighted avg']['recall']
    recall_test = cr_test['weighted avg']['recall']

    acc_train = accuracy_score(y_true = y_train, y_pred =
y_pred_train)
    acc_test = accuracy_score(y_true = y_test, y_pred =
y_pred_test)

    F1_train = cr_train['weighted avg']['f1-score']
    F1_test = cr_test['weighted avg']['f1-score']

    model_score = [precision_train, precision_test,
recall_train, recall_test, acc_train, acc_test, roc_auc_train,
roc_auc_test, F1_train, F1_test ]
    return model_score
```

ML Model: Multinomial Naive Bayes

```
# ML Model - 1 Implementation
# Create a machine learning pipeline using scikit-learn,
combining text vectorization (CountVectorizer)
# and a Multinomial Naive Bayes classifier for email spam
detection.
clf = Pipeline([
    ('vectorizer', CountVectorizer()),  # Step 1: Text data
transformation
    ('nb', MultinomialNB())  # Step 2: Classification using
Naive Bayes
```

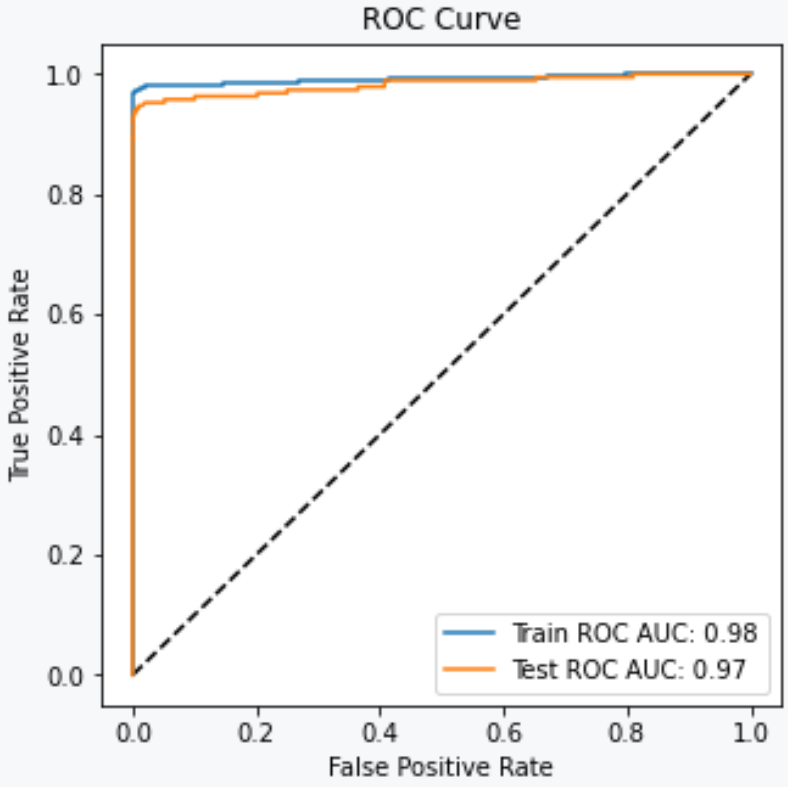
```
])

# Model is trained (fit) and predicted in the evaluate model
```

Explain the ML Model used and it's performance using Evaluation metric Score Chart.

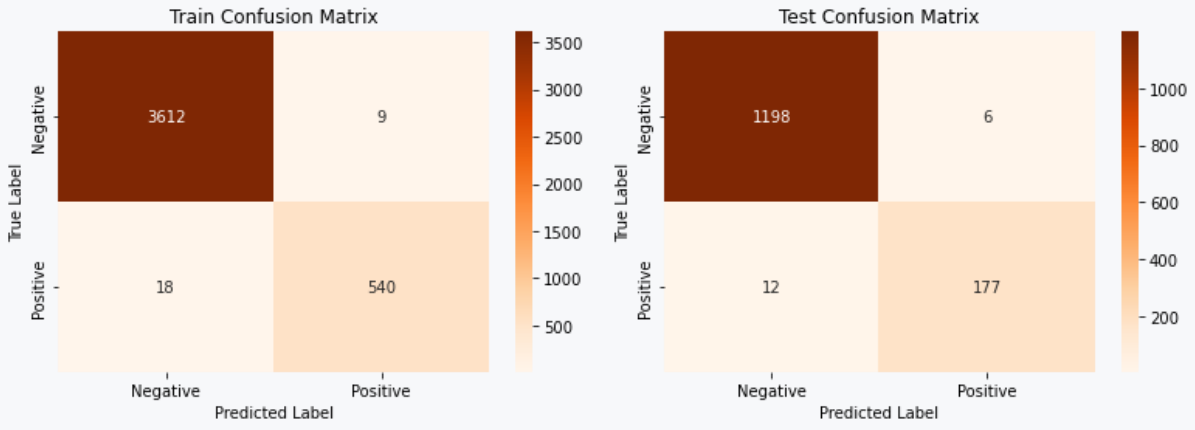
```
# Visualizing evaluation Metric Score chart
MultinomialNB_score = evaluate_model(clf, X_train, X_test,
y_train, y_test)
```

Program output:

```
Train ROC AUC: 0.9826282171205603
Test ROC AUC: 0.9657622739018088
```




```
Confusion Matrix:
```



```
Train Classification Report:
|              |   precision |   recall |   f1-score |
support |
|:-------------|------------:|---------:|-----------:|--------
----:|
| 0            |    0.995041 | 0.997514 |   0.996276 | 3621
|
| 1            |    0.983607 | 0.967742 |   0.97561  |  558
|
| accuracy     |    0.993539 | 0.993539 |   0.993539 |
0.993539 |
| macro avg    |    0.989324 | 0.982628 |   0.985943 | 4179
|
| weighted avg |    0.993514 | 0.993539 |   0.993517 | 4179
|

Test Classification Report:
|              |   precision |   recall |   f1-score |
support |
|:-------------|------------:|---------:|-----------:|--------
----:|
| 0            |    0.990083 | 0.995017 |   0.992543 | 1204
|
| 1            |    0.967213 | 0.936508 |   0.951613 |  189
|
| accuracy     |    0.987078 | 0.987078 |   0.987078 |
0.987078 |
| macro avg    |    0.978648 | 0.965762 |   0.972078 | 1393
|
| weighted avg |    0.98698  | 0.987078 |   0.98699  | 1393
|
```

Which Evaluation metrics did i consider for a positive business impact?

- After carefully considering the potential consequences of false positives and false negatives in the context of our business objectives, I have selected recall as the primary evaluation metric for our email spam detection model. Its gives 98.49% accuracy for recall test set.

```
# Defining a function for the Email Spam Detection System
def detect_spam(email_text):
    # Load the trained classifier (clf) here
    # Replace the comment with your code to load the
classifier model
```

```
    # Make a prediction using the loaded classifier
    prediction = clf.predict([email_text])

    if prediction == 0:
        return "This is a Ham Email!"
    else:
        return "This is a Spam Email!"
```

```
# Example of how to use the function
sample_email = 'Free Tickets for IPL'
result = detect_spam(sample_email)
print(result)
```

Program output:

```
This is a Spam Email!
```

Conclusion - key information

- **Dataset Distribution:** We found that about 13.41% of the messages were classified as spam while 86.59% were ham. This ratio provided a crucial starting point for our analysis and helped us understand the prevalence of spam in email communications.
- **Exploratory Data Analysis (EDA):** Through EDA, we identified commonly used words in spam messages such as "free", "call", "text", "txt" and "now". These keywords, often detected by spam filters, became important features of our model.
- **Model Selection and Performance:** Among the models we examined, the Multinomial Naive Bayes classifier stood out with an impressive 98.49% accuracy on the recall test set. This high recall means the model was excellent at catching spam, which is vital for email security and improves the user experience by keeping inboxes cleaner.

3.3.2

Project: SMS spam classifier

(by https://www.milindsoorya.co.uk/blog/build-a-spam-classifier-in-python)

In today's instant messaging world, SMS a IM spam is becoming a growing problem. As unwanted advertising messages, scams and phishing attempts are on the rise, it is essential to have effective tools to identify and filter these spam messages. In this project, we will develop a machine learning model to classify SMS/IM messages as spam or ham.

Our goal is to create a model that can analyze the content of an message and accurately predict whether it is spam. Machine learning models can learn patterns in the text itself, making them more adaptive and robust.

Used Spam Collection is a set of SMS tagged messages that have been collected for SMS Spam research. It contains one set of SMS messages in English of 5,574 messages, tagged according to being ham (legitimate) or spam. The data was obtained from UCI's Machine Learning Repository,

The local version is available at https://priscilla.fitped.eu/data/cybersecurity/spam/sms_spam_894.txt

The steps in the project will be focused on

Data processing

- Import packages
- Loading data
- Data set preprocessing and exploration
- Creating a word cloud to see which message is spam and which is not.
- Removing stop words and punctuation
- Convert text data to vectors

Creating a spam classification model for SMS

- Splitting data into train and test files
- Use built-in Sklearn classifiers to build models
- Training data on the model
- Making predictions based on new data

Import the required packages

```
import matplotlib.pyplot as plt
import csv
import sklearn
import pickle
from wordcloud import WordCloud
import pandas as pd
import numpy as np
import nltk
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import CountVectorizer,
TfidfTransformer
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import
GridSearchCV,train_test_split,StratifiedKFold,cross_val_score,
learning_curve
```

Loading the Dataset

```
data =
pd.read_csv('https://priscilla.fitped.eu/data/cybersecurity/spam/sms_spam_894.txt', encoding='latin-1', delimiter='\t',
header=None)
print(data.head())
```

Program output:

```
      0                                                  1
0   ham  Go until jurong point, crazy.. Available only ...
1   ham                      Ok lar... Joking wif u oni...
2  spam  Free entry in 2 a wkly comp to win FA Cup fina...
3   ham  U dun say so early hor... U c already then say...
4   ham  Nah I don't think he goes to usf, he lives aro...
```

Name the columns for better processing.

```
data.rename(columns={0: 'label', 1: 'text'}, inplace=True)
print(data.head())
```

Program output:

```
  label                                               text
0   ham  Go until jurong point, crazy.. Available only ...
1   ham                      Ok lar... Joking wif u oni...
2  spam  Free entry in 2 a wkly comp to win FA Cup fina...
3   ham  U dun say so early hor... U c already then say...
4   ham  Nah I don't think he goes to usf, he lives aro...
```

```
print(data['label'].value_counts())
```

Program output:

```
ham     4825
spam     747
Name: label, dtype: int64
```

Preprocessing and Exploring the Dataset

Build word cloud to see which message is spam and which is not

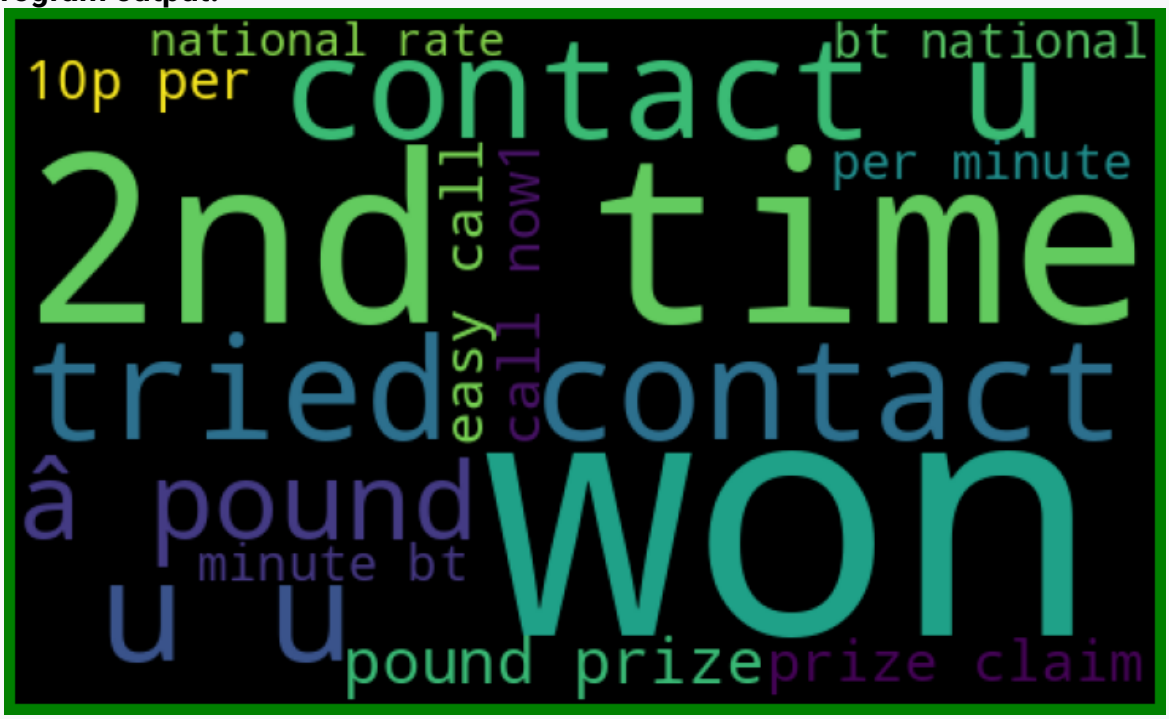
```
ham_words = ''
spam_words = ''
# Creating a corpus of spam messages
```

```
for val in data[data['label'] == 'spam'].text:
    text = val.lower()
    tokens = nltk.word_tokenize(text)
    for words in tokens:
        spam_words = spam_words + words + ' '

# Creating a corpus of ham messages
for val in data[data['label'] == 'ham'].text:
    text = text.lower()
    tokens = nltk.word_tokenize(text)
    for words in tokens:
        ham_words = ham_words + words + ' '
# Create Spam word cloud and ham word cloud.
spam_wordcloud = WordCloud(width=500,
height=300).generate(spam_words)
ham_wordcloud = WordCloud(width=500,
height=300).generate(ham_words)

#Spam Word cloud
plt.figure( figsize=(10,8), facecolor='w')
plt.imshow(spam_wordcloud)
plt.axis("off")
plt.tight_layout(pad=0)
plt.show()
```

Program output:


```
#Creating Ham wordcloud
plt.figure( figsize=(10,8), facecolor='g')
plt.imshow(ham_wordcloud)
plt.axis("off")
plt.tight_layout(pad=0)
plt.show()
```

Program output:



from the spam word cloud, we can see that "free" is most often used in spam.

Now, we can convert the spam and ham into 0 and 1 respectively so that the machine can understand.

```
data = data.replace(['ham','spam'],[0, 1])
print(data.head(10))
```

Program output:

```
   label                                                text
0      0  Go until jurong point, crazy.. Available only ...
1      0                      Ok lar... Joking wif u oni...
2      1  Free entry in 2 a wkly comp to win FA Cup fina...
3      0  U dun say so early hor... U c already then say...
4      0  Nah I don't think he goes to usf, he lives aro...
5      1  FreeMsg Hey there darling it's been 3 week's n...
6      0  Even my brother is not like to speak with me. ...
7      0  As per your request 'Melle Melle (Oru Minnamin...
8      1  WINNER!! As a valued network customer you have...
9      1  Had your mobile 11 months or more? U R entitle...
```

Removing punctuation and stopwords from the messages

- Punctuation and stop words do not contribute anything to our model, so we have to remove them. Using NLTK library we can easily do it.

```
#remove the punctuations and stopwords
import string
def text_process(text):

    text = text.translate(str.maketrans('', '',
string.punctuation))
    text = [word for word in text.split() if word.lower() not
in stopwords.words('english')]

    return " ".join(text)

data['text'] = data['text'].apply(text_process)
print(data.head(10))
```

Program output:

```
   label                                                text
0      0  Go jurong point crazy Available bugis n great ...
1      0                              Ok lar Joking wif u oni
2      1  Free entry 2 wkly comp win FA Cup final tkts 2...
```

```
3      0                 U dun say early hor U c already say
4      0        Nah dont think goes usf lives around though
5      1  FreeMsg Hey darling 3 weeks word back Id like ...
6      0     Even brother like speak treat like aids patent
7      0  per request Melle Melle Oru Minnaminunginte Nu...
8      1  WINNER valued network customer selected receiv...
9      1  mobile 11 months U R entitled Update latest co...
```

Now, create a data frame from the processed data before moving to the next step.

```
text = pd.DataFrame(data['text'])
label = pd.DataFrame(data['label'])
```

Converting words to vectors

We can convert words to vectors using either Count Vectorizer or by using TF-IDF Vectorizer.

TF-IDF is better than Count Vectorizers because it not only focuses on the frequency of words present in the corpus but also provides the importance of the words. We can then remove the words that are less important for analysis, hence making the model building less complex by reducing the input dimensions.

I have included both methods for your reference.

Converting words to vectors using Count Vectorizer

```
## Counting how many times a word appears in the dataset
from collections import Counter

total_counts = Counter()
for i in range(len(text)):
    for word in text.values[i][0].split(" "):
        total_counts[word] += 1

print("Total words in data set: ", len(total_counts))
```

Program output:

```
Total words in data set:  11426
# Sorting in decreasing order (Word with highest frequency
appears first)
vocab = sorted(total_counts, key=total_counts.get,
reverse=True)
print(vocab[:60])
```

Program output:

```
['u', '2', 'call', 'U', 'get', 'Im', 'ur', '4', 'ltgt',
'know', 'go', 'like', 'dont', 'come', 'got', 'time', 'day',
'want', 'Ill', 'lor', 'Call', 'home', 'send', 'one', 'going',
'need', 'Ok', 'good', 'love', 'back', 'n', 'still', 'text',
'im', 'later', 'see', 'da', 'ok', 'think', 'Ã¼', 'free',
'FREE', 'r', 'today', 'Sorry', 'week', 'phone', 'mobile',
'cant', 'tell', 'take', 'much', 'night', 'way', 'Hey',
'reply', 'work', 'give', 'make', 'new']
```

```
# Mapping from words to index
vocab_size = len(vocab)
word2idx = {}
#print vocab_size
for i, word in enumerate(vocab):
    word2idx[word] = 1

# Text to Vector
def text_to_vector(text):
    word_vector = np.zeros(vocab_size)
    for word in text.split(" "):
        if word2idx.get(word) is None:
            continue
        else:
            word_vector[word2idx.get(word)] += 1
    return np.array(word_vector)

# Convert all titles to vectors
word_vectors = np.zeros((len(text), len(vocab)),
dtype=np.int_)
for i, (_, text_) in enumerate(text.iterrows()):
    word_vectors[i] = text_to_vector(text_[0])

print(word_vectors.shape)
```

Program output:

```
(5572, 11426)
```

Converting words to vectors using TF-IDF Vectorizer

```
#convert the text data into vectors
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer()
```

```
vectors = vectorizer.fit_transform(data['text'])
print(vectors.shape)
```

Program output:

```
(5572, 9459)
```

```
# You can choose type of converted data
#features = word_vectors
features = vectors
```

Splitting into training and test set

```
#split the dataset into train and test set
X_train, X_test, y_train, y_test = train_test_split(features,
data['label'], test_size=0.15, random_state=111)
```

Classifying using sklearn's pre-built classifiers

- In this step we will use some of the most popular classifiers out there and compare their results.

```
#import sklearn packages for building classifiers
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.naive_bayes import MultinomialNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

#initialize multiple classification models
svc = SVC(kernel='sigmoid', gamma=1.0)
knc = KNeighborsClassifier(n_neighbors=49)
mnb = MultinomialNB(alpha=0.2)
dtc = DecisionTreeClassifier(min_samples_split=7,
random_state=111)
lrc = LogisticRegression(solver='liblinear', penalty='l1')
rfc = RandomForestClassifier(n_estimators=31,
random_state=111)

#create a dictionary of variables and models
clfs = {'SVC' : svc,'KN' : knc, 'NB': mnb, 'DT': dtc, 'LR':
lrc, 'RF': rfc}

#fit the data onto the models
```

```
def train(clf, features, targets):
    clf.fit(features, targets)

def predict(clf, features):
    return (clf.predict(features))

pred_scores_word_vectors = []
for k,v in clfs.items():
    train(v, X_train, y_train)
    pred = predict(v, X_test)
    pred_scores_word_vectors.append((k, [accuracy_score(y_test
, pred)]))
```

Predictions using TFIDF Vectorizer algorithm

```
print(pred_scores_word_vectors)
```

Program output:

```
[('SVC', [0.9784688995215312]), ('KN', [0.9342105263157895]),
('NB', [0.9832535885167464]), ('DT', [0.9629186602870813]),
('LR', [0.9509569377990431]), ('RF', [0.9772727272727273])]
```

Model predictions

```
#write functions to detect if the message is spam or not
def find(x):
    if x == 1:
        print ("Message is SPAM")
    else:
        print ("Message is NOT Spam")

newtext = ["Free entry"]
integers = vectorizer.transform(newtext)

x = mnb.predict(integers)
find(x)
xx = knc.predict(integers)
find(xx)
```

Program output:

```
Message is SPAM
Message is SPAM
```

Check Classification Results with Confusion Matrix

```
# insert code
```

🕮 3.3.3

Support Vector Machine

Support Vector Machine (SVM) is a supervised machine learning algorithm that is widely used for classification and regression tasks, especially in high-dimensional spaces. SVM works by finding the optimal hyperplane that best separates data points of different classes with maximum margin. This "margin" is the distance between the hyperplane and the nearest data points from each class, known as support vectors. The larger the margin, the better the generalization of the classifier to new data.

How SVM works in practice

For classification, SVM creates a decision boundary between two classes. In higher dimensions, it still tries to separate classes by finding the best boundary with maximum margin.

For regression (known as SVR - Support Vector Regression), the SVM finds a line or plane that fits within a specified margin around the data points.

Advantages and applications

- High dimensionality: SVM is efficient in high dimensional spaces and is memory efficient because it only uses support vectors in the decision function.
- Versatility: With different kernel functions, SVM can be applied to different types of data, including non-linearly separable data.
- Applications: Commonly used in text classification, image recognition, bioinformatics, and others where it is crucial to classify data with high accuracy.

SVM is powerful but may require careful tuning of parameters and can be computationally intensive for very large datasets. Nevertheless, its robustness and flexibility make it a popular choice in many machine learning tasks.

3.3.4

Project: Super Vector Machine

(by https://medium.com/@Coursesteach/spam-detection-using-machine-learning-methods-dd5dbc799b6b)

This project again brings a different approach with an emphasis on the preparation of a suitable dataset through several standard adjustments.

The basic sequence of steps remains of course unchanged.

Dataset:

- original https://raw.githubusercontent.com/Sanjay-dev-ds/spam_ham_email_detector/master/spam.csv
- local: https://priscilla.fitped.eu/data/cybersecurity/spam/spam_909.csv

Lets go to start!

Import libraries

```
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split ,
GridSearchCV , KFold
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score ,
classification_report , confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
import re
import nltk
from nltk.stem import PorterStemmer
from sklearn import metrics
```

Data Loading

```
df =
pd.read_csv("https://priscilla.fitped.eu/data/cybersecurity/spam/spam_909.csv", encoding= 'latin-1')
print(df.head())
```

Program output:

```
  Label                                              EmailText
0   ham  Go until jurong point, crazy.. Available only ...
1   ham                      Ok lar... Joking wif u oni...
2  spam  Free entry in 2 a wkly comp to win FA Cup fina...
3   ham  U dun say so early hor... U c already then say...
4   ham  Nah I don't think he goes to usf, he lives aro...
```

Data Preprocessing

- Remove duplicate values

```
df = df.drop_duplicates(keep='first')
```

Independent and dependent variables

To divide the data set into independent and dependent variables for training a spam detection model, we can define a dependent variable (the target we want to predict) and an independent variable (the features we will use for prediction).

Independent and dependent variables

Dependent variable (goal):

- Class: This column indicates whether the message is spam or not (eg "spam" or "ham").

Independent variable (function):

- Message: This column contains the actual text of the messages that we will analyze to determine if they are spam.

```
x = df['EmailText'].values
y = df['Label'].values
```

Text Pre-Processing

We will create a function to preprocess the text by converting it to lowercase, removing special characters, normalizing certain words, and applying stemming using the Porter Stemmer algorithm. This process will help ensure that our text data is clean and consistent, making it more suitable for analysis and modeling.

```
porter_stemmer=PorterStemmer()
def preprocessor(text):
    text=text.lower()
    text=re.sub("\W"," ",text)
```

```
    text=re.sub("\s+(in|the|all|for|and|on)\s+"," _connector_
",text)
    words=re.split("\s+",text)
    stemmed_words=[porter_stemmer.stem(word=word) for word in
words]
    return ' '.join(stemmed_words)
```

We will create a tokenizer function that performs two key tasks: it will add spaces around special characters and then split the text based on whitespace. This will help break down the text into individual tokens, making it easier to analyze and process further.

```
# new
def tokenizer(text):
    text=re.sub("(\W)"," \1 ",text)
    return re.split("\s+",text)
```

Feature Extraction

To use text data for predictions, we need to break it down and remove unnecessary words through a tokenization. After tokenization, we convert the remaining words into numerical values, either as integers or floating-point numbers, so they can be utilized in machine learning. This process is known as feature extraction (or vectorization).

One effective tool for this purpose is the **CountVectorizer** from Scikit-learn. It transforms a collection of text documents into a numerical representation by counting the occurrences of words. Additionally, it allows for text cleaning before conversion, making it a valuable asset for handling text data.

CountVectorizer converts a text corpus into a vector of terms, and we can customize its behavior with various parameters:

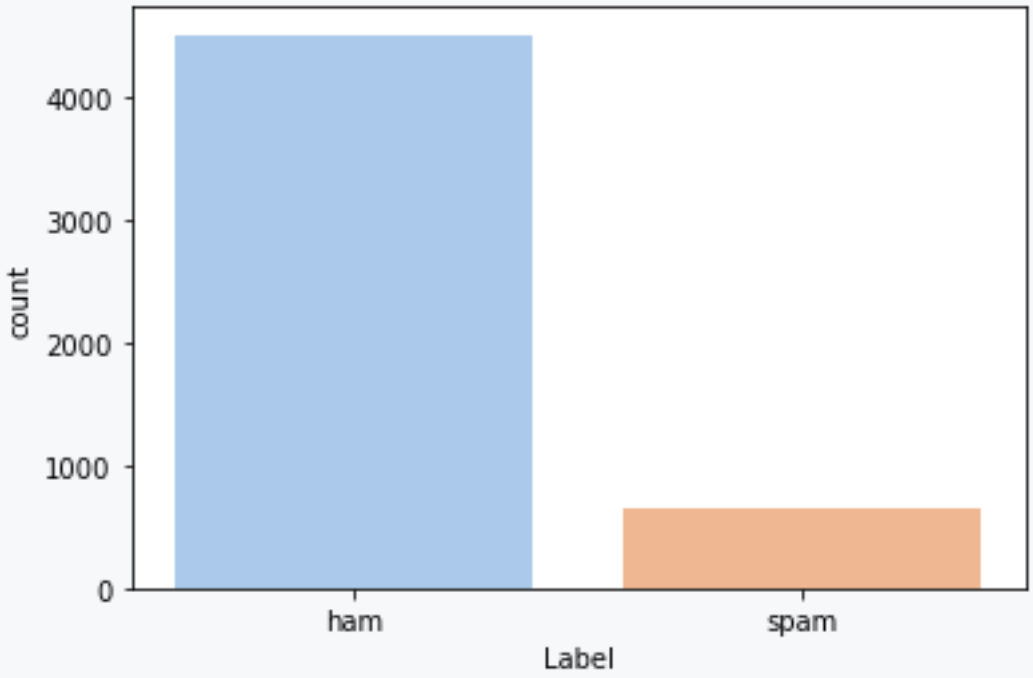
- **min_df = 0.06:** This parameter ensures that we only include words that appear in at least 6% of the documents, filtering out infrequent terms.
- **ngram_range = (1, 2):** This setting allows the extraction of both unigrams (single words) and bigrams (pairs of consecutive words), providing richer information from the text.

```
vectorizer =
CountVectorizer(tokenizer=tokenizer,ngram_range=(1,2),min_df=0
.006,preprocessor=preprocessor)
x  = vectorizer.fit_transform(x)
```

Data balance check

```
sns.countplot(data=df, x='Label', palette='pastel')
```

Program output:



To solve the problem of **uneven distribution** in the target class, we use a random resampling method to balance the observations of the target variable. This is a random duplication of examples in the minority class, which in this case is "Spam".

A random resampling process

- Identify the class distribution: First, we check our class distribution to confirm the imbalance between "ham" and "spam".
- Implement random resampling: We create a balanced dataset by randomly duplicating instances of the "spam" class until the number of "spam" messages matches the number of "ham" messages.
- Combine datasets: Finally, we combine the resampled "spam" instances with the original "ham" instances to create a balanced dataset.

```
from imblearn.under_sampling import NearMiss
from collections import Counter
from imblearn.over_sampling import RandomOverSampler

ros = RandomOverSampler(random_state=42)

print('Original dataset shape', Counter(y))
```

```
# fit predictor and target
x,y = ros.fit_resample(x, y)

print('Modified dataset shape', Counter(y))
```

Program output:

```
Original dataset shape Counter({'ham': 4516, 'spam': 653})
Modified dataset shape Counter({'ham': 4516, 'spam': 4516})
```

Data Splitting

```
x_train , x_test , y_train , y_test   = train_test_split(x, y,
test_size =0.2,random_state = 0)
```

Model Training and Testing

- MultinominalNB

```
MultinomialNB()
clf = MultinomialNB()
clf.fit(x_train,y_train)

# Accuracy
y_pred_NB = clf.predict(x_test)
NB_Acc=clf.score(x_test, y_test)
print('Accuracy score= {:.4f}'.format(clf.score(x_test,
y_test)))
```

Program output:

```
Accuracy score= 0.9590
```

Let's test this model by taking a user input as a message to detect whether it is spam or not:

```
input_message = input('Enter a message:')
# Step 1: Preprocess the input message
processed_message = preprocessor(input_message)

# Step 2: Transform the processed message using the same
vectorizer used for training
# Assuming `vectorizer` is your trained CountVectorizer
vectorized_message = vectorizer.transform([processed_message])

# Step 3: Make a prediction using the trained model
# Assuming `model` is your trained Naive Bayes classifier
prediction = clf.predict(vectorized_message)
```

```
# Display the result
if prediction[0] == 'spam':
    print("The message is classified as: Spam")
else:
    print("The message is classified as: Ham")
```

Program output:

```
Enter a message: i am not a spam but you can win in lotteryThe
message is classified as: Ham
```

- SVM approach

```
# Initialize the model
model = SVC(C=1, kernel='linear')

# Fit the model on the training data
model.fit(x_train, y_train)

# Accuracy
accuracy = metrics.accuracy_score(y_test,
model.predict(x_test))
accuracy_percentage = 100 * accuracy
print(accuracy_percentage)
```

Program output:

```
98.83785279468734
```

Hyperparameter Optimization using Grid Search CV

MultinomialNB

```
from sklearn.model_selection import KFold, GridSearchCV
from sklearn.naive_bayes import MultinomialNB
params = {
    'alpha': [0.1, 0.5, 1.0],  # Different values for alpha
    'fit_prior': [True, False]  # Whether to fit class prior
probabilities
}

cval = KFold(n_splits=2)
model = MultinomialNB()  # Using Multinomial Naive Bayes
TunedModel1 = GridSearchCV(model, params, cv=cval)
TunedModel1.fit(x_train, y_train)
```

```
accuracy = metrics.accuracy_score(y_test,
TunedModel1.predict(x_test))
accuracy_percentage = 100 * accuracy
print(accuracy_percentage)
```

Program output:

```
96.1261759822911
```

SVM

```
GridSearchCV(cv=KFold(n_splits=2, random_state=None,
shuffle=False),
             estimator=SVC(),
             param_grid={'C': [0.2, 0.5], 'kernel': ['linear',
'sigmoid']})

params  = {"C":[0.2,0.5] , "kernel" : ['linear', 'sigmoid'] }
cval = KFold(n_splits = 2)
model =  SVC();
TunedModel = GridSearchCV(model,params,cv= cval)
TunedModel.fit(x_train,y_train)

accuracy = metrics.accuracy_score(y_test,
TunedModel.predict(x_test))
accuracy_percentage = 100 * accuracy
print(accuracy_percentage)
```

Program output:

```
99.0038738240177
```

Explanation

- **GridSearchCV**: This class is used for hyperparameter tuning. It exhaustively searches through a specified parameter grid to find the best combination of hyperparameters for a given model.
- **cv=KFold(n_splits=2, random_state=None, shuffle=False)**: This specifies the cross-validation strategy.

1. KFold: This is a cross-validator that divides the dataset into n_splits (in this case, 2) parts.
2. The data will be split into two subsets for cross-validation. The model will be trained on one subset and validated on the other, and this process will be repeated, swapping the training and validation sets.

3. random_state=None: This means that the splitting will not be random; it will use the default behavior of KFold.
4. shuffle=False: This means that the data will not be shuffled before splitting.

- **estimator=SVC()**: This specifies the machine learning model to be used for tuning—in this case, a Support Vector Classifier.
- **param_grid={'C': [0.2, 0.5], 'kernel': ['linear', 'sigmoid']}**: This is the dictionary that defines the hyperparameters to be tuned and the values to be tested for each hyperparameter.

1. 'C': This parameter controls the trade-off between achieving a low training error and a low testing error. It can take values of 0.2 or 0.5.
2. 'kernel': This defines the type of kernel function to be used in the algorithm. In this case, it will test both a linear kernel and a sigmoid kernel.

What it Does:

- The **GridSearchCV** object will test all combinations of the specified parameters:
- C = 0.2 with kernel = linear
- C = 0.2 with kernel = sigmoid
- C = 0.5 with kernel = linear
- C = 0.5 with kernel = sigmoid
- For each combination, it will perform cross-validation (using KFold with 2 splits) to evaluate the model's performance.

Final Output:

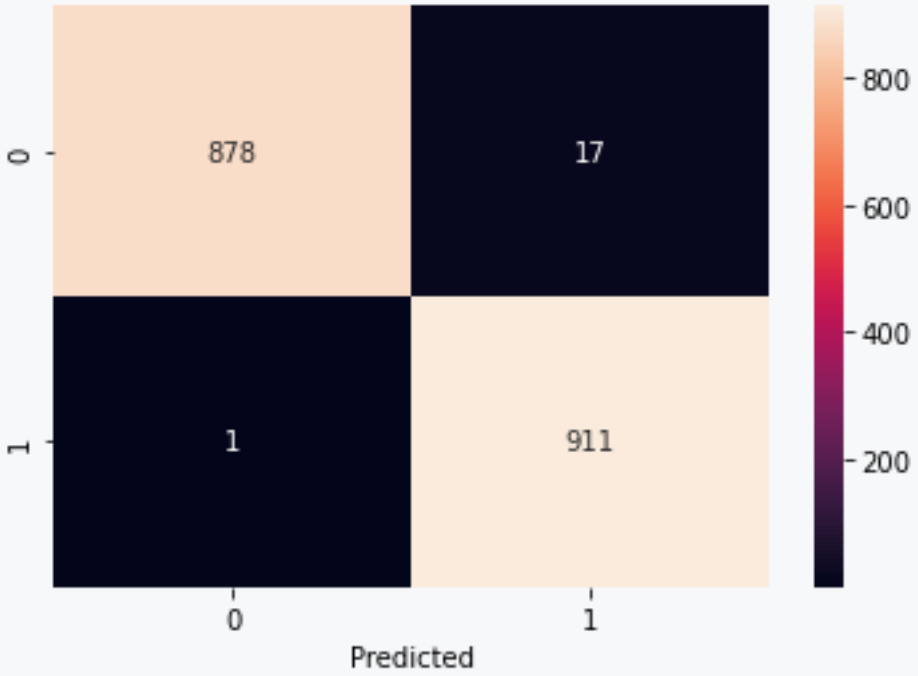
- The end result will be a fitted **GridSearchCV** object that contains the best combination of hyperparameters based on the model's performance across the validation sets. You can retrieve the best parameters using **grid_search.best_params_** and the best score with **grid_search.best_score_**.

Model Evaluation

Confusion-svm

```
sns.heatmap(confusion_matrix(y_test,TunedModel.predict(x_test)
),annot = True , fmt ="g")
plt.xlabel("Predicted")
plt.show("Actual")
plt.show()
```

Program output:



3.3.5

Logistic regression

Logistic regression is a popular and effective tool in data science, especially when it comes to solving classification problems. Known as the “workhorse” of machine learning, it is valued for being simple and reliable.

What is logistic regression?

Logistic regression is a supervised machine learning technique that helps categorize data into two groups, assuming there is a relationship between the input features and the output. Think of it as a sorting tool that classifies data into one of two categories, such as "spam" or "not spam". It is designed for situations where there are only two possible outcomes, often labeled as "yes" or "no" or "0" and "1".

Unlike linear regression, which is also based on relationships between variables but predicts continuous outcomes, logistic regression predicts the probability that an outcome will be in one category. For example, it might assign a 90% chance that the email is spam or a 2% chance that it will be important.

Logistic regression does not just provide a simple answer. Instead, it calculates the probability that an instance belongs to one group over another. This likelihood-based approach makes logistic regression powerful and practical for applications such as spam detection, disease prediction, and more.

The ability of logistic regression to estimate probabilities makes it an invaluable tool for many real-world applications. Here are some key examples:

- Spam Filtering: Logistic regression can effectively classify emails as spam or not, helping to keep spam out of your inbox.
- Fraud detection: Banks use logistic regression to identify suspicious transactions to help protect customers' finances.
- Loan approval: Financial institutions can evaluate the applicant's likelihood of repaying the loan, making the approval process more reliable.
- Medical diagnosis: Doctors can use it to assess the likelihood of a disease based on symptoms, which helps in early and accurate diagnosis.
- Predicting customer churn: Businesses use logistic regression to predict which customers may stop using their services, enabling proactive customer retention efforts.

How does logistic regression make predictions?

Now that you understand the basics, let's go explain how logistic regression actually works and the basic steps to prepare data to build a model.

1. Data preparation

Proper data preparation is critical to creating an accurate logistic regression model. The performance of the model depends largely on the quality of the data it is trained on. For example, if you are building a spam filter, using data that is not relevant to spam emails (such as disease prediction data) will lead to poor results. Here are the main steps of data preparation:

- Features: These are the characteristics or attributes used to make predictions. The following functions can be important in spam filtering: the words in the email subject, sender information, the presence of attachments. Choosing the right features helps the model distinguish between spam and non-spam messages. Including irrelevant features can confuse the model and reduce accuracy.
- Labels: Labels indicate the correct category for each data point. In spam filtering, the labels are simply "spam" or "not spam" for each email, which the model learns from during training.
- Data cleaning: Like cleaning ingredients before brewing, data preparation involves handling missing values, correcting inconsistencies, and correcting typos. Clean data ensures that the model learns accurately without being misled by errors.

Logistic regression equation

Let's look at the basic math behind logistic regression. Logistic regression combines the features of our data into a formula that assigns weight (importance) to each feature. It works like this:

- Each attribute is multiplied by its assigned weight.
- These weighted values are then summed.
- The sum provides a score that represents the probability that a particular data point (e.g. email) belongs to a particular class (e.g. "spam").

This score is converted to a probability, and logistic regression uses this probability to predict whether a data point belongs to one class or the other (eg spam or not). By training on labeled data, logistic regression adjusts the weights of each feature to improve its predictions.

Logistic regression uses an equation where the inputs are combined linearly using weights or coefficient values to predict the modeled output, but here the result is a binary value (0 or 1).

Equation for logistic regression:

$$y = \frac{e^{b_0+b_1x}}{1 + e^{b_0+b_1x}}$$

Where:

- x - input value
- y - predicted output
- b0 - bias or intercept
- b1 - coefficient for input (x)

This score of the model isn't directly our probability yet. Logistic regression uses the sigmoid function to map predicted values to probabilities and also convert the value into a range between 0 and 1.

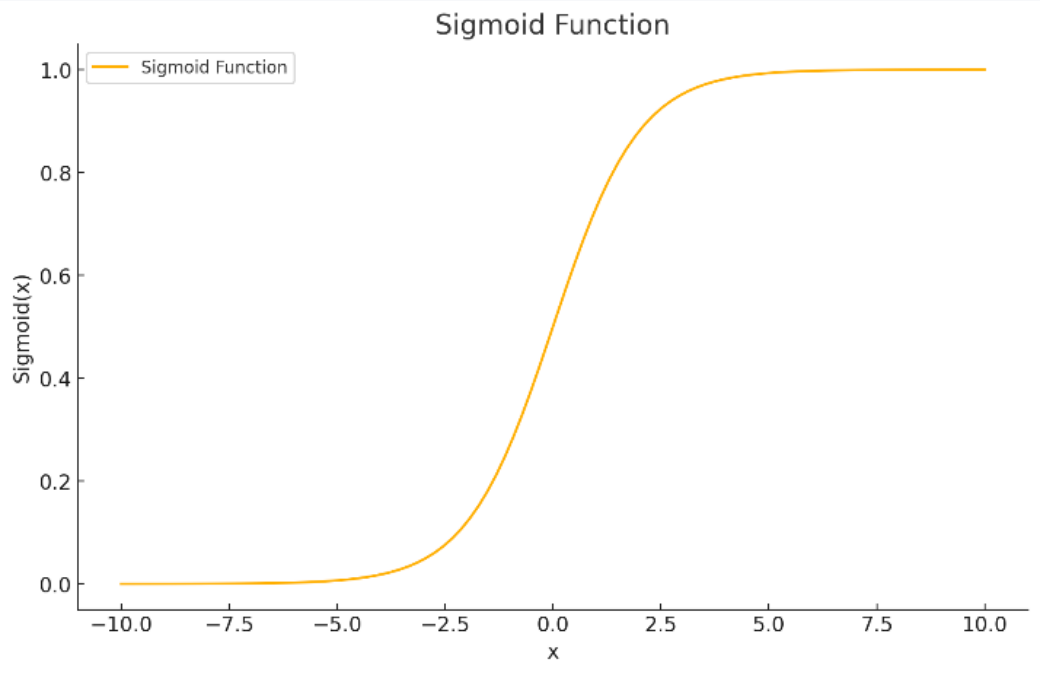
Logistic regression uses the concept of the threshold value for instance 0.5, where:

1. values below 0.5 get squashed towards 0 (very unlikely spam)
2. values above get pushed towards 1 (almost definitely spam)

$$f(x) = \frac{1}{1 + e^{-x}}$$

where

- e - base of natural logarithms
- x - numerical value to be transformed



2. Model training

In this step, the model is "learned" by analyzing labeled data, such as whether emails are marked as spam or not. It compares its predictions to these actual labels, and when it makes mistakes, it adjusts its internal weights to improve accuracy. This entire process is called optimization, where the model is refined based on errors, ultimately increasing its ability to make accurate predictions.

3. Making predictions

After training, the model is ready to classify new emails. For each email, it calculates a score using the same formula and uses a sigmoid function to convert that score into a probability. We can then set a threshold, such as 70%, for classification. If the model calculates a probability above this threshold that the email is spam, we classify it as spam.

4. Model evaluation

To check the performance of the model, we use metrics that provide insight into how well it is doing.

- Accuracy shows the overall percentage of correct predictions.
- Accuracy tells us how many emails classified as spam were actually spam.
- Recall shows the percentage of real spam emails that the model correctly identified.

It is essential to evaluate the model on a separate test set that was not used in training to get an unbiased assessment of its performance. This is similar to giving the student a new test instead of repeating the one they practiced with.

3.3.6

Project: Build a Logistic Regression Model

(by https://dev.to/oluwadamisisamuel1/how-to-build-a-logistic-regression-model-a-spam-filter-tutorial-261b)

Based on the previous explanation, we can now proceed to the creation of a concrete model

Dataset:

- original https://raw.githubusercontent.com/Sanjay-dev-ds/spam_ham_email_detector/master/spam.csv
- local: https://priscilla.fitped.eu/data/cybersecurity/spam/spam_909.csv

Lets go to start!

1. Import libraries

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
```

2. Load and prepare data

To make this as simple as possible we have a simple dataset with two columns, "EmailText" containing the email text, and "Label" indicating spam ("spam") or not spam ("ham").

We read data from a CSV file using pandas.read_csv.

train_test_split splits the data into training and test sets, ensuring that the model generalizes well to unseen data. The test_size parameter controls the size of the test suite (20% in this case).

```
data =
pd.read_csv("https://priscilla.fitped.eu/data/cybersecurity/spam/spam_909.csv")

# Split data into training and testing sets
X_train, X_test, y_train, y_test =
train_test_split(data["EmailText"], data["Label"],
test_size=0.2, random_state=42)
```

3. Feature engineering

Since our model works with numeric data, we need to convert text emails into functions. We will use a technique called TF-IDF (Term Frequency-Inverse Document Frequency), which takes into account the importance of each word in the document.

We will create a TfidfVectorizer object and use it to fit (learn the vocabulary) and transform the training data.

The transformed data (X_train_features) now contains numeric characters representing the importance of each word in each email. We repeat the same process for the test data (X_test_features).

```
# Create a TF-IDF vectorizer
vectorizer = TfidfVectorizer()

# Transform training and testing data into TF-IDF features
X_train_features = vectorizer.fit_transform(X_train)
X_test_features = vectorizer.transform(X_test)
```

4. Train the model

We will create a LogisticRegression object representing the model.

We use the fit method to train the model on prepared training features (X_train_features) and labels (y_train). During this process, the model learns the relationships between features and spam/non-spam labels and adjusts its internal weights.

```
# Create a logistic regression model
model = LogisticRegression()

# Train the model on the training data
model.fit(X_train_features, y_train)
```

5. Make predictions

We use the trained model to predict labels (spam/not spam) for unseen test data using a prediction method.

We calculate the accuracy of model predictions using the accuracy_score function.

```
# Predict labels for the test data
y_pred = model.predict(X_test_features)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
```

```
print("Accuracy:", accuracy)
```

Program output:

```
Accuracy: 0.9659192825112107
```

6. Interpretation of results

The output shows the accuracy of the model. While this is a decent starting point, it's important to note that accuracy alone may not be the most informative metric in all situations, especially when dealing with unbalanced data sets (where one class, such as spam, may be much smaller than the other).

Key features and limitations of logistic regression

Logistic regression is a versatile tool, but it is important to recognize its limitations:

- Assumptions: Assumes a linear relationship between properties and result. If the data shows non-linearity, the model may have problems. It shares this limitation with another regression model "Linear Regression". Nonparametric models such as decision trees or kernel methods such as support vector machines can handle such complexities.
- Overfitting: Overly complex models or models trained on limited data may become too specific to the training data and perform poorly on unseen data. Regularization techniques such as L1 or L2 regularization can help alleviate this problem by penalizing models with high complexity.
- Binary classification: Logistic regression is designed for problems with two result categories (eg spam/not spam). For multi-class problems (eg classification of different flower species) you may need to explore models such as multinomial logistic regression or random forests.

Phishing Protection

Chapter 4

4.1 Introduction into phishing

🕮 4.1.1

Phishing is a common type of cyber attack where attackers try to steal sensitive information like login credentials, credit card numbers, and other personal details. In phishing, the attacker poses as a trusted organization, convincing the victim to open an email, SMS, or visit a fake website.

The main goal of phishing is to trick the victim into willingly sharing their sensitive information. Attackers often use several techniques, including:

- **Fake Websites**: Attackers create sites that look like real ones (e.g., banks, social networks) to make people believe they're on a trusted site.
- **Phishing Emails**: These emails appear to come from reliable sources and contain links to fake websites designed to capture personal information.
- **Smishing (SMS Phishing)**: Attackers send fake SMS messages with links to malicious pages or requests for personal data.
- **Vishing (Voice Phishing)**: Attackers make phone calls pretending to be from a bank or other institution, asking the victim to "verify" personal information.

Phishing can happen in many forms, but it always involves pretending to be a legitimate source to gain a victim's trust and steal their information.

4.1.2

Fill in the blanks with the correct term to complete each sentence:

- _____ _____: Attackers create sites that look like real ones (e.g., banks, social networks) to make people believe they're on a trusted site.

- _____ _____: These emails appear to come from reliable sources and contain links to fake websites designed to capture personal information.

- _____ (SMS phishing): Attackers send fake SMS messages with links to malicious pages or requests for personal data.

- _____ (Voice phishing): Attackers make phone calls pretending to be from a bank or other institution, asking the victim to "verify" personal information.

- Smishing
- Fake
- websites
- emails
- Phishing
- Vishing

🕮 4.1.3

Phishing and spam are both forms of unwanted digital communication that are often used in cyber attacks, but they have different goals and methods:

Purpose

- Spam: Includes primarily unsolicited messages, usually promotional, that attempt to sell products or services. Spam is not necessarily intended to harm the recipient, but it can be annoying as it fills the inbox with unwanted advertisements.
- Phishing: Specially designed to trick the recipient into sharing sensitive information such as passwords, credit card numbers, or other personal information. Phishing is a malicious act aimed at causing harm, often resulting in identity theft or financial loss.

Techniques

- Spam: Often involves sending mass emails to a wide audience without personal targeting. These emails may advertise products, services or links to legitimate websites.
- Phishing: Phishing emails or messages, which are often more targeted and deceptive, are made to appear to come from trusted sources (eg banks, social media). They often contain links to fake websites that mimic legitimate ones in order to capture the victim's information.

Overlay

- Phishing attacks can sometimes be disguised as spam messages. For example, an email that appears to promote a new service may contain a malicious link or attachment. Spam filters are generally designed to detect and block these emails, but sophisticated phishing emails can bypass standard spam filters.

Impact

- Spam: Although it is annoying, the main risk is a waste of time and a cluttered inbox.
- Phishing: Much more dangerous because it directly threatens the victim's security and privacy by attempting to collect sensitive data.

In summary, spam is broad and usually has a harmless intent, while phishing is a targeted, malicious subset of spam that uses deception to obtain sensitive information from the recipient.

4.1.4

What is a key difference between spam and phishing emails?

- Spam is usually bulk, unsolicited advertising, whereas phishing is designed to deceive users into sharing sensitive information.
- Spam is usually bulk, unsolicited advertising, whereas phishing is designed to deceive users into sharing sensitive information.
- Spam emails typically contain links, while phishing emails never do.

4.1.5

How does the intent of phishing differ from that of typical spam?

- Phishing aims to collect personal data under false pretenses, while spam primarily advertises products or services.
- Phishing messages are usually harmless, while spam is always malicious.
- Phishing messages are only sent via SMS, while spam is only sent via email.

4.1.6

According to another point of view, phishing attacks can be divided into categories, according to specific methods and goals. Here's an overview:

- Spear Phishing: This is a highly targeted attack targeting a specific individual or organization. The attacker collects detailed information about the victim in order to send a personalized email or message that looks particularly convincing. Unlike general phishing, spear phishing focuses on making the message look credible in order to increase the chance of success.
- Whaling: A specialized form of spear phishing that targets executives and high-level executives, often referred to as "big fish." Because these individuals have access to sensitive company information, they are valuable targets for attackers.
- Spoofing: This technique involves disguising communications from an unknown or malicious source to appear as if they are from a trusted source. Spoofing can involve emails, phone calls, websites, or even technical aspects like your computer's IP address. Its goal is to trick victims into trusting and interacting with the attacker's message or website.
- Smishing: This form of phishing occurs via SMS (text) messages, where attackers attempt to obtain unauthorized data by sending fake alerts or links that direct the victim to a malicious website or prompt them for personal information.
- Vishing: Similar to smishing, but involves voice calls instead. Attackers impersonate trusted sources, such as a bank or government agency, and attempt to obtain sensitive information over the phone. This often involves asking the victim to "verify" personal information.

Phishing attacks are constantly evolving and becoming more sophisticated. Therefore, it is essential to remain cautious and apply basic security measures such as verifying sender information, avoiding suspicious links and using multi-factor authentication to reduce the risk of falling victim to phishing.

4.1.7

Match the following scenarios with the correct type of phishing attack.

- An attacker sends a fake SMS message with a link to a malicious website asking for personal data. This type of attack is called _____.

- An attacker disguises their email address to look like it's from a trusted source, tricking recipients into responding. This tactic is known as _____.

- An attacker targets a high-level executive or senior manager with a phishing attack to obtain confidential business data. This attack type is called _____.

- An attacker sends a highly personalized email to a specific individual, like a company employee, to gain sensitive information. This is known as _____.

- An attacker calls pretending to be a representative from a bank, asking the victim to "verify" personal details. This technique is known as _____.

- spear phishing
- whaling
- vishing
- spoofing
- smishing

4.1.8

Which of the following are effective security measures to protect against phishing attacks?

- Verifying the sender's email address
- Using multi-factor authentication
- Avoiding clicking on suspicious links
- Sharing personal information only with verified contacts
- Ignoring all messages from unknown sources
- Downloading files from unfamiliar sources

4.2 Fake web sites

📖 4.2.1

Creating fake websites in phishing attacks

One of the most common phishing techniques is the creation of fake websites designed to capture sensitive information from users. Attackers create websites that closely mimic the look and feel of legitimate sites, such as those of banks, online stores, or social media platforms. The goal of these fake websites is to trick users into entering confidential information such as login information, credit card numbers, or social security numbers.

How attackers create fake websites:

- Skinning: Attackers replicate HTML code, images, and other design elements from legitimate sites to create nearly identical fake sites. This helps make the fake page look trustworthy, which often fools users who don't pay much attention to detail.
- URL manipulation: Attackers use URLs that look like legitimate site addresses but contain minor changes or typos (eg "g00gle.com" instead of "google.com"). This tactic takes advantage of small differences that can be easily overlooked.
- Look-Alike Domains: Attackers register domains that look like legitimate domains by changing the domain ending (eg using ".net" instead of ".com") or making minor changes such as adding or removing a single letter. This can make a fake page look very close to the original, making it more likely to trick users.
- Redirection: Redirection methods are used so that clicking on a link in a phishing email takes the victim to a fake page, even if the link appears to lead to a legitimate address. This trick relies on the user trusting the displayed URL in the email.
- Hide the real URL: Attackers use JavaScript or other techniques to mask the real URL in the browser's address bar, making the fake URL look like it belongs to a trusted site. This scam makes it more likely that the user will feel safe on the fake site.

📝 4.2.2

Fill in the gaps:

Match each scenario with the correct technique used in creating fake websites in phishing attacks.

- Attackers replicate the HTML code and design elements of a legitimate website to build a visually similar fake page. This method is called _____.

- Attackers register a domain with a slight change in spelling or an alternative ending, like ".net" instead of ".com", to mimic a legitimate site. This tactic is known as _____.

- The URL in the phishing email looks legitimate, but clicking it redirects the user to a fake page. This method is called _____.

- redirecting
- skin copying
- look-alike domains

📝 4.2.3

Which of the following techniques involves altering the appearance of the browser's address bar to display a fake URL instead of the real one?

- Hiding the real URL
- URL manipulation
- Skin copying
- Look-alike domains

📝 4.2.4

URL manipulation in phishing relies on changing the actual content of the legitimate website to deceive users.

- True
- False

🕮 4.2.5

Why fake websites are effective at phishing?

- Trusted appearance: Fake websites are designed to look almost identical to legitimate sites, making them difficult to distinguish from the real ones. By using similar logos, layouts, and colors, attackers create a sense of trust that can lead users to believe they are on a safe, familiar site.
- Urgency and psychological manipulation: Phishing messages often create a sense of urgency and compel users to act quickly. Common tactics include messages urging the recipient to "verify account details", "confirm payment" or "immediately update personal information". This psychological manipulation makes users more likely to overlook warning signs and enter sensitive information.
- Lack of user vigilance: Many users don't check URLs or take extra steps to confirm the legitimacy of a site, which can leave them vulnerable. Attackers rely on this tendency because they know that users are less likely to notice small inconsistencies that reveal a site as fake.

📝 4.2.6

Fill in the blank:

Match each statement with a reason why fake websites are effective in phishing attacks.

- Fake websites are often visually similar to real websites, making it difficult for users to tell the difference. This tactic is called _____.

- Phishing messages often contain urgent calls to action, making users feel the need to act quickly. This technique is known as _____.

- Many users fail to check the URL of a website, making it easy for them to be fooled by a fake page. It is caused by _____.

- urgency and psychological manipulation
- trustworthy appearance
- lack of user vigilance

📝 4.2.7

Which of the following best describes how urgency is used in phishing attacks?

- To pressure users into acting without thinking
- To mimic legitimate branding
- To verify the user's credentials accurately
- To redirect users to the original website

📝 4.2.8

Phishing websites are often effective because users can easily spot the fake URL if they check carefully.

- True
- False

🕮 4.2.9

How to protect from fake websites

- Always verify the URL: Before entering sensitive information, take a moment to verify that the website URL is correct and belongs to a legitimate site. Small differences, such as an extra letter or a slightly modified domain (eg .net instead of .com) can signal a fake site.
- Look for security certificates: Trusted websites use SSL certificates, which create a secure, encrypted connection. You can identify them by the

"https://" at the beginning of the URL and often by the padlock icon in the browser's address bar.
- Be careful with emails and messages: Do not click on links in emails or messages from unknown senders, especially those asking for sensitive information. Phishing attempts often come via unsolicited emails or text messages asking you to "verify" or "update" your account information.
- Use antivirus software: Antivirus programs often have built-in protection against known phishing sites. They can help detect and block access to malicious sites before they pose a risk.
- Educate yourself about phishing: Knowing phishing techniques is one of your best defenses. Understanding common tactics like fake URLs or urgent requests will help you spot potential scams and stay safe.

4.2.10

One way to ensure a website is safe before entering personal information is to check for _____ in the URL, which indicates an encrypted connection.

- ftps://
- .net
- http://
- ftp://
- .com
- https://
- http:

4.2.11

Clicking on links in unsolicited emails or messages is generally safe, especially when they ask for personal information.

- False
- True

4.3 Phishing emails

4.3.1

Fake emails

Phishing is one of the most common and effective online scams. Attackers often send emails that appear to come from trusted sources, such as banks or popular websites, to trick recipients into clicking on malicious links. These links lead to fake websites designed to steal sensitive information such as usernames, passwords or credit card numbers.

How attackers create fake emails

- Impersonating the sender: Attackers often forge email headers, which are the parts of an email that contain information about the sender. In this way, they can give the impression that the email is from a legitimate organization such as your bank, online store or social media platform. This deception builds trust and increases the chances that the recipient will interact with the email.
- Use of email templates: Many phishing attacks use professional-looking email templates that mimic the style and format of genuine emails from trusted senders. These templates often contain the same logos, colors and layout as authentic communications, making it difficult for recipients to tell the difference at first glance.
- Email Address Manipulation: Attackers can use email addresses that are almost identical to those of legitimate senders, but contain slight variations. This may include replacing a letter with a similar-looking character (such as using a "0" instead of an "O") or adding additional characters. These subtle changes can mislead users into thinking that the email is from a trusted source.
- Inserting links to fake websites: Emails sent by attackers usually contain links that direct recipients to fake websites. These scam sites are designed to look as similar as possible to the legitimate sites they mimic, both visually and functionally. When a victim clicks on a link and enters their information, attackers capture that data for malicious purposes.
- Use of social engineering techniques: In order for the victim to take immediate action, phishing emails often use social engineering techniques. This may include urgent messages indicating that the recipient needs to update their account details, verify information or confirm payment. A sense of urgency is a psychological tactic designed to compel individuals to act quickly without fully considering the consequences of their actions.

It is important to exercise caution when receiving unsolicited email and verify the authenticity of any communication before taking action. Always double-check the sender's email address, look for signs of tampering, and watch out for unexpected requests for sensitive information.

4.3.2

Select two methods that attackers use to create fake emails:

- Using email templates that resemble genuine communications.
- Manipulating email addresses to look like legitimate ones.
- Sending emails with attachments that cannot be opened.
- Writing emails without any formatting or structure.

4.3.3

Which of the following statements best describes how attackers create fake emails?

- They impersonate trusted senders by forging email headers.
- They use generic email addresses that cannot be traced.
- They send emails with no links to avoid detection.
- They use random words in the subject line to grab attention.

4.3.4

Why phishing emails are effective

Phishing emails remain a prevalent threat in the digital world, and understanding their effectiveness can help students recognize and avoid them. Here are some reasons why these emails can fool even careful users:

- Trusted appearance - Fake emails are often designed to look almost identical to legitimate emails from trusted organizations. Attackers use professional layouts, logos, and fonts that mimic those of real companies, which can make it difficult for recipients to tell the difference. This trustworthy appearance creates a false sense of security and leads individuals to believe that they can trust the content of the email. As a result, they may not examine the details as carefully as they should.
- Urgency and psychological manipulation - Phishing emails often use urgency to manipulate recipients into taking quick action. Attackers may claim that the account has been compromised or that immediate verification is required to avoid consequences. This sense of urgency can create anxiety and prompt individuals to act without thinking, such as clicking on links or providing personal information. By exploiting human emotions, attackers increase their chances of success.
- Lack of user vigilance - Many users fail to verify the sender's email address or check the content of the email, making them vulnerable to phishing attacks. Some individuals may not recognize signs of a phishing attempt, such as poor grammar, suspicious links, or unusual requests for sensitive information. This lack of vigilance can lead to users inadvertently providing their information to fraudsters, making it easier for attackers to carry out their schemes.

4.3.5

What is one reason why phishing emails can be difficult to detect?

- They are often visually indistinguishable from genuine emails.
- They often come from unfamiliar senders.
- They are always sent during business hours.
- They contain multiple attachments.

4.3.6

Select factors that contribute to the effectiveness of phishing emails:

- They create a sense of urgency for immediate action.
- They are designed to look credible and professional.
- They are always written in formal language.
- They often lack any links to external websites.

4.3.7

Complete the following sentences:

Phishing emails can take advantage of a user's lack of _____, leading them to fall for scams.

Attackers often rely on a sense of _____, making recipients feel they need to act quickly.

A _____ appearance in a phishing email can mislead users into thinking it's legitimate.

- urgency
- vigilance
- credible

4.3.8

How to protect from phishing emails

Phishing attacks can take many forms, including emails, text messages and phone calls. Here are some important steps you can take to protect yourself from phishing emails:

- Always verify the sender - Before clicking on any link in an email, it is important to verify that the email is indeed from a trusted sender. Check the sender's email address carefully; look for typos, unusual domain names, or other suspicious signs that could indicate the email is fraudulent. If you are unsure about the legitimacy of an email, feel free to contact the sender directly using another method, such as calling them or visiting their official website. This extra step can help confirm whether the email is genuine or a phishing attempt.
- Do not click on links in suspicious emails - If the email raises any suspicion, do not click on the links it contains. Instead, manually type the website address into your browser to make sure you're visiting the right page. This procedure will help you bypass potentially dangerous links that could lead to phishing sites that aim to steal your personal information.
- Be wary of requests for sensitive information - Legitimate organizations will never ask you to provide sensitive information such as login information,

credit card numbers, or social security numbers via email. If you receive a request for such information, please ignore this email and contact the organization directly using the official communication channel. Trust your instincts; if something doesn't seem right, it's better to check it than to risk your personal information.

- Use antivirus software - Investing in antivirus software can be an effective defense against phishing emails. Good antivirus programs can identify and block malicious links and attachments before they can cause damage. Make sure your software is regularly updated to provide the best protection against emerging threats.
- Educate yourself about phishing - Staying informed about phishing techniques is one of the best ways to protect yourself. Familiarize yourself with terms like spoofing, spear phishing, whaling and smishing. Understanding these concepts will improve your ability to spot potential threats and avoid becoming a victim of fraud. The more you know, the better prepared you will be to defend against phishing attempts.

In addition to these strategies, it is important to remain vigilant and use common sense when dealing with emails. If something seems suspicious, take the time to investigate before risking your personal information. By following these precautions, you can significantly reduce the risk of becoming a victim of phishing attacks.

4.3.9

What should you do before clicking on any link in an email?

- Verify that the email is from a trusted sender.
- Forward the email to an admin.
- Reply to the email asking for clarification.
- Click the link to see where it goes.

4.3.10

Select actions you should take if you receive a suspicious email:

- Manually type the website address into your browser.
- Contact the sender through another method to verify the email.
- Click on the links to check their safety.
- Ignore the email and delete it without any further action.

4.3.11

Complete the following sentences with the correct word:

Antivirus software can help block phishing emails by identifying and blocking malicious _____ and attachments.

Trusted organizations will never ask you for sensitive _____ like credit card numbers through email.

It is essential to be aware of common _____ techniques to protect yourself from phishing attacks.

- links
- information
- scam

4.4 Smishing and phishing

🕮 4.4.1

Understanding Smishing: a form of SMS phishing

Smishing is a type of phishing attack that uses Short Message Service (SMS) messages to trick users into providing sensitive information. Attackers often impersonate trusted institutions such as banks, mobile operators or delivery services to gain the victim's trust. These deceptive messages may include links to fraudulent websites or directly ask users to share personal information, including login information, credit card numbers, social security numbers, or PINs.

How Smishing works

- Obtaining a phone number: Attackers can obtain phone numbers in a variety of ways. They could exploit personal data leaks, obtain numbers from online directories, or even generate random phone numbers. This ability to obtain a victim's phone number is a critical first step in executing a smishing attack.
- Creating fake messages: Once a phone number is obtained, attackers create SMS messages designed to appear trustworthy and authoritative. These messages often attempt to create a sense of urgency or fear in the recipient and encourage them to act quickly. For example, the message may claim that there is a problem with the recipient's bank account that requires immediate attention, prompting them to click on a link or provide information.
- Embedding links to fake websites: Smishing messages often contain links that lead to fake websites imitating legitimate sites. These scam sites are designed to capture the victim's login credentials or other sensitive information when they try to log in or update their information. The look and feel of these fake sites can be remarkably similar to the authentic ones, making it easy for users to be fooled.
- Direct request for data: In some cases, smishing messages may explicitly ask the victim to provide personal data. This can be presented as a need to verify an account, confirm a payment or update personal information. By creating a sense of necessity, attackers try to manipulate individuals into sharing sensitive data without taking the time to think critically about the request.

4.4.2

What is a primary goal of smishing attacks?

- To obtain sensitive information from users.
- To steal physical property.
- To promote legitimate services.
- To increase internet traffic.

4.4.3

Why Smishing is effective

- Mobile devices are always connected: Mobile devices keep users constantly connected to the Internet and individuals usually check their phones frequently. When an SMS message arrives, users often open it immediately without hesitation. This immediate attention makes them more likely to click on a link or respond to a request without fully considering the risks involved. For attackers, this behavior is advantageous because it plays on the user's instinct to respond quickly to messages.
- Limited display of information: On mobile devices, only part of the URL address is displayed in the SMS message. This restriction makes it difficult for users to judge whether a link is legitimate. For example, a link may appear to be from a well-known bank, but if the full URL is not visible, it may lead to a fraudulent site. This ambiguity allows attackers to create convincing messages that are difficult to verify at first glance.
- Trust in SMS messages: Many people have come to expect important notifications such as transaction alerts or updates from trusted institutions via SMS. This built-up trust can lead users to limit themselves when they receive messages that appear to be from reputable sources. As a result, users may not think critically about the content of these messages, making them more susceptible to manipulation.

4.4.4

Which of the following factors contribute to the effectiveness of smishing?

- Only part of the URL is displayed on mobile devices.
- Users often receive important messages via SMS from trusted institutions.
- Users rarely check their phones.
- SMS messages are never exploited by users.

📖 4.4.5

How to protect from Smishing

- Be careful with SMS messages from unknown senders: Be careful when receiving SMS messages from unknown sources. Do not click on any links or enter personal information in these messages. Instead, take a moment to assess the situation. If you don't know the sender, it's best to ignore or delete the message rather than risking your personal information.
- Always verify the sender: If you receive a message that raises doubts about its authenticity, verify the identity of the sender. Contact the institution listed in the message using its official phone number or website - not the contact details provided in the SMS. This extra step can help confirm whether the message is genuine or a phishing attempt.
- Don't click on links in suspicious messages: If a message looks suspicious, don't click on any links in it. Instead, manually enter the address of the institution's website into the browser. This procedure ensures that you are going to a legitimate site and not a fraudulent site that wants to steal your information.
- Be careful when entering sensitive data: Be aware that trusted organizations will never ask for sensitive data such as passwords or financial information via SMS. If you receive such a request, it is a clear sign that the message may be a phishing attempt. Always ignore such requests and report them if possible.
- Use security software: Installing security software on your mobile device can provide an additional layer of protection. Such software can help block phishing SMS messages, detect malicious websites and alert you to potential threats, allowing you to navigate the digital world more safely.
- Educate yourself about smishing: Knowledge is a powerful tool in the fight against smishing. Take the time to familiarize yourself with common smishing techniques and signs of a phishing attempt. The more you understand about how these attacks work, the better prepared you will be to detect and avoid them.

📝 4.4.6

Which of the following actions are recommended when you receive an SMS message from an unknown sender?

- Verify the sender by contacting them through official channels.
- Ignore the message and delete it without responding.
- Click on any links provided in the message.
- Provide personal information to the sender if asked.

📖 4.4.7

Overview of Vishing

Vishing, short for voice phishing, is a form of cyberattack in which an attacker uses telephone communications to obtain sensitive data from a victim. Attackers often pretend to be trustworthy people, such as employees of banks, insurance companies, technical support services or government institutions. The primary goal of vishing is to get the victim to voluntarily provide sensitive information, including login information, credit card numbers, social security numbers, PIN codes, or other private information.

Vishing methods:

- Caller ID spoofing: This technique involves the attacker using technologies to mask their phone number so it looks like they are calling from an official number associated with a bank or other trusted institution. This can mislead the victim into believing they are talking to a legitimate agent, making it more likely that they will share personal information.
- Urgency and fear-mongering: Vishing calls often contain false warnings about security threats, suspicious account activity, or payment problems to induce panic in the victim. This sense of urgency can lead individuals to act rashly without properly evaluating the situation, making them more vulnerable to manipulation.
- Spoofing: This broader technique refers to the general practice of disguising communications from an unknown source as if they were from a trusted source. In the context of vishing, this means that attackers can make their calls appear more legitimate by impersonating well-known organizations or individuals.
- Using social engineering: Attackers often use social engineering tactics to build a relationship with the victim. They may flatter victims, act friendly, or pretend to help them, creating a false sense of security that encourages the victim to divulge sensitive information.
- Collecting information from public sources: Attackers can obtain personal information about a victim from publicly available sources, such as social networks. By tailoring their approach with specific details, attackers can increase their credibility and make their attacks more convincing.

📝 4.4.8

What is one common tactic used in vishing to persuade victims to provide sensitive information?

- Requesting immediate action due to a security threat.
- Providing a refund for a non-existent purchase.
- Offering a free vacation.
- Sending a promotional code.

📖 4.4.9

Why Vishing is effective

- Telephone communication feels more trustworthy: Many individuals perceive telephone conversations as more personal and authentic compared to email or text messages. This perception may lead them to more easily trust information transmitted over the phone, making them more susceptible to manipulation by the caller. The human voice can convey emotion and urgency in a way that written communication cannot, increasing the likelihood that victims will comply with requests for sensitive information.
- Emotional pressure: Vishing calls often use emotional tactics to manipulate victims. Attackers can create a sense of fear, panic, or urgency, causing the victim to react quickly without fully considering the situation. This emotional pressure can cloud judgment and make it difficult for the victim to think critically and assess the legitimacy of the call. As a result, they may reveal personal information that they would normally protect.
- Lack of vigilance: Many individuals are not sufficiently aware of the risks associated with vishing attacks. This lack of awareness can lead to complacency, causing them to miss red flags during phone conversations. Some may not have been educated on the signs of a vishing attempt, making them more vulnerable to falling victim to these types of scams.

📝 4.4.10

Which of the following factors contribute to the effectiveness of vishing?

- The emotional pressure placed on victims to act quickly.
- The perceived trustworthiness of telephone communication.
- The use of well-designed phishing scenarios.
- The fact that all phone calls are recorded.

📖 4.4.11

How to protect against Vishing

- Beware of calls from unknown numbers: If a number you do not recognize calls you and the caller asks for personal information, proceed with caution. Unknown numbers may belong to scammers trying to trick you into providing sensitive information. It is wise to let such calls go to voicemail or ask for more information about the caller before connecting.
- Never share sensitive information over the phone: Reputable institutions will never ask you to share sensitive information, such as social security numbers, passwords, or bank account details, over the phone. If the caller requests such information, it is probably a vishing attempt. Always prioritize your privacy and security by keeping your personal information private.
- Verify the identity of the caller: If you are not sure about the legitimacy of the call, do not hesitate to verify the identity of the caller. Hang up and contact

the institution directly using its official phone number or website. This extra step can help confirm whether the call was genuine or part of a phishing scheme.

- Hang up and call back: If you suspect a call is in progress, it's best to hang up immediately. Then use the institution's official contact number and call back. This approach ensures that you are communicating with a legitimate agent, eliminating the risk of providing information to a potential fraudster.
- Use call blocking: Use the call blocking features available on your phone. Most smartphones allow you to block specific numbers, which can be useful if you notice repeated calls from known numbers when you hang up. This proactive measure can help reduce the number of unwanted calls received.
- Educate yourself about vishing: Staying informed about vishing techniques and tactics is one of the best defenses against these types of scams. Learn about common methods used by attackers and stay up-to-date on the latest cybersecurity trends. The more knowledge you have, the better you will be able to recognize and avoid vishing attempts.

4.4.12

f you receive a call from an unknown number asking for personal _____, it is important to verify the _____ of the caller before providing any information. If you suspect it’s a scam, hang up and call back using the official _____.

- information
- number
- identity

AI in Phishing Protection

Chapter 5

5.1 Role of AI

📖 5.1.1

The role of AI in phishing

AI plays a key role in the cyber threat landscape, especially in the creation and distribution of phishing messages. One of the main uses of AI in this context is AI-powered language models that can be (not) used to create highly persuasive messages. These messages are often tailored to manipulate the recipient's emotions and trust, prompting them to take risky actions such as clicking on links that lead to fraudulent websites or providing confidential information. The sophistication of AI-generated content makes it increasingly difficult for individuals to distinguish between legitimate communications and malicious attempts to extract sensitive data, increasing the overall threat level.

In addition to creating fake news, AI technology can also be used to create realistic copies of legitimate websites. This capability allows attackers to design phishing sites that closely resemble trusted entities such as banks, social media platforms, or e-commerce sites. By mimicking the appearance and functionality of these legitimate websites, attackers can effectively trick victims into believing they are interacting with a trusted source. This tactic not only increases the likelihood that victims will enter their sensitive information, but also complicates efforts to identify and mitigate phishing attacks. As a result, using AI in this way greatly increases the effectiveness of phishing campaigns and poses a significant risk to users who may not be aware of the tactics being used against them.

The role of AI in phishing is not limited to facilitating attacks; it also offers potential defense options. The same artificial intelligence technologies that enable attackers to launch sophisticated phishing attempts can be repurposed to develop tools to detect and prevent such threats. For example, machine learning algorithms can analyze communication patterns and identify characteristics common to phishing attempts. These algorithms can be trained to recognize signs of suspicious activity, such as unusual sender behavior or the presence of known malicious links. By implementing these advanced detection methods, organizations and individuals can improve their security posture and reduce the likelihood of falling victim to phishing scams.

Phishing methods are constantly evolving, with attackers constantly refining their strategies to bypass detection measures. For example, they may use social engineering techniques that use psychological factors such as urgency or fear to force victims to make rash decisions. In response, cybersecurity professionals must remain vigilant and adaptable, using AI-based solutions that can keep up with emerging threats. Ongoing battle between attackers and defenders underscores the importance of ongoing education and awareness of phishing tactics, ensuring that users are equipped with the knowledge to effectively recognize and respond to potential threats.

In short, AI has a dual role in phishing – acting as a tool for attackers to create more convincing messages and fake websites, and as a resource for defenders to identify and combat these threats. The growing sophistication of AI technology poses significant challenges to cyber security, requiring a proactive approach to awareness, detection and prevention.

5.1.2

https://towardsdatascience.com/phishing-classification-with-an-ensemble-model-d4b15919c2d7

5.1.3

What are two primary tasks AI can perform in relation to phishing?

- Generating compelling phishing messages
- Creating fraudulent copies of legitimate webpages
- Sending spam emails
- Protecting users from all cyber threats
- Providing financial advice

5.1.4

How do AI-generated phishing messages typically deceive recipients?

- By manipulating emotions and trust
- By mimicking legitimate communication styles
- By prompting immediate action or urgency
- By appearing as marketing emails
- By using official government email addresses

5.1.5

Why is the ability to replicate legitimate webpages significant in phishing attacks?

- It increases the likelihood of victims entering sensitive information
- It makes the phishing attack seem more credible
- It helps to bypass traditional security measures
- It allows attackers to track user activity
- It provides attackers with a direct communication line to victims

5.1.6

AI as a defense against phishing

AI is increasingly recognized as a powerful tool in the fight against phishing attacks. While AI-powered language models can be used by cybercriminals to create

convincing phishing messages, the same technologies can be used to strengthen defenses against such threats. One of the most promising applications of AI in this area is the detection of phishing messages. This application is particularly important because it aligns with previous discussions about the role of AI in spam detection. However, it is essential to understand that phishing detection requires a more nuanced approach than traditional spam detection, focusing heavily on the content of the message itself.

When detecting spam, metadata such as sender information, delivery timestamps, and message routing can help identify unwanted communications. Unfortunately, when it comes to phishing attempts, this metadata is often insufficient. Phishing messages are typically designed to closely mimic legitimate communications, making it difficult to distinguish between genuine and fraudulent messages based solely on metadata. As a result, AI-based phishing detection systems must analyze the actual content of messages to identify potentially malicious links or requests for sensitive information.

One effective strategy for detecting phishing messages is to identify the characteristic features that are commonly found in such attacks. For example, phishing emails often contain clickable links that direct users to fraudulent websites designed to obtain sensitive information. Using AI algorithms, it is possible to scan these links in messages and assess their legitimacy. In addition, advanced AI techniques can evaluate the surrounding context and language used in the message to determine if it matches patterns typically associated with phishing attempts. This comprehensive approach increases the accuracy of phishing detection systems and allows them to flag suspicious messages more effectively.

Spear phishing, a targeted form of phishing aimed at specific individuals or organizations, presents a unique challenge. Detection methods effective for general phishing may not be sufficient for spear phishing, as these messages often contain customized information that can make them appear credible. Artificial intelligence can play a key role in developing adaptive detection mechanisms that learn from new phishing techniques and patterns and continuously improve their ability to identify even the most sophisticated attacks. Machine learning models can analyze huge amounts of data, adjusting their algorithms based on the latest phishing trends, keeping them one step ahead of cybercriminals.

Although AI technology can significantly reduce the risk of falling victim to phishing attacks, users must remain vigilant. Organizations can use AI tools not only to detect, but also to train employees to recognize phishing attempts. By incorporating AI-based insights into training programs, users can become more adept at detecting suspicious communications and understanding attacker tactics.

5.1.7

What characteristics of phishing messages can AI detection systems analyze?

- Presence of clickable links
- The emotional tone of the message
- The sender's email address
- Requests for sensitive personal information
- Frequency of message delivery

5.1.8

In what ways can AI enhance phishing detection mechanisms?

- By recognizing patterns in previous phishing attempts
- By analyzing the content of messages in real time
- By learning from user feedback on detected phishing messages
- By relying only on metadata for detection
- By generating automated responses to phishing attempts

5.1.9

Which of the following are true about spear phishing?

- It is targeted at specific individuals or organizations.
- It often includes personalized information to increase credibility.
- AI can assist in identifying patterns unique to spear phishing attacks.
- It typically uses generic messages to reach a broad audience.
- It is less harmful than regular phishing attacks.

5.2 AI models

5.2.1

Artificial intelligence and machine learning in phishing detection

Artificial intelligence and machine learning are key tools in identifying phishing websites. Traditional methods such as blacklists and heuristics rely on lists of known phishing websites or rule-based systems, but may miss newly created or slightly modified phishing sites. However, AI and ML are adaptive and can learn to detect new phishing patterns by analyzing website features and behavior. This makes them more effective and reliable at detecting phishing sites compared to older methods, as they are constantly improving with more data. Their ability to analyze complex patterns in data gives them an edge in predicting and preventing phishing attacks before they are exposed.

Phishing detection

- Identifying suspicious URLs: Machine learning algorithms are trained to recognize suspicious characteristics in URLs that are often associated with phishing attempts. For example, they may notice typos in domain names, the use of an IP address instead of a regular domain name, or the presence of unusual characters such as the "@" symbol that may mislead users. Shortened URLs can also be a warning as they can hide the actual destination of the link. By identifying these URL features, ML algorithms help filter out potentially dangerous links before users click on them.
- Website Content Analysis: AI and ML algorithms can go beyond a URL to examine website content and detect phishing attempts. They analyze site elements commonly found on phishing sites, such as fake login forms, prompts for sensitive information, or redirects to other suspicious sites. For example, phishing websites often mimic the appearance of legitimate sites, but have minor inconsistencies that AI can detect. These algorithms can identify differences in content structure, language or images that are likely signs of a phishing site. This in-depth content analysis helps ensure that even well-disguised phishing sites are flagged.
- Anomaly detection: Websites that harvest data without authorization often display unusual patterns of behavior that can be detected using anomaly detection algorithms. For example, these websites may load certain elements differently or have abnormal interaction sequences. AI can learn what typical, secure websites look like, and then identify behavior that doesn't fit those patterns and flag them as suspicious. Anomaly detection is especially useful when detecting new phishing methods that do not yet exist in databases.
- Predictive Modeling: By analyzing historical data from previous phishing attacks, AI and ML algorithms can create predictive models. These models are designed to recognize patterns and features typical of phishing sites and predict with high accuracy whether a new site is likely to represent a phishing threat. For example, predictive modeling can estimate risk levels based on how similar a new site is to known phishing sites.

5.2.2

Which of the following are characteristics that machine learning algorithms may look for in a URL to identify it as potentially phishing?

- Typos in the domain name
- Use of an IP address instead of a domain name
- Use of unusual symbols like "@" in the URL
- Presence of a long, complex URL path
- Presence of HTTPS encryption

5.2.3

What types of website content can AI and ML algorithms analyze to help detect phishing attempts?

- Presence of fake login forms
- Requests for sensitive information
- Redirection to other suspicious pages
- Large images or videos embedded on the page
- Use of a search bar on the page

5.2.4

Which of the following are indicators that anomaly detection algorithms might flag as suspicious on a phishing website?

- Which of the following are indicators that anomaly detection algorithms might flag as suspicious on a phishing website?
- Unusual interaction sequences on the website
- Use of strong, complex passwords for login
- Requests for personal information from users
- Consistent layout with secure websites

5.2.5

Examples of AI and ML algorithms used in phishing detection

- Decision trees: These algorithms work by creating a series of decision rules that divide data into categories, such as phishing or legitimate sites. A decision tree is easy to understand because it visually breaks down the steps taken to achieve a classification. This clarity makes it a popular choice for interpreting how different features contribute to phishing identification. Decision trees can capture complex relationships, but may require fine-tuning to avoid misclassification. They are particularly useful in educational contexts because students can follow the decision-making process step by step.
- Random Forests: Random forests are an extension of decision trees that combine the results of many trees to produce a final prediction. This "voting" process increases the overall accuracy and reliability of the model. Random forests can handle a large number of input variables and are less error-prone than individual decision trees. However, they can be slower due to the number of trees involved. Random forests are widely used because of their balance between accuracy and interpretability.
- Multilayer Perceptrons: This is a type of artificial neural network that can classify data by learning complex patterns in multiple layers. Each layer transforms the input data, allowing the network to capture deeper relationships between variables. Multilayer perceptrons are ideal for cases where there are many input features because they can handle and learn from

this complexity. However, they require more computing power and can be more difficult to interpret. They are suitable for phishing detection where there are many subtle factors to consider.

- XGBoost: Known for its high accuracy and fast performance, XGBoost is a popular choice in machine learning competitions and real-world applications. This algorithm uses gradient boosting, which combines multiple weak models to produce a strong predictor. XGBoost is efficient in processing noisy data and can achieve high performance with minimal modifications. It can detect phishing sites by focusing on small details that may be missed by other algorithms. Due to its complexity, XGBoost is often used in large applications where accuracy is critical.
- Support Vector Machines (SVM): Support Vector Machines are powerful classifiers that work well with high-dimensional data, making them suitable for analyzing multiple characteristics of web pages. SVMs create a boundary that best separates phishing from legitimate sites based on their features. This threshold helps ensure that even small differences in data are taken into account when classifying a site. Although SVMs can be computationally intensive, they are highly accurate when properly configured. SVMs are particularly effective in applications where both accuracy and robustness are required.
- K-Nearest Neighbors (KNN): KNN is a simple algorithm that classifies web pages based on their similarity to the "K" nearest examples in the training data set. It's easy to understand and implement, but KNN can be slow if the data set is large because it compares each new data point to all existing points. For phishing detection, KNN works well if there are clear clusters of phishing compared to legitimate sites. This simplicity makes KNN a good choice for initial survey or educational purposes, although it may not always be the most accurate option.
- Artificial Neural Networks (ANN): Inspired by the human brain, artificial neural networks can learn from complex data patterns and relationships. They are efficient at working with unstructured data such as images and text, which is useful for analyzing web page layouts and content. ANNs can adapt to various phishing detection tasks, but they require significant computing resources and large amounts of training data. Their flexibility makes them valuable in phishing detection, although they can be more difficult to interpret. ANNs are often used in advanced applications where the detection of subtle patterns is critical.

5.2.6

Which machine learning algorithms are best suited for analyzing large datasets with complex, non-linear relationships, such as those often found in phishing detection?

- Random Forests
- Multilayer Perceptrons
- Support Vector Machines
- Decision Trees
- K-Nearest Neighbors

5.2.7

When aiming for a balance between model accuracy and interpretability in phishing detection, which algorithms would be appropriate choices?

- Decision Trees
- Random Forests
- XGBoost
- Artificial Neural Networks
- Support Vector Machines

5.2.8

Which algorithms would be most suitable for detecting phishing by identifying subtle patterns within high-dimensional data?

- Multilayer Perceptrons
- Support Vector Machines
- Artificial Neural Networks
- K-Nearest Neighbors
- Decision Trees

5.2.9

Which algorithms are generally preferred for their simplicity and ease of implementation in educational contexts or initial surveys for phishing detection?

- Decision Trees
- K-Nearest Neighbors
- Random Forests
- Support Vector Machines
- Artificial Neural Networks

5.2.10

If the goal is to detect phishing websites with high speed and minimal computational resources, which algorithms are best suited for this requirement?

- Decision Trees
- XGBoost
- K-Nearest Neighbors
- Support Vector Machines
- Random Forests

📖 5.2.11

Choosing the best AI model for phishing detection

Choosing the best AI model for phishing detection depends on specific priorities such as balance accuracy and recall. For example, if the goal is to minimize false alarms and focus on accuracy, a high-accuracy model such as XGBoost may be more appropriate. On the other hand, if it is essential to catch as many phishing sites as possible, even if it means some false positives, a model with a high recovery rate may be more appropriate. Some algorithms are also more interpretable than others, which can be important for understanding decision-making processes. Ultimately, the choice of model should be consistent with the goals and constraints of a given phishing detection task.

5.3 AI projects

📝 5.3.1

Project: Phishing email detection

(by https://www.kaggle.com/code/kirollosashraf/phishing-email-detection-using-deep-learning/notebook)

Compare different algorithms to identify phishing emails.

Dataset:

- original: https://www.kaggle.com/datasets/subhajournal/phishingemails
- reduced: https://priscilla.fitped.eu/data/cybersecurity/phishing/phishing_email_reduced.csv

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import plotly.express as px
from sklearn.feature_extraction.text import
TfidfVectorizer,CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.text import Tokenizer
```

```
from tensorflow.keras.layers import
Embedding,GRU,LSTM,Bidirectional,SimpleRNN
from tensorflow.keras.utils import pad_sequences
from sklearn.preprocessing import LabelEncoder
from keras.models import Sequential
from keras.layers import Dense,Dropout
import tensorflow as tf
import warnings

warnings.filterwarnings('ignore')
```

1. Data understanding

```
df =
pd.read_csv("https://priscilla.fitped.eu/data/cybersecurity/ph
ishing/phishing_email_reduced.csv", delimiter=",")
# be patient
print(df.head())
```

Program output:

```
   id                                          Email Text
Email Type
0   0  re : 6 . 1100 , disc : uniformitarianism , re ...
Safe Email
1   1  the other side of * galicismos * * galicismo *...
Safe Email
2   2  re : equistar deal tickets are you still avail...
Safe Email
3   3  \nHello I am your hot lil horny toy.\n    I am...
Phishing Email
4   4  software at incredibly low prices ( 86 % lower...
Phishing Email
```

Drop duplicates and null values

```
df.dropna(inplace=True,axis=0)
df.drop_duplicates(inplace=True)

print("Dimension of the row data:",df.shape)
```

Program output:

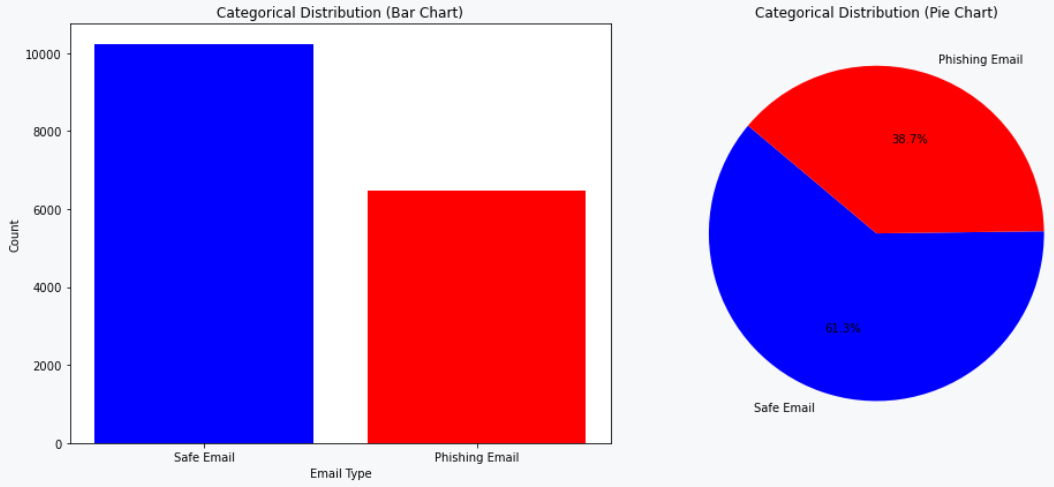
```
Dimension of the row data: (16705, 3)
```

Dataset visualisation

```
import matplotlib.pyplot as plt
```

```
# Get value counts for the 'Email Type' column
email_counts = df['Email Type'].value_counts()

# Define colors for each bar (adjust this list based on the
number of categories)
colors = ['blue', 'red'][:len(email_counts)]

# Create a figure with two subplots: one for the bar chart,
one for the pie chart
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6))

# Bar chart
ax1.bar(email_counts.index, email_counts.values, color=colors)
ax1.set_title("Categorical Distribution (Bar Chart)")
ax1.set_xlabel("Email Type")
ax1.set_ylabel("Count")

# Pie chart
ax2.pie(email_counts, labels=email_counts.index,
colors=colors, autopct='%1.1f%%', startangle=140)
ax2.set_title("Categorical Distribution (Pie Chart)")

# Adjust layout and display
plt.tight_layout()
plt.show()
```

Program output:



2. Data preprocessing

- Integer Encoding

```
le = LabelEncoder()
df["Email Type"] = le.fit_transform(df["Email Type"])
print(df)
```

Program output:

```
          id                                               Email
Text  Email Type
0          0  re : 6 . 1100 , disc : uniformitarianism , re
...           1
1          1  the other side of * galicismos * * galicismo
*...           1
2          2  re : equistar deal tickets are you still
avail...           1
3          3  \nHello I am your hot lil horny toy.\n    I
am...            0
4          4  software at incredibly low prices ( 86 %
lower...           0
...      ...
...         ...
16700  16703  \nRick Moen  a ĂÂ©crit:> > I'm confused. I
th...            1
16701  16704  date a lonely housewife always wanted to date
...           0
16702  16705  request submitted : access request for anita
....           1
16703  16706  re : important - prc mtg hi dorn & john , as
y...           1
16704  16707  press clippings - letter on californian
utilit...           1

[16705 rows x 3 columns]
```

Remove hyperlinks, punctuations, extra space

```
import re

def preprocess_text(text):
    # Remove hyperlinks
    text = re.sub(r'http\S+', '', text)
    # Remove punctuations
    text = re.sub(r'[^\w\s]', '', text)
```

```
    # Convert to lowercase
    text = text.lower()
    # Remove extra spaces
    text = re.sub(r'\s+', ' ', text).strip()
    return text

# Apply the preprocess_text function to the specified column
in the DataFrame
df["Email Text"] =df["Email Text"].apply(preprocess_text)
print(df.head())
```

Program output:

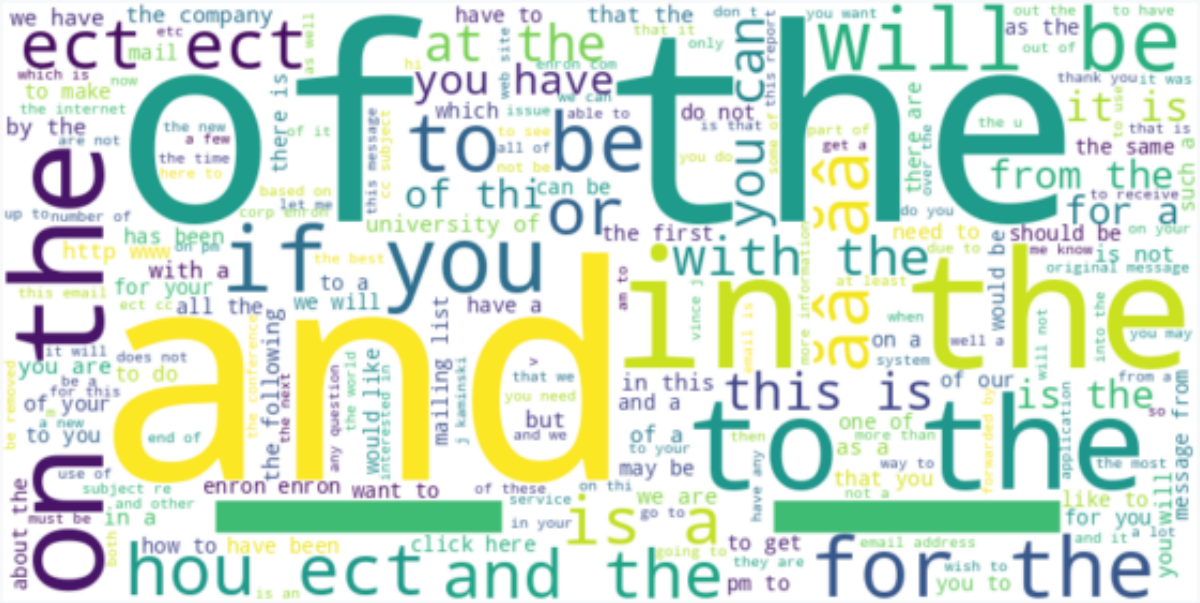
```
   id                                          Email Text
Email Type
0   0  re 6 1100 disc uniformitarianism re 1086 sex l...
1
1   1  the other side of galicismos galicismo is a sp...
1
2   2  re equistar deal tickets are you still availab...
1
3   3  hello i am your hot lil horny toy i am the one...
0
4   4  software at incredibly low prices 86 lower dra...
0
```

WordCloud

- of avaiable stopwords

```
from wordcloud import WordCloud

#combine all rows into a single string
all_mails = " ".join(df['Email Text'])

#create a wordcloud object
word_cloud =
WordCloud(stopwords="english",width=800,height=400,background_
color='white').generate(all_mails)

plt.figure(figsize=(10,6))
plt.imshow(word_cloud,interpolation='bilinear')
plt.axis("off")
plt.show()
```

Program output:



- wordcloude of unique words

```
#combine all rows into a single string
all_mails = " ".join(df['Email Text'])

#create a wordcloud object
word_cloud =
WordCloud(width=800,height=400,background_color='white',max_wo
rds=10000).generate(all_mails)
plt.figure(figsize=(10,6))
plt.imshow(word_cloud,interpolation='bilinear')
plt.axis("off")
plt.show()
```

Program output:

Converting text into vector

- Tfidf vectorizer

```
from sklearn.feature_extraction.text import
TfidfVectorizer,CountVectorizer

tf = TfidfVectorizer(stop_words="english",max_features=10000)
#dimension reduction

feature_x = tf.fit_transform(df["Email Text"]).toarray()
y_tf = np.array(df['Email Type']) # convert the label into
numpy array
```

Splitting into train and test

```
x_train,x_test,y_train,y_test =
train_test_split(feature_x,y_tf,train_size=0.8,random_state=0)
```

3. Applying different algorithm

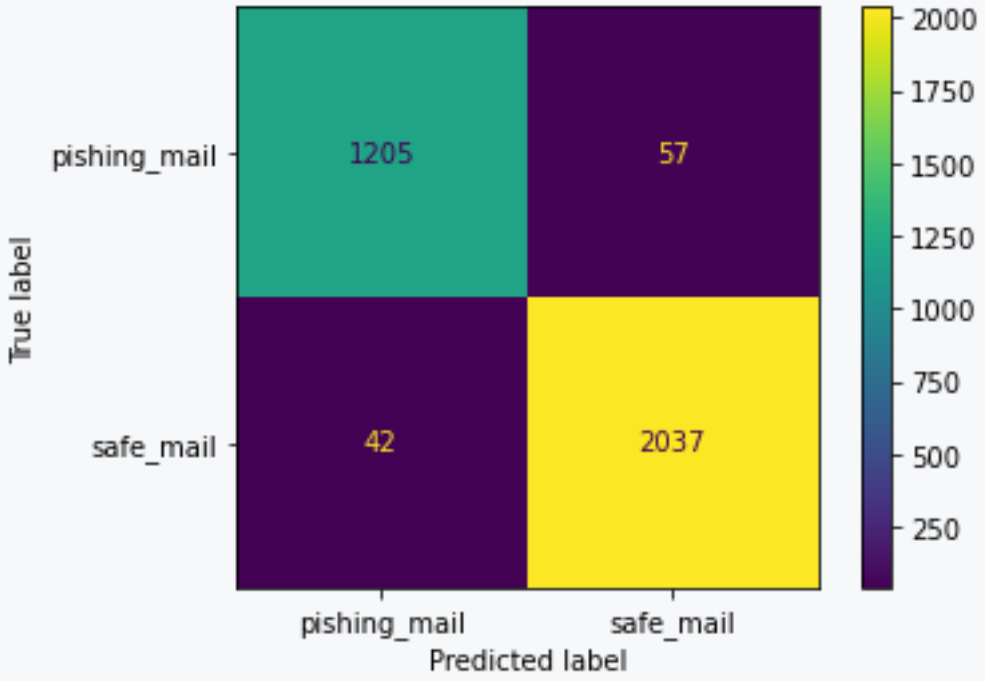
a. Naive Bayes

```
#naive bayes works with condtional probability
from sklearn.naive_bayes import MultinomialNB
nb = MultinomialNB()
nb.fit(x_train,y_train)
```

```
from sklearn.metrics import
accuracy_score,f1_score,classification_report,ConfusionMatrixD
isplay,confusion_matrix
pred_nav = nb.predict(x_test)

# Checking the performance
print(f"accuracy from native bayes:
{accuracy_score(y_test,pred_nav)*100:.2f} %")
print(f"f1 score from naive bayes:
{f1_score(y_test,pred_nav)*100:.2f} %")
print("classification report
:\n\n",classification_report(y_test,pred_nav))

#confusion matrix
clf_nav = confusion_matrix(y_test,pred_nav)
cx_ =
ConfusionMatrixDisplay(clf_nav,display_labels=['pishing_mail',
'safe_mail']).plot()
plt.show()
```

Program output:

```
accuracy from native bayes: 97.04 %
f1 score from naive bayes: 97.63 %
classification report :

              precision    recall  f1-score   support

           0       0.97      0.95      0.96      1262
           1       0.97      0.98      0.98      2079

    accuracy                           0.97      3341
   macro avg       0.97      0.97      0.97      3341
weighted avg       0.97      0.97      0.97      3341
```



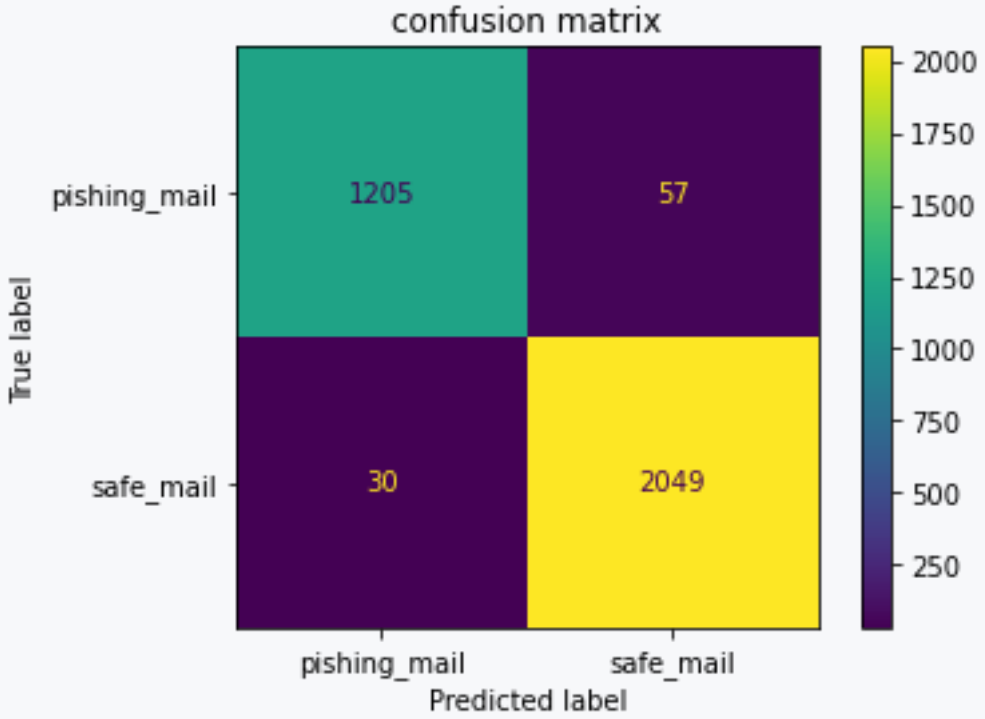
b. Logistic Regression

```
lg = LogisticRegression()
lg.fit(x_train,y_train)

# prediction
pred_lg = lg.predict(x_test)
# performance
print("")
print(f"accuracy from logistic
regression:{accuracy_score(y_test,pred_lg)*100:.2f} %")
print(f"f1 score from logistic regression:
{f1_score(y_test,pred_lg)*100:.2f} %")
print("classification report :
\n",classification_report(y_test,pred_lg))

clf_lg = confusion_matrix(y_test,pred_lg)
cx_ =
ConfusionMatrixDisplay(clf_lg,display_labels=['pishing_mail','
safe_mail']).plot()
plt.title("confusion matrix")
plt.show()
```

Program output:

```
accuracy from logistic regression:97.40 %
f1 score from logistic regression: 97.92 %
classification report :
```

```
              precision    recall  f1-score   support

           0       0.98      0.95      0.97      1262
           1       0.97      0.99      0.98      2079

    accuracy                           0.97      3341
   macro avg       0.97      0.97      0.97      3341
weighted avg       0.97      0.97      0.97      3341
```



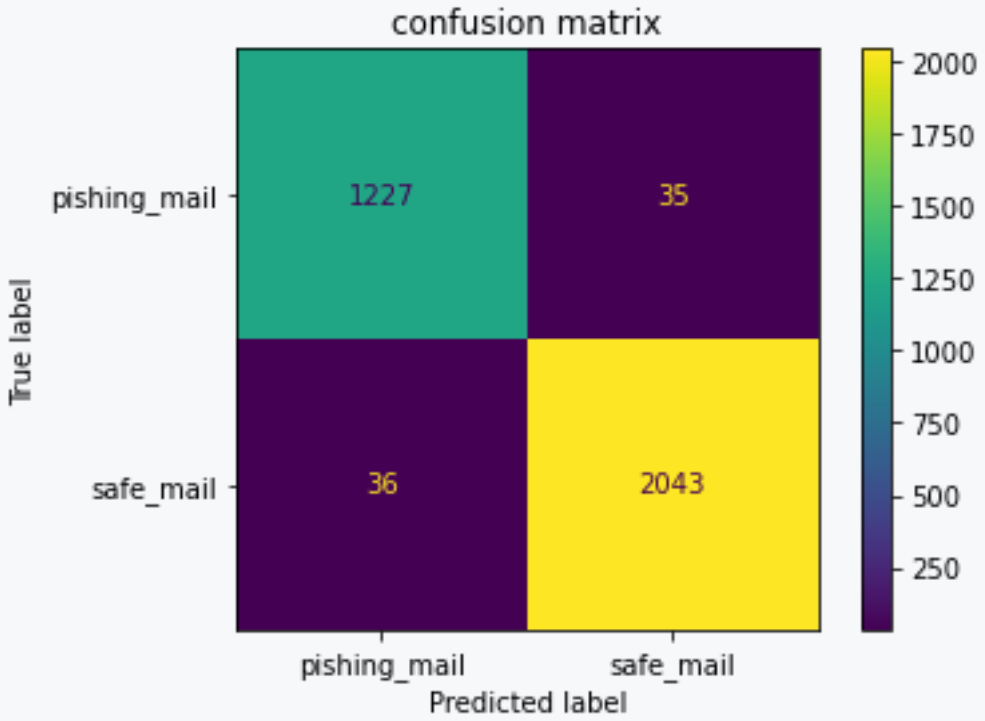
c. SGD Classifier

```
from sklearn.linear_model import SGDClassifier

# passing object
sgd = SGDClassifier()
sgd.fit(x_train,y_train)

# prediction
pred_sgd = sgd.predict(x_test)
# performance
print(f"accuracy from logistic
regression:{accuracy_score(y_test,pred_sgd)*100:.2f} %")
print(f"f1 score from logistic regression:
{f1_score(y_test,pred_sgd)*100:.2f} %")
print("classification report :
\n",classification_report(y_test,pred_sgd))
```

```
clf_sgd = confusion_matrix(y_test,pred_sgd)
cx_ =
ConfusionMatrixDisplay(clf_sgd,display_labels=['pishing_mail',
'safe_mail']).plot()
plt.title("confusion matrix")
plt.show()
```

Program output:

```
accuracy from logistic regression:97.87 %
f1 score from logistic regression: 98.29 %
classification report :
              precision    recall  f1-score   support

           0       0.97      0.97      0.97      1262
           1       0.98      0.98      0.98      2079

    accuracy                           0.98      3341
   macro avg       0.98      0.98      0.98      3341
weighted avg       0.98      0.98      0.98      3341
```



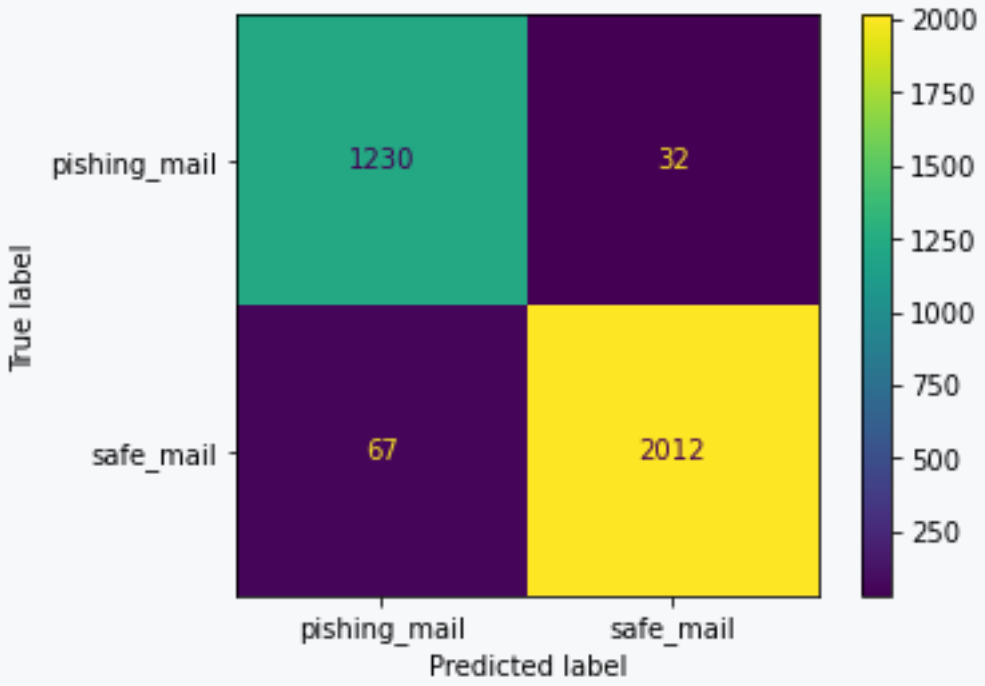
d. XGBoost

```
# applying boosting algorithm
from xgboost import XGBClassifier
xgb = XGBClassifier()
```

```
xgb.fit(x_train,y_train)

#prediction
pred_xgb = xgb.predict(x_test)

#performance
print(f"accuracy from
XGB:{accuracy_score(y_test,pred_xgb)*100:.2f} %")
print(f"f1 score from XGB: {f1_score(y_test,pred_xgb)*100:.2f}
%")
print("classification report :
\n",classification_report(y_test,pred_xgb))

#confusion matrix
clf_xgb = confusion_matrix(y_test,pred_xgb)
cx_ =
ConfusionMatrixDisplay(clf_xgb,display_labels=['pishing_mail',
'safe_mail']).plot()
plt.show()
```

Program output:

```
accuracy from XGB:97.04 %
f1 score from XGB: 97.60 %
classification report :
              precision    recall  f1-score   support

           0       0.95      0.97      0.96      1262
           1       0.98      0.97      0.98      2079

    accuracy                           0.97      3341
   macro avg       0.97      0.97      0.97      3341
weighted avg       0.97      0.97      0.97      3341
```



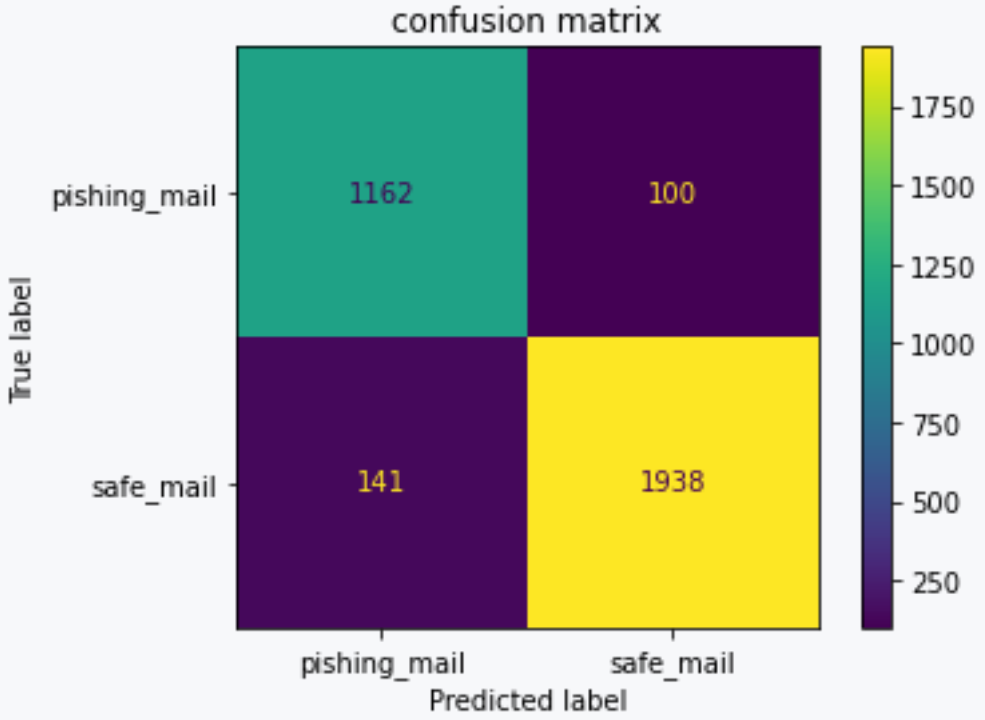
e. Decision tree

```
dtr = DecisionTreeClassifier() #passing object
dtr.fit(x_train,y_train)

#prediction
pred_dtr = dtr.predict(x_test)

#performance
print(f"accuracy from Decision
Tree:{accuracy_score(y_test,pred_dtr)*100:.2f} %")
print(f"f1 score from Decision Tree:
{f1_score(y_test,pred_dtr)*100:.2f} %")
print("classification report :
\n",classification_report(y_test,pred_dtr))

#confusion matrix
clf_dtr = confusion_matrix(y_test,pred_dtr)
cx_ =
ConfusionMatrixDisplay(clf_dtr,display_labels=['pishing_mail',
'safe_mail']).plot()
plt.title("confusion matrix")
plt.show()
```

Program output:

```
accuracy from Decision Tree:92.79 %
f1 score from Decision Tree: 94.15 %
```

```
classification report :
               precision    recall  f1-score   support

           0       0.89      0.92      0.91      1262
           1       0.95      0.93      0.94      2079

    accuracy                           0.93      3341
   macro avg       0.92      0.93      0.92      3341
weighted avg       0.93      0.93      0.93      3341
```



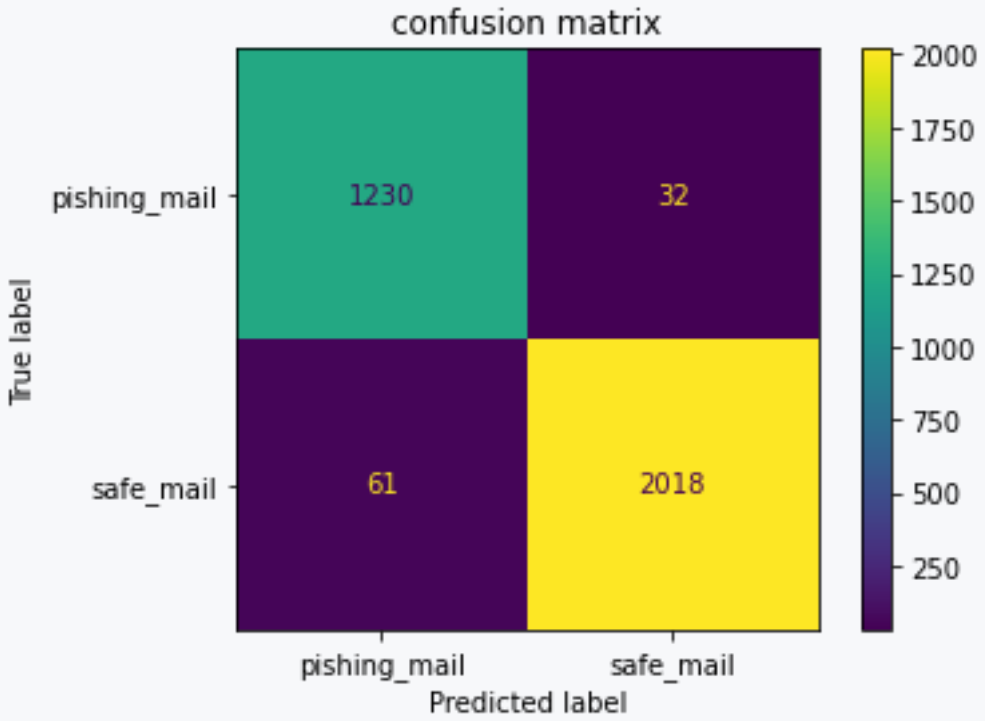
f. Random forest

```
rnf = RandomForestClassifier() #passing object
rnf.fit(x_train,y_train)

#prediction
pred_rnf = rnf.predict(x_test)

#performance
print(f"accuracy from rnadom
forest:{accuracy_score(y_test,pred_rnf)*100:.2f} %")
print(f"f1 score from random forest:
{f1_score(y_test,pred_rnf)*100:.2f} %")
print("classification report :
\n",classification_report(y_test,pred_rnf))
```

```
#confusion matrix
clf_rnf = confusion_matrix(y_test,pred_rnf)
cx_ =
ConfusionMatrixDisplay(clf_rnf,display_labels=['pishing_mail',
'safe_mail']).plot()
plt.title("confusion matrix")
plt.show()
```

Program output:

```
accuracy from rnadom forest:97.22 %
f1 score from random forest: 97.75 %
classification report :
              precision    recall  f1-score   support

           0       0.95      0.97      0.96      1262
           1       0.98      0.97      0.98      2079

    accuracy                           0.97      3341
   macro avg       0.97      0.97      0.97      3341
weighted avg       0.97      0.97      0.97      3341
```



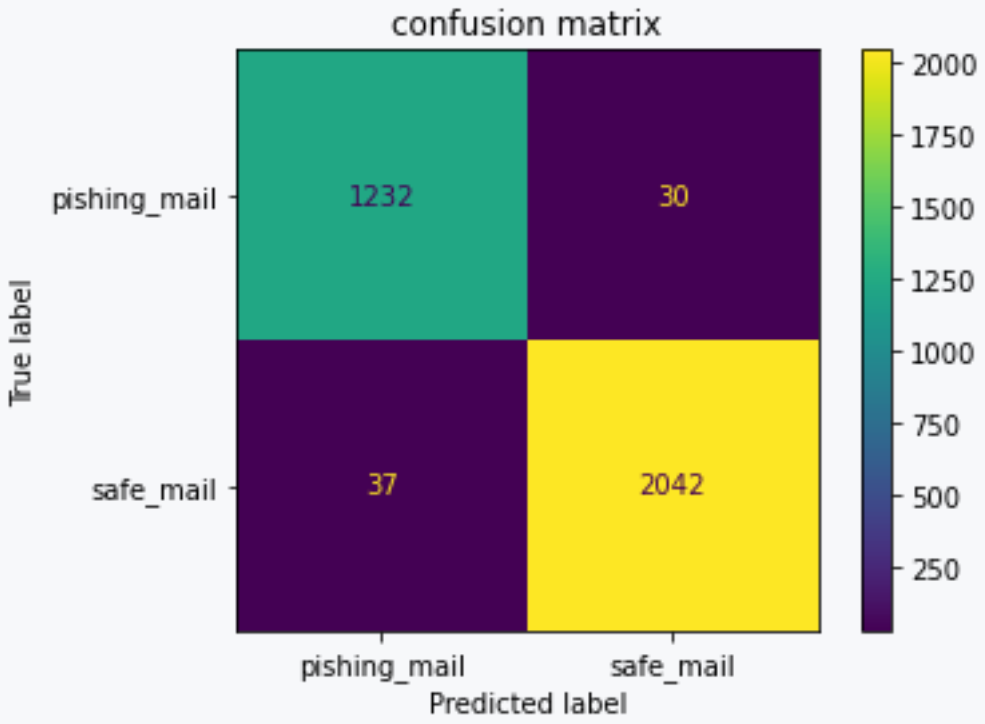
g. MLP Classifier (Multi-Layer perceptrons)

```
mlp = MLPClassifier()  # passing object
mlp.fit(x_train,y_train)
```

```
#prediction
pred_mlp = mlp.predict(x_test)

#performance
print(f"accuracy from
MLP:{accuracy_score(y_test,pred_mlp)*100:.2f} %")
print(f"f1 score from MLP: {f1_score(y_test,pred_mlp)*100:.2f}
%")
print("classification report :
\n",classification_report(y_test,pred_mlp))

#confusion matrix
clf_mlp = confusion_matrix(y_test,pred_mlp)
cx_ =
ConfusionMatrixDisplay(clf_mlp,display_labels=['pishing_mail',
'safe_mail']).plot()
plt.title("confusion matrix")
plt.show()
```

Program output:

```
accuracy from MLP:97.99 %
f1 score from MLP: 98.39 %
classification report :
              precision    recall  f1-score   support

           0       0.97      0.98      0.97      1262
           1       0.99      0.98      0.98      2079

    accuracy                           0.98      3341
   macro avg       0.98      0.98      0.98      3341
weighted avg       0.98      0.98      0.98      3341
```

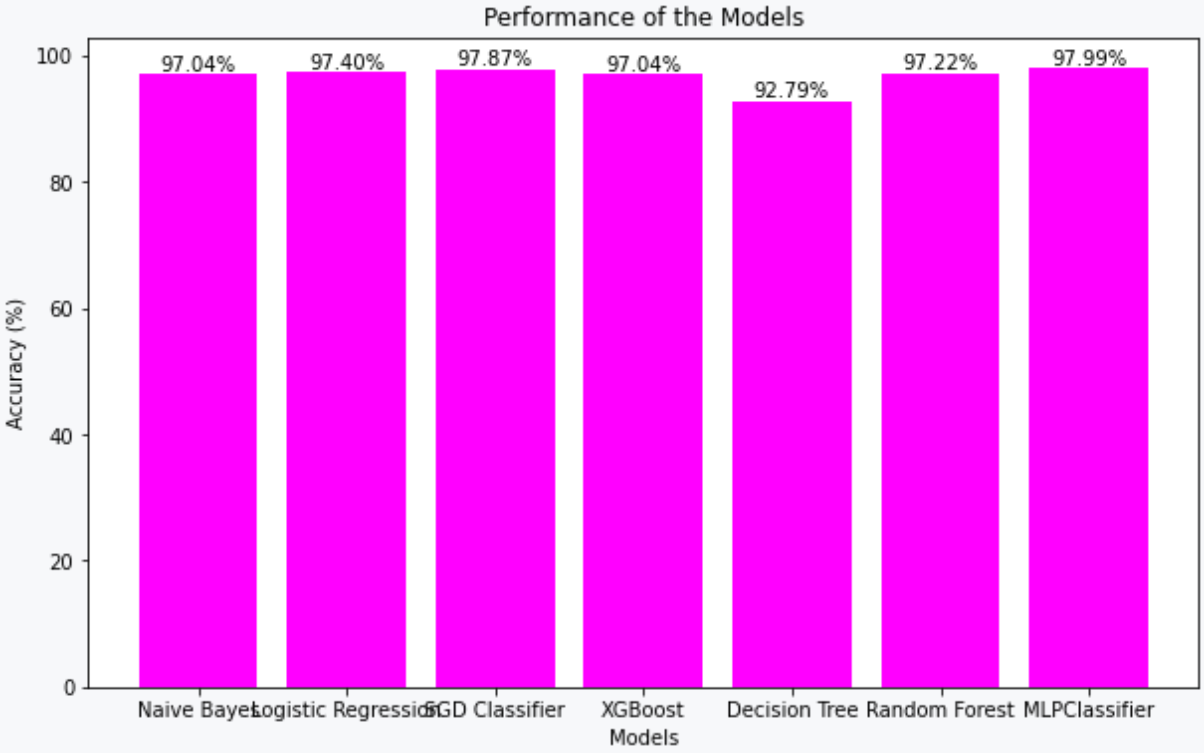


4. EDA comparison of the models performances

```
import matplotlib.pyplot as plt

# Data
models = ['Naive Bayes', 'Logistic Regression', 'SGD
Classifier', 'XGBoost', 'Decision Tree', 'Random Forest',
'MLPClassifier']
accuracies = [accuracy_score(y_test,pred_nav)*100,
accuracy_score(y_test,pred_lg)*100,
accuracy_score(y_test,pred_sgd)*100,
accuracy_score(y_test,pred_xgb)*100,
accuracy_score(y_test,pred_dtr)*100,
accuracy_score(y_test,pred_rnf)*100,
accuracy_score(y_test,pred_mlp)*100]

# Create the bar chart
plt.figure(figsize=(10, 6))
bars = plt.bar(models, accuracies, color='magenta')

# Add text labels above bars
for bar, accuracy in zip(bars, accuracies):
    plt.text(bar.get_x() + bar.get_width() / 2,
bar.get_height(), f'{accuracy:.2f}%',
             ha='center', va='bottom')
```

```
# Add titles and labels
plt.title("Performance of the Models")
plt.xlabel("Models")
plt.ylabel("Accuracy (%)")

# Show the plot
plt.show()
```

Program output:



5.3.2

Project: Phishing email detection using Neural Networks

(by https://www.kaggle.com/code/kirollosashraf/phishing-email-detection-using-deep-learning/notebook)

Compare different algorithms to identify phishing emails by Neural Networks.

Dataset:

- original: https://www.kaggle.com/datasets/subhajournal/phishingemails
- reduced: https://priscilla.fitped.eu/data/cybersecurity/phishing/phishing_email_reduced.csv

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import plotly.express as px
from sklearn.feature_extraction.text import
TfidfVectorizer,CountVectorizer
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.layers import
Embedding,GRU,LSTM,Bidirectional,SimpleRNN
from tensorflow.keras.utils import pad_sequences
from sklearn.preprocessing import LabelEncoder
from keras.models import Sequential
from keras.layers import Dense,Dropout
import tensorflow as tf
import warnings

warnings.filterwarnings('ignore')
```

1. Data understanding

```
df =
pd.read_csv("https://priscilla.fitped.eu/data/cybersecurity/ph
ishing/phishing_email_reduced.csv", delimiter=",")
# be patient
print(df.head())
```

Program output:

```
   id                                          Email Text
Email Type
0   0  re : 6 . 1100 , disc : uniformitarianism , re ...
Safe Email
1   1  the other side of * galicismos * * galicismo *...
Safe Email
2   2  re : equistar deal tickets are you still avail...
Safe Email
3   3  \nHello I am your hot lil horny toy.\n    I am...
Phishing Email
4   4  software at incredibly low prices ( 86 % lower...
Phishing Email
```

```
   id                                              Email Text
Email Type
0   0  re : 6 . 1100 , disc : uniformitarianism , re ...
Safe Email
1   1  the other side of * galicismos * * galicismo *...
Safe Email
2   2  re : equistar deal tickets are you still avail...
Safe Email
3   3  \nHello I am your hot lil horny toy.\n    I am...
Phishing Email
4   4  software at incredibly low prices ( 86 % lower...
Phishing Email
```

2. Data preprocessing

- Drop duplicates and null values
- Integer Encoding
- Remove hyperlinks, punctuations, extra space
- Converting text into vector

```
df.dropna(inplace=True,axis=0)
df.drop_duplicates(inplace=True)

le = LabelEncoder()
df["Email Type"] = le.fit_transform(df["Email Type"])

import re

def preprocess_text(text):
    # Remove hyperlinks
    text = re.sub(r'http\S+', '', text)
    # Remove punctuations
    text = re.sub(r'[^\w\s]', '', text)
    # Convert to lowercase
    text = text.lower()
    # Remove extra spaces
    text = re.sub(r'\s+', ' ', text).strip()
    return text

# Apply the preprocess_text function to the specified column
in the DataFrame
df["Email Text"] =df["Email Text"].apply(preprocess_text)

# Define the maximum length for the padded sequences
max_len = 150
```

```
# Initialize a tokenizer, which will convert text to a
sequence of integers
tk = Tokenizer()

# Fit the tokenizer on the text data in the 'Email Text'
column
# This step creates a vocabulary based on word frequency in
the text data
tk.fit_on_texts(df['Email Text'])

# Convert the text data into sequences of integers, where each
integer represents a word
# This step maps each word in the text to a unique integer
based on the tokenizer's vocabulary
sequences = tk.texts_to_sequences(df['Email Text'])

# Pad the sequences so they all have the same length of
`max_len`
# Padding is applied to the end of each sequence ('post') to
make them uniform in size
vector = pad_sequences(sequences, padding='post',
maxlen=max_len)

x = np.array(vector)
y = np.array(df["Email Type"])
print(len(vector))
```

Program output:

```
16705
```

Splitting into train and test

```
#Split the dataset into train and test set
x_train, x_test, y_train, y_test =
train_test_split(vector,df['Email Type'], test_size=0.2,
random_state =0)
```

a. Simple RNN

```
# Import necessary modules
from keras.models import Sequential
from keras.layers import Embedding, SimpleRNN, Dropout, Dense

# Initialize the model using the Sequential API
```

```
model_smp = Sequential()  # Sequential API allows adding
layers step-by-step.

# Add an Embedding layer
model_smp.add(Embedding(
    input_dim=len(tk.word_index) + 1,  # Vocabulary size (+1
for padding)
    output_dim=50,                     # Embedding dimension
(50-dimensional vector for each word)
    input_length=150                   # Input sequence length
(each sequence has 150 words)
))

# Add a SimpleRNN layer
model_smp.add(SimpleRNN(units=100))  # RNN layer with 100
units (neurons), designed to capture temporal dependencies.

# Add a Dropout layer
model_smp.add(Dropout(0.45))  # Dropout layer with a rate of
0.45 (randomly sets 45% of input units to zero during
training).

# Add a Dense output layer
model_smp.add(Dense(1, activation='sigmoid'))  # Output layer
for binary classification with a sigmoid activation.

# Compile the model
model_smp.compile(
    loss='binary_crossentropy',  # Binary cross-entropy is
appropriate for binary classification.
    optimizer='adam',            # Adam optimizer, a popular
and efficient optimization algorithm.
    metrics=['accuracy']         # Accuracy metric to evaluate
model performance during training and testing.
)

# Display the model architecture summary
model_smp.summary()  # Shows layer details, output shapes, and
number of parameters.
```

Program output:

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
```

```
=================================================================
===
 embedding_1 (Embedding)      (None, 150, 50)           8262150

 simple_rnn_1 (SimpleRNN)     (None, 100)               15100

 dropout_1 (Dropout)          (None, 100)               0

 dense_1 (Dense)              (None, 1)                 101

=================================================================
===
```

```
# Train the model with the training data (x_train and y_train)
# The model will train for 7 epochs with a batch size of 16
# During training, it also evaluates performance on the
validation data (x_test, y_test)
historical_smp = model_smp.fit(
    x_train,             # Training features
    y_train,             # Training labels
    epochs=7,            # Number of times to iterate over the
training data
    batch_size=16,       # Number of samples per gradient
update
    validation_data=(x_test, y_test)  # Data for validation
after each epoch
)

import matplotlib.pyplot as plt
pd.DataFrame(historical_smp.history)

pd.DataFrame(historical_smp.history)[['accuracy',
'val_accuracy']].plot()
plt.title('Training Accuracy')
plt.xlabel('Epochs')
plt.ylabel('accuracy')

pd.DataFrame(historical_smp.history)[['loss',
'val_loss']].plot()
plt.title('Model Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
```

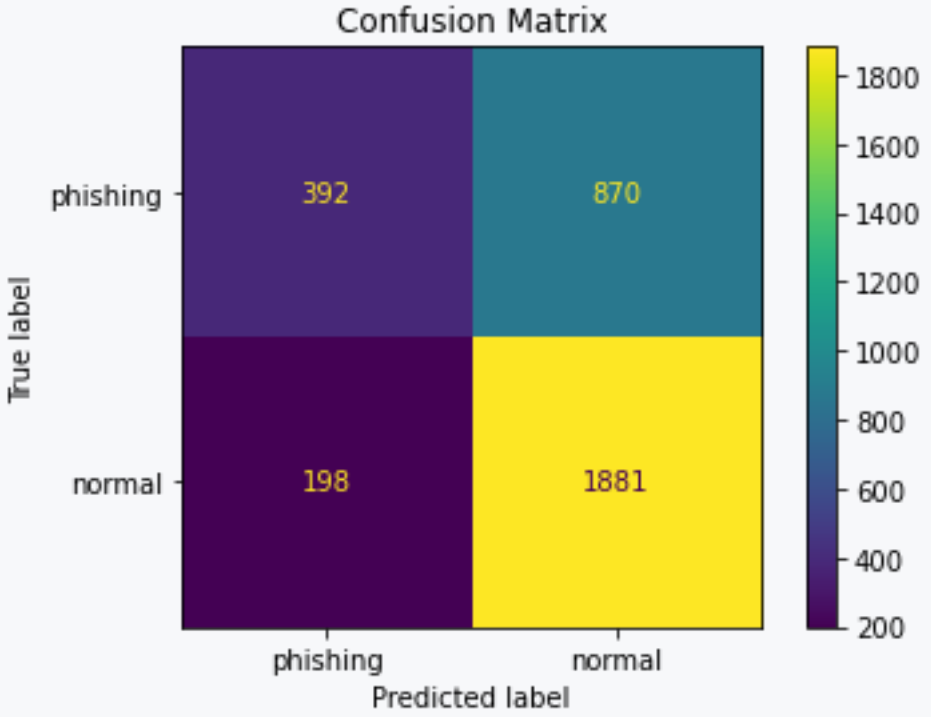
```
# Predict probabilities on the test data
```

```
# The model outputs probabilities for each sample in x_test
y_pred_prob_smp = model_smp.predict(x_test)

# Convert probabilities to binary predictions (0 or 1)
# A threshold of 0.5 is used: values greater than 0.5 are
classified as 1, otherwise as 0
y_pred_smp = (y_pred_prob_smp > 0.5).astype(int)

# Import necessary libraries for confusion matrix
from sklearn.metrics import confusion_matrix,
ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Generate a confusion matrix to evaluate the model's
predictions
cnf_smp = confusion_matrix(y_test, y_pred_smp)

# Create a ConfusionMatrixDisplay object for better
visualization
# The display labels ('phishing' and 'normal') are used to
indicate the two classes
ax_smp = ConfusionMatrixDisplay(confusion_matrix=cnf_smp,
display_labels=['phishing', 'normal']).plot()

# Add a title to the confusion matrix plot
plt.title("Confusion Matrix")

# Display the confusion matrix plot
plt.show()
```

Program output:

```
  1/105 [..............................] - ETA: 30s
  5/105 [>.............................] - ETA: 1s
  9/105 [=>............................] - ETA: 1s
 14/105 [===>.........................] - ETA: 1s
 19/105 [====>........................] - ETA: 1s
 24/105 [=====>.......................] - ETA: 1s
 29/105 [=======>.....................] - ETA: 0s
 34/105 [========>....................] - ETA: 0s
 39/105 [==========>..................] - ETA: 0s
 44/105 [===========>.................] - ETA: 0s
 49/105 [=============>...............] - ETA: 0s
 54/105 [==============>..............] - ETA: 0s
 59/105 [===============>.............] - ETA: 0s
 64/105 [=================>...........] - ETA: 0s
```

```
 69/105 [==================>...........] - ETA: 0s
 74/105 [====================>.........] - ETA: 0s
 79/105 [=====================>........] - ETA: 0s
 84/105 [======================>......] - ETA: 0s
 89/105 [=======================>.....] - ETA: 0s
 94/105 [========================>....] - ETA: 0s
 99/105 [=========================>..] - ETA: 0s
104/105 [==========================>.] - ETA: 0s
105/105 [==============================] - 2s 12ms/step
```



b. LSTM

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) designed to capture long-term dependencies in sequential data, making them particularly useful for tasks such as time series prediction, natural language processing, and more.

```
# Importing necessary libraries
from keras.models import Sequential
from keras.layers import Embedding, LSTM, Dropout, Dense

# Initialize a sequential model using the Sequential API
model = Sequential()  # This creates an empty model where
layers can be added sequentially.

# Add an Embedding layer
model.add(Embedding(
```

```
    input_dim=len(tk.word_index) + 1,  # Size of the
vocabulary (+1 for padding)
    output_dim=50,                     # Dimension of the
dense embedding (50-dimensional vectors)
    input_length=150                    # Length of input
sequences (150 words)
))

# Add an LSTM layer
model.add(LSTM(units=100))  # This layer contains 100 LSTM
units (cells) for learning sequences.

# Add a Dropout layer
model.add(Dropout(0.5))  # This layer randomly sets 50% of the
input units to 0 during training to prevent overfitting.

# Add a Dense output layer
model.add(Dense(1, activation='sigmoid'))  # Output layer with
a single unit for binary classification, using a sigmoid
activation function.

# Compile the model
model.compile(
    loss='binary_crossentropy',  # Loss function for binary
classification problems
    optimizer='adam',            # Adam optimizer for
adjusting weights during training
    metrics=['accuracy']         # Metric to evaluate the
model's performance during training and testing
)

# Display the model summary
model.summary()  # This prints a summary of the model
architecture, including layer types, output shapes, and number
of parameters.
```

Program output:

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_2 (Embedding)     (None, 150, 50)           8262150
```

```
 lstm (LSTM)                 (None, 100)               60400

 dropout_2 (Dropout)         (None, 100)               0

 dense_2 (Dense)             (None, 1)                 101
```

```
# Train the model with the training data (x_train and y_train)
# The model will train for 5 epochs with a batch size of 16
# During training, it also evaluates performance on the
validation data (x_test, y_test)
historical = model.fit(
    x_train,            # Training features
    y_train,            # Training labels
    epochs=2,           # Number of times to iterate over the
training data
    batch_size=16,      # Number of samples per gradient
update
    validation_data=(x_test, y_test)  # Data for validation
after each epoch
)
```

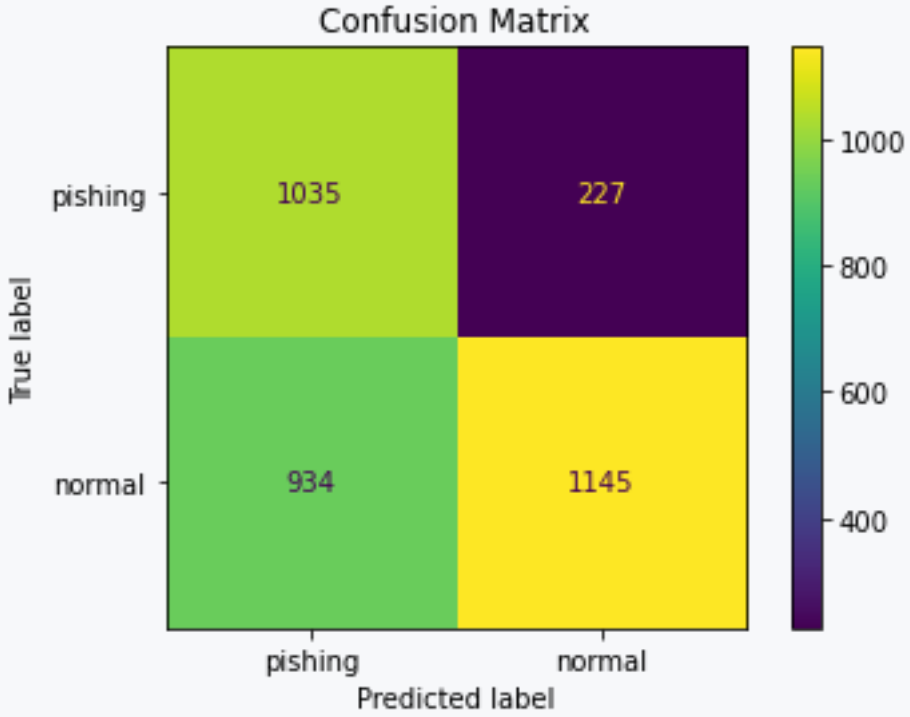
3. Performance

```
# Evaluate the model on the test data (x_test and y_test)
results = model.evaluate(x_test, y_test)

# Extract the loss value from the evaluation results
loss = results[0]  # First element is the model's loss on the
test data

# Extract the accuracy value from the evaluation results
accuracy = results[1]  # Second element is the model's
accuracy on the test data

# Print out the results with formatted strings
print(f"Model Loss: {loss}")
print(f"Model Accuracy: {accuracy * 100}")

# Generate predicted probabilities for the test set
y_pred_prob = model.predict(x_test)

# Apply a threshold of 0.5 to convert probabilities to binary
predictions (1 or 0)
```

```
y_pred = (y_pred_prob > 0.5).astype(int)

# Convert the training history into a DataFrame for easier
analysis
pd.DataFrame(historical.history)

# Plot training and validation accuracy over epochs
pd.DataFrame(historical.history)[['accuracy',
'val_accuracy']].plot()
plt.title('Training Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')

# Plot training and validation loss over epochs
pd.DataFrame(historical.history)[['loss', 'val_loss']].plot()
plt.title('Model Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
```

```
cnf = confusion_matrix(y_test,y_pred)
ax =
ConfusionMatrixDisplay(confusion_matrix=cnf,display_labels=['p
ishing','normal'])
ax.plot()
plt.title("Confusion Matrix")
plt.show()
```

Program output:



c. Bidirectional

This bidirectional LSTM model is designed for binary classification of text sequences. It starts with an embedding layer that converts words into vectors, followed by a bidirectional LSTM layer that captures context from both past and future words. A dropout layer is added to prevent overfitting, and a dense layer with sigmoid activation outputs the probability of the positive class. The model is compiled with binary cross-entropy loss, Adam optimizer, and accuracy metric for training and evaluation.

```
# Import necessary modules
from keras.models import Sequential
from keras.layers import Embedding, LSTM, Dropout, Dense,
Bidirectional

# Initialize the model using the Sequential API
model_bi = Sequential()  # Allows adding layers in sequence.

# Add an Embedding layer
model_bi.add(Embedding(
    input_dim=len(tk.word_index) + 1,  # Vocabulary size (+1
for padding)
    output_dim=50,                     # Embedding dimension
(50-dimensional vector for each word)
```

```
    input_length=150                    # Input sequence length
(each sequence has 150 words)
))

# Add a Bidirectional LSTM layer
model_bi.add(Bidirectional(LSTM(units=100)))  # Bidirectional
LSTM with 100 units, processing input in both directions.

# Add a Dropout layer
model_bi.add(Dropout(0.5))  # Dropout with rate 0.5 to reduce
overfitting by randomly zeroing out 50% of inputs during
training.

# Add a Dense output layer
model_bi.add(Dense(1, activation='sigmoid'))  # Output layer
for binary classification with sigmoid activation.

# Compile the model
model_bi.compile(
    loss='binary_crossentropy',  # Binary cross-entropy loss
for binary classification.
    optimizer='adam',            # Adam optimizer, effective
for a wide range of tasks.
    metrics=['accuracy']         # Track accuracy during
training and evaluation.
)

# Display the model architecture summary
model_bi.summary()  # Shows model layers, output shapes, and
parameter counts.
```

```
# Train the model
historical = model_bi.fit(
    x_train,                # Training data features
    y_train,                # Training data labels
    epochs=2,               # Number of epochs (iterations over
the entire dataset)
    batch_size=16,          # Number of samples per gradient
update
    validation_data=(x_test, y_test)  # Data for evaluating
loss and accuracy at the end of each epoch
)

# Evaluate the model on the test set
```

```
model_bi.evaluate(x_test, y_test)

# Generate predicted probabilities on the test set
y_pred_prob_bi = model_bi.predict(x_test)

# Apply a threshold to convert probabilities to binary
predictions
y_pred_bi = (y_pred_prob_bi > 0.5).astype(int)

# Create and display a confusion matrix
from sklearn.metrics import confusion_matrix,
ConfusionMatrixDisplay
cnf_bi = confusion_matrix(y_test, y_pred_bi)

# Plot confusion matrix
ax_bi = ConfusionMatrixDisplay(confusion_matrix=cnf_bi,
display_labels=['Phishing', 'Normal'])
ax_bi.plot()
plt.show()
```

d. GRU (Gated Recurrent Unit)

This GRU model is designed for binary classification of text sequences. It starts with an embedding layer that maps words to dense vectors, followed by a GRU layer to capture sequence dependencies. A dropout layer helps reduce overfitting, and a dense layer with sigmoid activation outputs the probability of the positive class. The model is compiled with binary cross-entropy loss, Adam optimizer, and accuracy metric for training and evaluation.

```
# Import necessary modules
from keras.models import Sequential
from keras.layers import Embedding, GRU, Dropout, Dense

# Initialize the model using the Sequential API
model_gru = Sequential()  # Sequential model to stack layers
linearly.

# Add an Embedding layer
model_gru.add(Embedding(
    input_dim=len(tk.word_index) + 1,  # Vocabulary size (+1
to account for padding index)
    output_dim=50,                     # Embedding dimension
(each word is represented by a 50-dimensional vector)
    input_length=150                   # Input sequence length
(each input has 150 words)
```

```
))

# Add a GRU layer
model_gru.add(GRU(units=100))  # GRU layer with 100 units to
capture sequential patterns in the input.

# Add a Dropout layer
model_gru.add(Dropout(0.5))  # Dropout layer with a 50%
dropout rate to prevent overfitting by randomly setting half
of the input units to zero during training.

# Add a Dense output layer
model_gru.add(Dense(1, activation='sigmoid'))  # Output layer
for binary classification; sigmoid activation outputs a
probability between 0 and 1.

# Compile the model
model_gru.compile(
    loss='binary_crossentropy',  # Binary cross-entropy loss
for binary classification tasks.
    optimizer='adam',            # Adam optimizer, commonly
used for text and sequence tasks.
    metrics=['accuracy']         # Track accuracy during
training and evaluation.
)

# Display the model architecture summary
model_gru.summary()  # Summarizes the model architecture,
displaying each layer, output shape, and parameter count.

model_gru.fit(x_train,y_train, epochs=3, batch_size=16,
validation_data=(x_test,y_test))
y_pred_prob_gru = model_gru.predict(x_test)
y_pred_gru = (y_pred_prob_gru > 0.5).astype(int)



cnf_gru = confusion_matrix(y_test,y_pred_gru)
ax_gru =
ConfusionMatrixDisplay(confusion_matrix=cnf_gru,display_labels
=['Pishing','normal'])
ax_gru.plot()
plt.show()
```

5.3.3

Project: Phishing identification based on URL (Dataset description)

(by https://www.kaggle.com/code/akashkr/phishing-url-eda-and-modelling)

Dataset:

- original: https://www.kaggle.com/datasets/akashkr/phishing-website-dataset?select=dataset.csv
- local: https://priscilla.fitped.eu/data/cybersecurity/phishing/phishing_website_dataset.csv

Phishing often begins by delivering a message that contains malware targeting the user's computer or contains links to direct victims to malicious websites in order to trick them into divulging sensitive information.

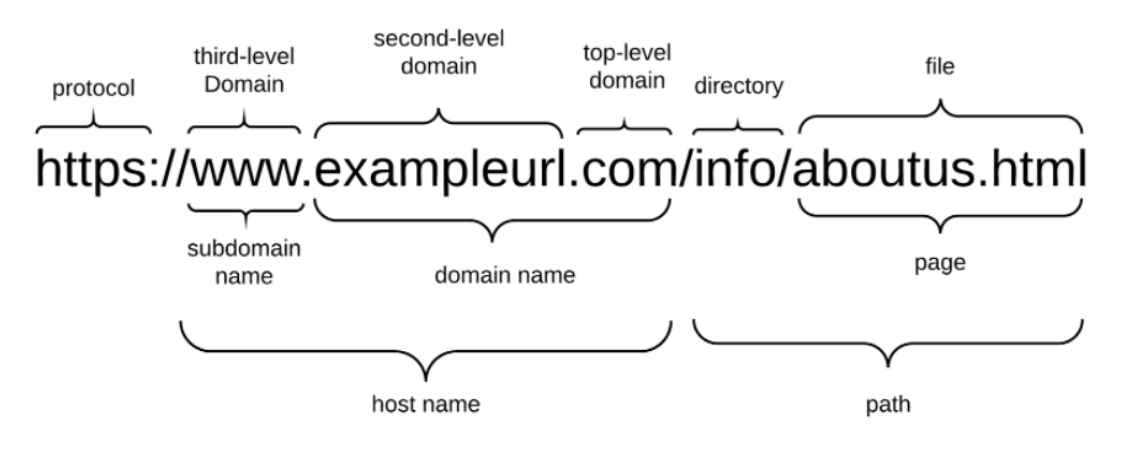
The following analysis presents what all functions we can derive and use from the domain/URL of a website to determine whether it is phishing or not.

URL components

A Uniform Resource Locator (URL) is an address used to locate web pages. The image below highlights the key parts of a typical URL.

In a phishing attack, the phisher (attacker) has full control over parts of the URL, such as the subdomain and path, and can modify them to create convincing fake addresses. For example, a phisher can use a known subdomain and path to trick users. We refer to these changeable parts of the URL as "FreeURL".

While an attacker can only register any available domain name (the main part of a URL) once, they can often modify FreeURL and create new URLs. This flexibility makes it difficult for security guards to detect phishing domains, as each FreeURL can look unique even if it leads to the same fake page. However, once a domain is confirmed to be fraudulent, defenders can block it to prevent users from accessing it.



Source: https://www.kaggle.com/code/akashkr/phishing-url-eda-and-modelling/notebook

Domain analysis and dataset description

Dataset Description

In this dataset, URLs have been analyzed for certain features that can indicate whether a URL is legitimate or phishing. These features fall into four main categories:

1. **Address bar-based features**
2. **Abnormal-based features**
3. **HTML and JavaScript-based features**
4. **Domain-based features**

1. Address bar-based features

These features help identify phishing URLs based on suspicious patterns in address bars, domains, and structure.

Using an IP address

- If the URL uses an IP address (e.g., `http://125.98.3.123/fake.html`), it's likely phishing.
- Rule: If Domain Part has IP Address → Phishing; otherwise → Legitimate.

Long URL to hide suspicious parts

- Phishing URLs often use long URLs to conceal suspicious information.
- Rule: URL length < 54 → Legitimate; length ≥ 54 and ≤ 75 → Suspicious; otherwise → Phishing.

Using URL shortening services (e.g., TinyURL)

- Shortened URLs can disguise the real destination.
- Rule: If TinyURL → Phishing; otherwise → Legitimate.

URLs containing "@" symbol

- The "@" symbol in a URL causes the browser to ignore everything before it, often redirecting users.
- Rule: If URL Contains @ Symbol → Phishing; otherwise → Legitimate.

Redirecting using "//"

- If "//" appears outside of the first few characters in the URL, it can indicate a redirection to a phishing site.
- Rule: Last occurrence of "//" in URL > 7 → Phishing; otherwise → Legitimate.

Adding prefix/suffix with "-" in domain

- Phishing sites may use "-" in domain names to mimic legitimate websites.
- Rule: If Domain Name has "-" Symbol → Phishing; otherwise → Legitimate.

Subdomain and multi-subdomains

- Extra subdomains may indicate phishing.
- Rule: 1 dot → Legitimate; 2 dots → Suspicious; more than 2 dots → Phishing.

HTTPS (Secure)

- HTTPS indicates a secure connection but does not guarantee legitimacy. A trusted certificate is required.
- Rule: Trusted HTTPS with age ≥ 1 year → Legitimate; Untrusted HTTPS → Suspicious; otherwise → Phishing.

Domain registration length

- Phishing domains often have short registration periods.
- Rule: Domain expires ≤ 1 year → Phishing; otherwise → Legitimate.

Favicon

- If the favicon (website icon) loads from a different domain, it may indicate phishing.
- Rule: Favicon loaded from external domain → Phishing; otherwise → Legitimate.

Using non-standard port

- Phishing sites may use unusual ports.
- Rule: Non-standard port → Phishing; otherwise → Legitimate.

“HTTPS” token in domain

- Phishers may add “HTTPS” in the domain name to appear legitimate.
- Rule: “HTTPS” in domain part → Phishing; otherwise → Legitimate.

2. Abnormal-based features

These features examine the structure of websites and their components to detect potential phishing. Here is an overview of each feature with classification rules:

Request URL

- This function checks whether embedded content (eg images, videos) is loaded from an external domain. Legitimate sites often have these resources on the same domain.
- rule:
- % of Request URL < 22% → Legitimate
- 22% ≤ % of request URL ≤ 61% → Suspicious
- Otherwise → Phishing

Anchor URL

- An anchor (defined by <a> tags) usually links within the same domain on legitimate pages. Phishing sites often link to other domains or use placeholders.
- rule:
- % Anchor URL < 31% → Legitimate
- 31% ≤ % of anchor URL ≤ 67% → Suspicious
- Otherwise → Phishing

Links in <Meta>, <Script> and <Link> tags

- Legitimate sites often use these tags to link to resources within the same domain.
- rule:
- % of links in <Meta>, <Script> and <Link> < 17% → Legitimate
- 17% ≤ % of links ≤ 81% → Suspicious
- Otherwise → Phishing

Server form handler (SFH)

- The SFH should match the site's domain. Phishing sites often use blank or unrelated domains in their form handlers.

- rule:
- SFH is about:blank or empty → Phishing
- SFH refers to another domain → Suspicious
- Otherwise → Legitimate

Sending information to e-mail

- Phishers can use mailto: or server-side scripts (eg mail() in PHP) to send user information directly to their email.
- rule:
- If you use mailto: or mail() → Phishing
- Otherwise → Legitimate

Unusual URL

- This function uses the WHOIS database to check if the website identity is part of the URL. Legitimate sites usually have the host name in the URL.
- rule:
- If the hostname is not included in the URL → Phishing
- Otherwise → Legitimate

3. HTML and JavaScript-based features

These features identify phishing by detecting unusual behavior and website structures. They help identify phishing attempts by detecting hidden redirects, status bar manipulation, right-click disabling, and pop-up requests for personal data.

Website redirection

- Legitimate websites are usually only redirected once, while phishing sites often redirect users multiple times.
- rule:
- Redirects ≤ 1 → Legitimate
- Redirects between 2 and 4 → Suspicious
- Redirects ≥ 4 → Phishing

Customize the status bar

- Phishers can use JavaScript to change the URL of the status bar using the onMouseOver event, hiding the actual link.
- rule:
- If onMouseOver changes the status bar to → Phishing
- Otherwise → Legitimate

Disable right click

- Phishers often disable the right-click functionality (usually via JavaScript) to prevent users from viewing the source code.
- rule:
- Right-click Disabled → Phishing
- Otherwise → Legitimate

Using a pop-up window

- Phishing sites may use pop-ups to collect personal information. Legitimate sites may also use pop-ups, but they generally don't ask for sensitive data.
- rule:
- The pop-up window contains text fields for information → Phishing
- Otherwise → Legitimate

IFrame redirection

- Iframes embed one web page inside another and are sometimes used by fraudsters to display hidden content.
- rule:
- Uses <iframe> → Phishing
- Otherwise → Legitimate

4. Domain based features

These features help determine if a website is phishing based on domain age, DNS records, website traffic, other web metrics, and inclusion in known phishing databases.

Age of domain

- Legitimate domains generally have a minimum age of 6 months, as phishing sites are often newly created.
- Rule:
- Domain age ≥ 6 months → Legitimate
- Otherwise → Phishing

DNS record

- Phishing domains may lack DNS records or WHOIS information, as they are often created quickly and without verification.
- Rule:
- No DNS record → Phishing
- Otherwise → Legitimate

Website traffic

- Legitimate websites are typically recognized by Alexa and rank within the top 100,000. Phishing websites, with little traffic, may not appear in Alexa.
- Rule:
- Alexa rank < 100,000 → Legitimate
- Alexa rank > 100,000 → Suspicious
- Otherwise → Phishing

PageRank

- PageRank (from 0 to 1) measures site importance. Phishing sites usually have a PageRank below 0.2.
- Rule:
- PageRank < 0.2 → Phishing
- Otherwise → Legitimate

Google Index

- Google indexing indicates visibility. Phishing sites are often unindexed due to their short lifespan.
- Rule:
- Indexed by Google → Legitimate
- Otherwise → Phishing

Number of links pointing to page

- Legitimate sites often have multiple external links pointing to them, while phishing sites lack these links.
- Rule:
- No links → Phishing
- 1-2 links → Suspicious
- More than 2 links → Legitimate

Statistical-reports based feature

- Services like PhishTank and StopBadware publish lists of common phishing IPs and domains. If a domain or IP matches these lists, it is likely phishing.
- Rule:
- Domain/IP listed in top phishing reports → Phishing
- Otherwise → Legitimate

5.3.4

Project: Phishing identification based on URL (EDA and modelling)

(by https://www.kaggle.com/code/akashkr/phishing-url-eda-and-modelling)

Dataset:

- original: https://www.kaggle.com/datasets/akashkr/phishing-website-dataset?select=dataset.csv
- local: https://priscilla.fitped.eu/data/cybersecurity/phishing/phishing_website_dataset.csv

To begin our exploratory data analysis (EDA) on the phishing detection dataset, we will examine features in four key categories: URL-based, Anomalous Data-based, HTML and JavaScript-based, and Domain-based. Each category contributes unique indicators that help us detect phishing websites based on a range of structural, behavioral and domain-specific characteristics.

- URL-based features examines properties of the URL itself, such as its length, the presence of certain symbols, and potential manipulations such as the use of shortened URLs. Phishing sites often use confusing URLs to impersonate legitimate addresses, with the goal of tricking users into thinking they are on a trusted site.
- Abnormal based features focus on using external elements within a web page and aligning resources such as images or anchors to the domain. By evaluating discrepancies in these elements, we can detect suspicious activity, as legitimate websites usually associate their resources with their own domain.
- HTML and JavaScript based features examine the use of HTML and JavaScript elements that are often manipulated by fraudsters, such as changing the status bar, disabling right-clicking, or using popups. Such tactics are common on phishing sites to prevent users from verifying the authenticity of sites or prompting them to enter sensitive information.
- Domain based features analyzes domain-related attributes such as domain age, DNS records, website traffic, and PageRank. Phishing sites tend to have new domains, lack significant traffic, and often do not appear in search engine indexes, making these metrics useful for flagging potential phishing sites.

Our EDA evaluates the characteristics of each category, identifies patterns, and assesses their distribution across legitimate and phishing sites. By systematically analyzing these characteristics, we aim to better understand which features are most indicative of phishing, thereby guiding the development of effective detection models.

```
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np

%matplotlib inline
```

Data overview

```
df =
pd.read_csv('https://priscilla.fitped.eu/data/cybersecurity/ph
ishing/phishing_website_dataset.csv')
print(df.head())
```

Program output:

```
   index  having_IPhaving_IP_Address  URLURL_Length
Shortining_Service  \
0      1                           -1              1
1
1      2                            1              1
1
2      3                            1              0
1
3      4                            1              0
1
4      5                            1              0
-1

   having_At_Symbol  double_slash_redirecting  Prefix_Suffix
\
0                 1                        -1             -1
1                 1                         1             -1
2                 1                         1             -1
3                 1                         1             -1
4                 1                         1             -1

   having_Sub_Domain  SSLfinal_State
Domain_registeration_length  ...  \
0                 -1              -1
-1  ...
1                  0               1
-1  ...
2                 -1              -1
-1  ...
```

```
3                   -1                -1
1  ...
4                    1                 1
-1  ...

   popUpWidnow  Iframe  age_of_domain  DNSRecord  web_traffic
Page_Rank  \
0             1       1              -1         -1            -1
-1
1             1       1              -1         -1             0
-1
2             1       1               1         -1             1
-1
3             1       1              -1         -1             1
-1
4            -1       1              -1         -1             0
-1

   Google_Index  Links_pointing_to_page  Statistical_report
Result
0             1                       1                  -1
-1
1             1                       1                   1
-1
2             1                       0                  -1
-1
3             1                      -1                   1
-1
4             1                       1                   1
1

[5 rows x 32 columns]
   index  having_IPhaving_IP_Address  URLURL_Length
Shortining_Service  \
0      1                          -1              1
1
1      2                           1              1
1
2      3                           1              0
1
3      4                           1              0
1
4      5                           1              0
-1
```

```
   having_At_Symbol  double_slash_redirecting  Prefix_Suffix
\
0                 1                         -1             -1
1                 1                          1             -1
2                 1                          1             -1
3                 1                          1             -1
4                 1                          1             -1

   having_Sub_Domain  SSLfinal_State
Domain_registeration_length  ...  \
0                 -1               -1
-1  ...
1                  0                1
-1  ...
2                 -1               -1
-1  ...
3                 -1               -1
1  ...
4                  1                1
-1  ...

   popUpWidnow  Iframe  age_of_domain  DNSRecord  web_traffic
Page_Rank  \
0            1       1              -1         -1           -1
-1
1            1       1              -1         -1            0
-1
2            1       1               1         -1            1
-1
3            1       1              -1         -1            1
-1
4           -1       1              -1         -1            0
-1

   Google_Index  Links_pointing_to_page  Statistical_report
Result
0             1                       1                  -1
-1
1             1                       1                   1
-1
2             1                       0                  -1
-1
```

```
3              1                          -1                       1
-1
4              1                           1                       1
1

[5 rows x 32 columns]
```

```
print(list(df.columns))
```

Program output:

```
['index', 'having_IPhaving_IP_Address', 'URLURL_Length',
'Shortining_Service', 'having_At_Symbol',
'double_slash_redirecting', 'Prefix_Suffix',
'having_Sub_Domain', 'SSLfinal_State',
'Domain_registeration_length', 'Favicon', 'port',
'HTTPS_token', 'Request_URL', 'URL_of_Anchor',
'Links_in_tags', 'SFH', 'Submitting_to_email', 'Abnormal_URL',
'Redirect', 'on_mouseover', 'RightClick', 'popUpWidnow',
'Iframe', 'age_of_domain', 'DNSRecord', 'web_traffic',
'Page_Rank', 'Google_Index', 'Links_pointing_to_page',
'Statistical_report', 'Result']
```

```
df.info()
```

Program output:

```

RangeIndex: 11055 entries, 0 to 11054
Data columns (total 32 columns):
 #   Column                       Non-Null Count  Dtype
---  ------                       --------------  -----
 0   index                        11055 non-null  int64
 1   having_IPhaving_IP_Address   11055 non-null  int64
 2   URLURL_Length                11055 non-null  int64
 3   Shortining_Service           11055 non-null  int64
 4   having_At_Symbol             11055 non-null  int64
 5   double_slash_redirecting     11055 non-null  int64
 6   Prefix_Suffix                11055 non-null  int64
 7   having_Sub_Domain            11055 non-null  int64
 8   SSLfinal_State               11055 non-null  int64
 9   Domain_registeration_length  11055 non-null  int64
 10  Favicon                      11055 non-null  int64
 11  port                         11055 non-null  int64
 12  HTTPS_token                  11055 non-null  int64
```

```
 13  Request_URL                  11055 non-null  int64
 14  URL_of_Anchor                11055 non-null  int64
 15  Links_in_tags                11055 non-null  int64
 16  SFH                          11055 non-null  int64
 17  Submitting_to_email          11055 non-null  int64
 18  Abnormal_URL                 11055 non-null  int64
 19  Redirect                     11055 non-null  int64
 20  on_mouseover                 11055 non-null  int64
 21  RightClick                   11055 non-null  int64
 22  popUpWidnow                  11055 non-null  int64
 23  Iframe                       11055 non-null  int64
 24  age_of_domain                11055 non-null  int64
 25  DNSRecord                    11055 non-null  int64
 26  web_traffic                  11055 non-null  int64
 27  Page_Rank                    11055 non-null  int64
 28  Google_Index                 11055 non-null  int64
 29  Links_pointing_to_page       11055 non-null  int64
 30  Statistical_report           11055 non-null  int64
 31  Result                       11055 non-null  int64
dtypes: int64(32)
memory usage: 2.7 MB
```

There are no missing values in the dataset.

According to the Data description, these are the meaning of the values in the data

- 1 means legitimate
- 0 is suspicious
- -1 is phishing

```
for col in df.columns:
    unique_value_list = df[col].unique()
    if len(unique_value_list) > 10:
        print(f'{col} has {df[col].nunique()} unique values')
    else:
        print(f'{col} contains:\t\t\t{unique_value_list}')
```

Program output:

```
index has 11055 unique values
having_IPhaving_IP_Address contains:                [-1  1]
URLURL_Length contains:                 [ 1  0 -1]
Shortining_Service contains:                  [ 1 -1]
having_At_Symbol contains:                    [ 1 -1]
double_slash_redirecting contains:                  [-1  1]
Prefix_Suffix contains:                 [-1  1]
```

```
having_Sub_Domain contains:                  [-1  0  1]
SSLfinal_State contains:              [-1  1  0]
Domain_registeration_length contains:              [-1  1]
Favicon contains:               [ 1 -1]
port contains:                  [ 1 -1]
HTTPS_token contains:                  [-1  1]
Request_URL contains:                  [ 1 -1]
URL_of_Anchor contains:                [-1  0  1]
Links_in_tags contains:                [ 1 -1  0]
SFH contains:                   [-1  1  0]
Submitting_to_email contains:                [-1  1]
Abnormal_URL contains:                 [-1  1]
Redirect contains:              [0 1]
on_mouseover contains:                 [ 1 -1]
RightClick contains:                   [ 1 -1]
popUpWidnow contains:                  [ 1 -1]
Iframe contains:                [ 1 -1]
age_of_domain contains:                [-1  1]
DNSRecord contains:                    [-1  1]
web_traffic contains:                  [-1  0  1]
Page_Rank contains:                    [-1  1]
Google_Index contains:                 [ 1 -1]
Links_pointing_to_page contains:                  [ 1  0 -1]
Statistical_report contains:                [-1  1]
Result contains:                [-1  1]
```

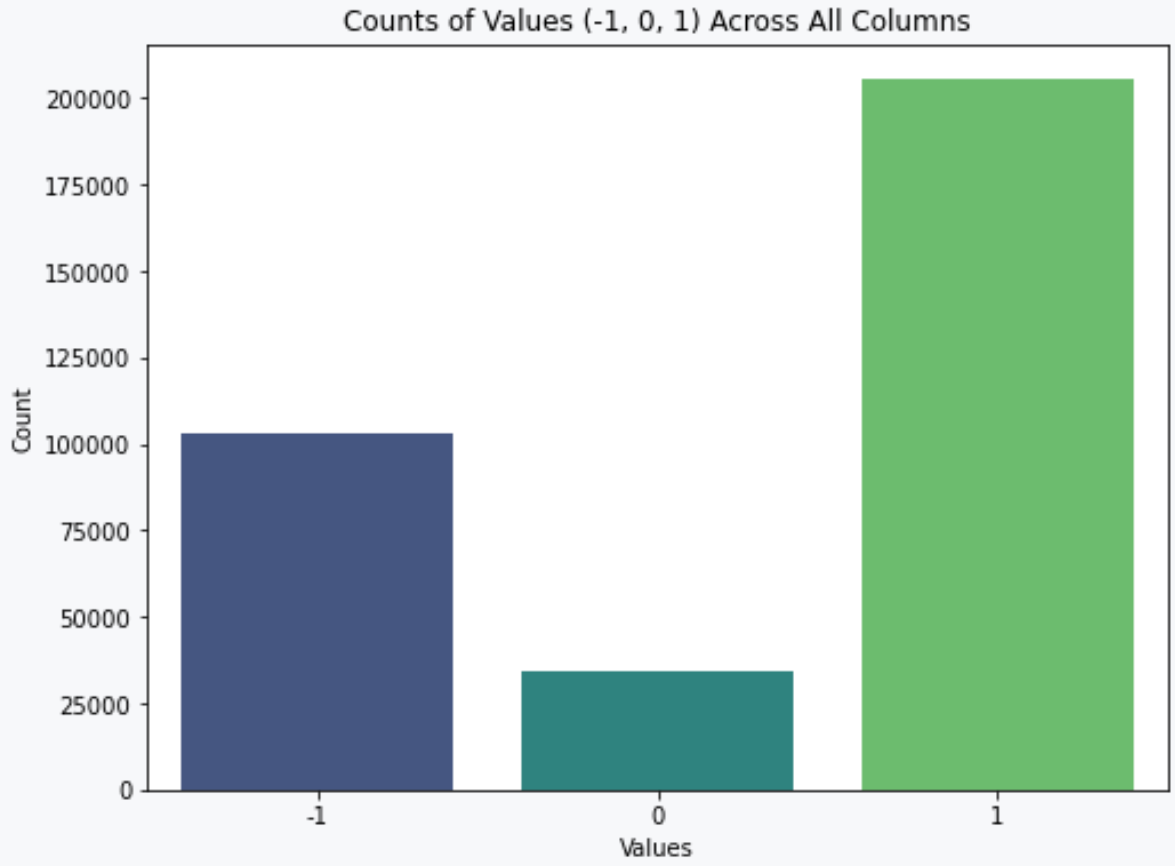
EDA

We can drop the index column because that acts as a primary key and has no significance in EDA and modelling.

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Count occurrences of each unique value across all columns
value_counts = df.apply(lambda x: x.value_counts()).fillna(0)

# Sum counts across all columns for -1, 0, and 1
total_counts = value_counts.sum(axis=1)

# Create a bar plot to visualize the counts
plt.figure(figsize=(8, 6))
```

```
sns.barplot(x=total_counts.index, y=total_counts.values,
palette='viridis')
plt.title('Counts of Values (-1, 0, 1) Across All Columns')
plt.xlabel('Values')
plt.ylabel('Count')
plt.xticks(rotation=0)  # Keep x-axis labels horizontal
plt.show()
```

Program output:



Now let's create a heatmap to visualize the correlation matrix of our data set. Here are some reasons why we use this technique:

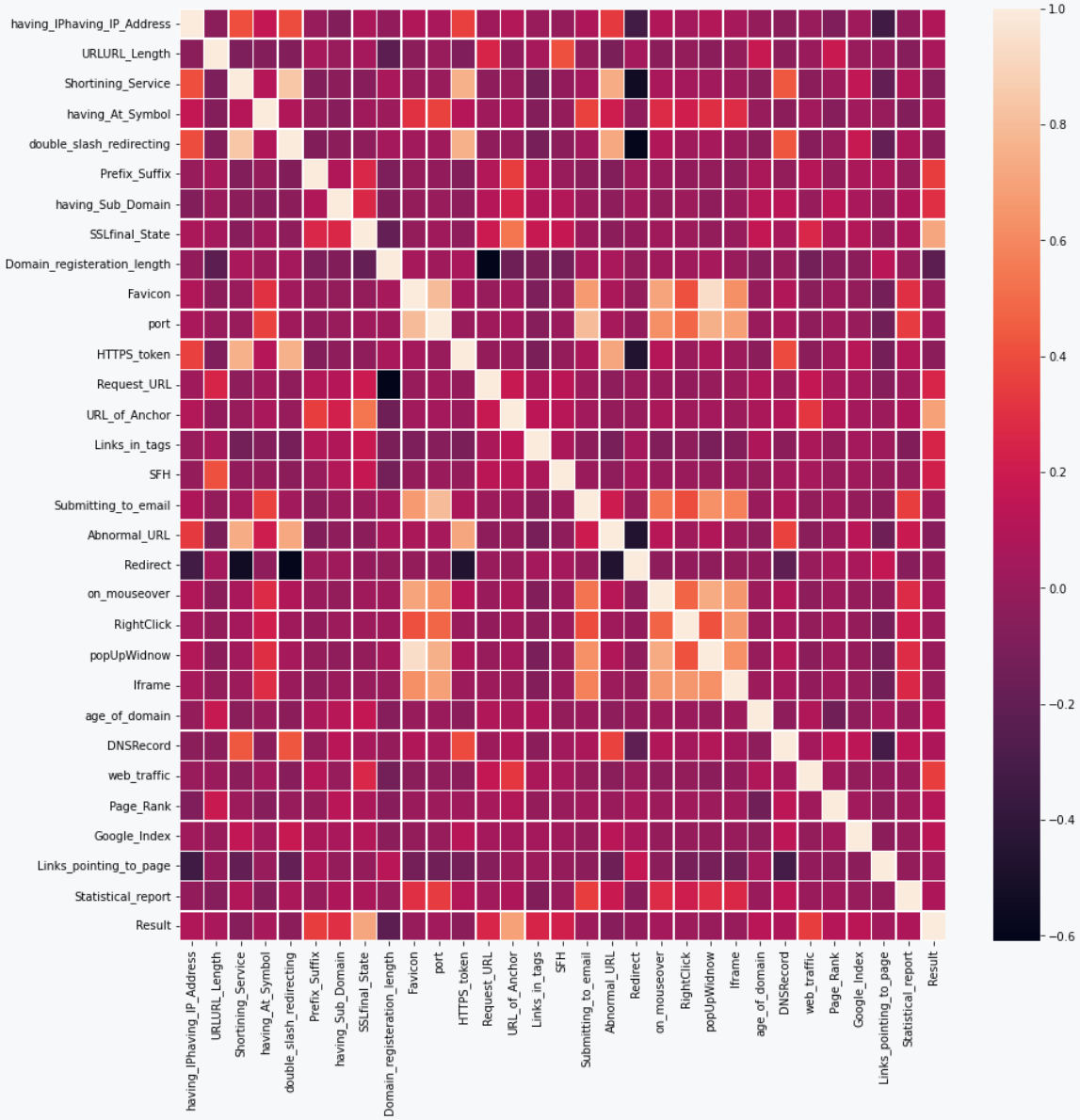
- Understanding Relationships: A heatmap allows us to easily observe and interpret relationships between different features in our dataset. By visualizing correlation coefficients, we can quickly identify which features are positively or negatively correlated, helping to understand underlying patterns in the data.
- Intuitive visualization: Color coding in the heatmap provides an intuitive way to evaluate correlations. Darker colors may indicate stronger correlations, while lighter colors indicate weaker correlations. This visual display makes it easier to understand complex relationships at a glance, as opposed to examining numerical values in a tabular format.
- Identifying multicollinearity: In many machine learning models, high multicollinearity between features can adversely affect performance. A heat

map allows us to recognize features that are highly correlated with each other, which guides us in feature selection and engineering processes. By identifying pairs of highly correlated features (close to +1 or -1), we can make informed decisions about which features to keep, combine, or remove.

- Insight into data structure: The overall structure and distribution of correlations in a data set can be revealing by looking at the data itself. For example, if we see clusters of highly correlated features, this may indicate redundancy or the presence of latent factors affecting multiple features.
- Facilitate further analysis: Insights gained from a heat map can lead to subsequent analyses, such as regression modeling or principal component analysis (PCA). By understanding the relationships between variables, we can make more informed decisions about model selection and data preprocessing steps.
- Aesthetic and informative presentation: Using a heat map increases the aesthetic appeal of our data visualizations. It not only provides background information but also engages the audience, making it an effective way to present findings in reports or presentations.

```
plt.figure(figsize=(15, 15))
sns.heatmap(df.corr(), linewidths=.5)
```

Program output:



The features **PopUpWindow** and F**avicon** show a high correlation. Based on their definitions in the Data Description, we can infer that when a website loads its favicon from external links, the pop-up window often contains text fields. This observation stems from the strong positive correlation between these two features.

Additionally, it's important to note that some features exhibit negative correlations. The minimum correlation in this context is around -0.6. Negative correlations indicate instances where one feature flags a website as phishing while another feature does not, highlighting contrasting evaluations of the website's legitimacy.

Modelling

We will use a simple tree-based classifier without hyperparameter tuning to model and test our dataset. It is important to note that we replace -1 with 0, where 0 indicates a phishing website

```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from xgboost import XGBClassifier

from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
import numpy as np

def binary_classification_accuracy(actual, pred):
    """
    This function prints the confusion matrix, accuracy score,
and classification report
    for the predicted values compared to the actual values.

    Parameters:
    - actual: Actual labels from the dataset.
    - pred: Predicted labels from the model.
    """
    print(f'Confusion matrix: \n{confusion_matrix(actual,
pred)}')
    print(f'Accuracy score: \n{accuracy_score(actual, pred)}')
    print(f'Classification report:
\n{classification_report(actual, pred)}')

# Replacing -1 with 0 in the target variable
df['Result'] = np.where(df['Result'] == -1, 0, df['Result'])
target = df['Result']
features = df.drop(columns=['Result'])

# Initialize K-Fold cross-validation
folds = KFold(n_splits=4, shuffle=True, random_state=42)

# Lists to store training and validation indices
train_index_list = list()
validation_index_list = list()

# Iterate through each fold for cross-validation
for fold, (train_idx, validation_idx) in
enumerate(folds.split(features, target)):
    # Initialize the XGBoost classifier
    model = XGBClassifier()
    # Train the model using the training data
```

```
    model.fit(np.array(features)[train_idx, :],
np.array(target)[train_idx])
    # Make predictions on the validation data
    predicted_values =
model.predict(np.array(features)[validation_idx, :])

    print(f'==== FOLD {fold + 1} ====')
    # Evaluate the model's performance

binary_classification_accuracy(np.array(target)[validation_idx
], predicted_values)
```

Program output:

```
==== FOLD 1 ====
Confusion matrix:
[[1129   54]
 [  32 1549]]
Accuracy score:
0.9688856729377714
Classification report:
              precision    recall  f1-score   support

           0       0.97      0.95      0.96      1183
           1       0.97      0.98      0.97      1581

    accuracy                           0.97      2764
   macro avg       0.97      0.97      0.97      2764
weighted avg       0.97      0.97      0.97      2764

==== FOLD 2 ====
Confusion matrix:
[[1171   45]
 [  32 1516]]
Accuracy score:
0.9721418234442837
Classification report:
              precision    recall  f1-score   support

           0       0.97      0.96      0.97      1216
           1       0.97      0.98      0.98      1548

    accuracy                           0.97      2764
   macro avg       0.97      0.97      0.97      2764
weighted avg       0.97      0.97      0.97      2764
```

```
==== FOLD 3 ====
Confusion matrix:
[[1218   34]
 [  35 1477]]
Accuracy score:
0.9750361794500724
Classification report:
              precision    recall  f1-score   support

           0       0.97      0.97      0.97      1252
           1       0.98      0.98      0.98      1512

    accuracy                           0.98      2764
   macro avg       0.97      0.97      0.97      2764
weighted avg       0.98      0.98      0.98      2764

==== FOLD 4 ====
Confusion matrix:
[[1194   53]
 [  37 1479]]
Accuracy score:
0.9674267100977199
Classification report:
              precision    recall  f1-score   support

           0       0.97      0.96      0.96      1247
           1       0.97      0.98      0.97      1516

    accuracy                           0.97      2763
   macro avg       0.97      0.97      0.97      2763
weighted avg       0.97      0.97      0.97      2763
```

The results presented here are from a four-fold cross-validation of a binary classification model, likely using the XGBoost classifier to identify phishing websites. Each fold's results provide insights into the model's performance in terms of various metrics, including accuracy, precision, recall, and F1-score. Let’s break down and interpret the results for each fold.

Fold 1

Confusion Matrix:

- True Negatives (TN): 1129 (correctly predicted legitimate websites)

- False Positives (FP): 54 (illegitimate websites incorrectly predicted as legitimate)
- False Negatives (FN): 32 (legitimate websites incorrectly predicted as phishing)
- True Positives (TP): 1549 (correctly predicted phishing websites)

Accuracy Score: 0.9689 (or 96.89%)

- This indicates that approximately 96.89% of the predictions made by the model were correct.

Classification Report:

- Precision for 0 (legitimate): 0.97 (97% of predicted legitimate websites were indeed legitimate)
- Recall for 0: 0.95 (95% of actual legitimate websites were correctly identified)
- F1-Score for 0: 0.96 (harmonic mean of precision and recall)
- Precision for 1 (phishing): 0.97 (97% of predicted phishing websites were indeed phishing)
- Recall for 1: 0.98 (98% of actual phishing websites were correctly identified)
- F1-Score for 1: 0.97

Macro and Weighted Averages: Both average scores for precision, recall, and F1-score are 0.97, indicating a balanced performance across classes.

Fold 2

- Confusion Matrix: TN: 1171, FP: 45, FN: 32, TP: 1516
- Accuracy Score: 0.9721 (or 97.21%)
- Classification Report: Precision, recall, and F1-score for both classes are similar to Fold 1, with very slight improvements.

Fold 3

- Confusion Matrix:TN: 1218, FP: 34, FN: 35, TP: 1477
- Accuracy Score: 0.9750 (or 97.50%)
- Classification Report: This fold shows the highest performance, especially with a precision of 0.98 for phishing websites.

Fold 4

- **Confusion Matrix**: TN: 1194, FP: 53, FN: 37, TP: 1479
- **Accuracy Score**: 0.9674 (or 96.74%)
- **Classification Report**: Performance is slightly lower than in other folds, but still maintains high precision and recall.

Summary of results across all folds

- **Overall Accuracy**: The accuracy across all folds ranges from about 96.74% to 97.50%, indicating the model is robust and performs consistently well in distinguishing between legitimate and phishing websites.
- **High Precision and Recall**: The precision and recall for both classes are generally high, suggesting that the model effectively minimizes false positives and false negatives.
- **Generalization**: The use of K-Fold cross-validation helps ensure that the model is not overfitting to any specific subset of the data, as it is evaluated on different segments of the dataset.
- **Balanced Performance**: The macro and weighted averages across all metrics are consistently around 0.97, indicating that the model maintains a balance in performance across both classes without favoring one over the other.

In conclusion, these results suggest that the tree-based classifier (XGBoost) is effective for the task of phishing detection, exhibiting high levels of accuracy, precision, recall, and F1 scores across multiple folds of validation.

5.3.5

Project: Phishing fraud email dataset

(by https://www.kaggle.com/datasets/charlottehall/phishing-email-data-by-type?select=phishing_data_by_type.csv)

This dataset contains textual data extracted from 160 emails, which includes the subject, text, and classification of each email as a specific type of phishing or spam. The dataset includes four categories: fraud, false alarms (legitimate emails), phishing and commercial spam, with 40 samples assigned to each category. Such data can be used to develop a more sophisticated spam filter and has potential applications in cyber security.

Dataset

- Original: https://www.kaggle.com/datasets/charlottehall/phishing-email-data-by-type?select=phishing_data_by_type.csv
- Local: https://priscilla.fitped.eu/data/cybersecurity/phishing/phishing_fraud.csv

Analyzing and classifying phishing emails using machine learning

Project Overview

In this project, you will analyze a dataset containing 160 emails, categorized into four types: fraud, false positives (legitimate emails), phishing, and commercial spam, with 40 examples of each category. The goal is to perform exploratory data analysis (EDA)

and develop a machine learning model to classify these emails based on their content and subject lines.

Objectives

1. **Exploratory Data Analysis (EDA)**: Understand the dataset by exploring the distribution of email types, examining text characteristics, and visualizing patterns.
2. **Data Preprocessing**: Clean and prepare the email data for analysis, including text normalization, tokenization, and feature extraction.
3. **Model Development**: Build a machine learning classifier to distinguish between different types of emails using appropriate algorithms.
4. **Model Evaluation**: Assess the performance of the classification model using metrics like accuracy, precision, recall, and F1 score.
5. **Documentation and Reporting**: Compile a report detailing the findings from EDA, the modeling process, and recommendations for improving email classification systems.

Data exploration:

- Load the dataset using `pandas` and display basic statistics.
- Visualize the distribution of email types using bar plots.
- Analyze the text data by examining word counts, unique words, and the length of emails.

```
# write your code
```

Data preprocessing:

- Clean the text data by removing punctuation, special characters, and stop words.
- Normalize the text (lowercasing).
- Tokenize the emails and convert them into a suitable format for machine learning (e.g., using Bag of Words or TF-IDF vectorization).

```
# write your code
```

Feature engineering:

- Explore additional features such as the length of the subject line, the presence of specific keywords, or the frequency of certain terms.
- Create a feature set that combines various characteristics of the emails.

```
# write your code
```

Model development:

- Split the dataset into training and testing sets.

- Choose machine learning algorithms to experiment with (e.g., Logistic Regression, Decision Trees, Random Forests, or Support Vector Machines).
- Train the models on the training set and evaluate them using the testing set.

```
# write your code
```

Model evaluation:

- Use classification metrics (accuracy, precision, recall, F1 score) to evaluate model performance.
- Analyze confusion matrices to understand misclassifications.

```
# write your code
```

Reporting:

- Document the findings of the EDA, preprocessing steps, model performance, and conclusions.
- Suggest possible improvements to the classification process, such as using more advanced techniques (e.g., deep learning or ensemble methods).

```
# write your code and / or report
```

5.4 Challenges in phishing detection

🕮 5.4.1

Challenges in link detection

In the past, some email protection systems have taken the drastic measure of removing all hyperlinks from incoming messages or blocking such communications outright. However, this overly simplistic approach has proven to be too restrictive and ineffective, especially considering that the inclusion of hyperlinks in digital communication has become commonplace. As a result, a more elaborate and sophisticated strategy is necessary to balance security with the legitimate use of links in emails.

One of the significant challenges in detecting hyperlinks is the various ways attackers can disguise links in message bodies. For example, links can be hidden in images, alt text, or even represented as QR codes, complicating detection efforts. While identifying a URL that is hidden behind an image or alt text is generally easy—since it can often be found in the message's source code—detecting URLs hidden in QR codes presents a more serious challenge. In this case, the URL is embedded in an image and is not represented as text anywhere in the message itself, making detection by traditional methods much more difficult.

The proliferation of QR codes in marketing and digital communications further complicates the issue as they become increasingly popular for providing quick access to websites. Attackers are taking advantage of this trend by inserting malicious URLs into QR codes, leading unsuspecting users to fraudulent websites. Therefore, the task of detecting images that contain QR codes becomes critical.

By training these algorithms to recognize patterns associated with malicious QR codes, we can greatly improve our ability to detect and combat phishing attempts. Machine learning can analyze vast amounts of data to identify common features of malicious QR codes, including their typical visual patterns and the types of websites they often link to. As technology advances, improving the detection capabilities of machine learning systems will be key to defending against evolving phishing tactics.

In addition, a multi-layered approach to hyperlink detection can be developed that combines traditional methods with advanced machine learning techniques. This could include integrating heuristics that evaluate the reputation of URLs, analyzing the context in which links appear, and using image recognition technologies to identify suspicious QR codes.

5.4.2

What methods do attackers use to hide links in phishing messages?

- Images
- Alternative text
- QR codes
- Text-based links only
- Color changes in text

5.4.3

Why is detecting URLs embedded in QR codes particularly challenging?

- The URL is embedded within an image.
- QR codes are frequently used for legitimate purposes.
- Traditional detection methods are ineffective for images.
- The URL is displayed as text in the message.
- QR codes can be easily generated by anyone.

5.4.4

How can machine learning improve the detection of malicious QR codes?

- By recognizing patterns associated with harmful QR codes.
- By training on vast amounts of data to identify common traits.
- By analyzing user feedback on detected codes.
- By comparing QR codes to a database of known good URLs.
- By blocking all QR codes without analysis.

Malicious Code Detection

Chapter 6

6.1 Introduction

📖 6.1.1

Malware, short for malicious software, is a significant threat to IT systems and networks, evolving continuously since it first appeared in the 1980s. Initially, malware consisted of basic viruses, but as technology has advanced, so have the types and sophistication of malware. Today, there are many types of malware, such as viruses, ransomware, trojans, spyware, and adware. Each type serves different malicious purposes, ranging from disrupting system operations to stealing sensitive data or even encrypting files for ransom.

The increasing complexity of malware makes it more challenging to detect and analyze. Criminals deploy malware for various activities, including data theft, spam distribution, and critical system attacks. The continuous development of malware requires constant innovation in security measures to counteract these evolving threats effectively.

📝 6.1.2

Which of the following is NOT a type of malware?

- Firewall
- Trojan
- Spyware
- Adware

📝 6.1.3

What are some of the primary uses of malware by cybercriminals?

- Encrypting user files for ransom
- Stealing sensitive data
- Increasing system speed
- Sending spam emails

📖 6.1.4

Signature-based detection is one of the foundational methods used to identify malware in cybersecurity. This method works by comparing files or packets of data against a database of known malware signatures. A malware "signature" is a unique string or identifier associated with a specific type of malware. If a file's signature matches one in the database, it’s flagged as malicious. Signature-based detection is effective at identifying known threats, making it widely used in antivirus software for rapid, reliable detection.

However, this method is limited because it cannot identify new or unknown malware. Cybercriminals often change the code of malware slightly to avoid detection, creating new signatures that don't match any known database entries. Despite its limitations, signature-based detection is still essential in cybersecurity as a first layer of defense against known threats.

📝 6.1.5

Signature-based detection compares files to which of the following?

- Database of known malware signatures
- Network traffic logs
- System performance reports
- User activity history

🕮 6.1.6

Anomaly detection, also known as behavioral analysis, is another powerful technique for identifying malware. Instead of relying on known malware signatures, this method monitors the behavior of software and flags any unusual or suspicious actions that deviate from normal behavior. For instance, if a program begins accessing files it shouldn't or consuming abnormal system resources, anomaly detection systems may classify it as a potential threat.

Anomaly detection is particularly valuable for identifying new or unknown malware because it doesn't rely on prior knowledge of specific threats. However, one drawback of this method is its susceptibility to false positives, where legitimate software might be flagged as malicious due to unusual but harmless behavior. Despite this limitation, anomaly detection is a crucial component in modern malware detection strategies.

📝 6.1.7

Anomaly detection is especially useful for identifying which of the following?

- New and unknown malware
- Only encrypted malware
- Known malware
- Regular system files

📝 6.1.8

Anomaly detection analyzes software _____ to identify _____ that deviate from _____ behavior.

- behavior
- patterns

- normal

📖 6.1.9

Malware is constantly evolving to bypass security measures, often using advanced techniques such as obfuscation, code packing, and anti-analytical tools. Obfuscation involves hiding the true purpose of the code, while code packing compresses the malware code to evade detection by signature-based methods. Advanced malware can even use artificial intelligence (AI) to adapt and target specific individuals or organizations.

AI is also used on the defense side, enabling the analysis of large data sets to detect subtle patterns and characteristics of malware. For instance, sandboxes and dynamic binary instrumentation (DBI) are advanced tools that analyze malware behavior in a controlled environment, providing deeper insights into its capabilities. As AI continues to evolve, it enhances the ability of cybersecurity systems to detect and respond to sophisticated threats.

📝 6.1.10

Which of the following techniques are used by cybercriminals to bypass malware detection?

- Code obfuscation
- Code packing
- Signature creation
- Anti-analytical tools

📝 6.1.11

AI in cybersecurity can detect _____ in data and analyze _____ behavior in _____ environments.

- patterns
- controlled
- mallware

6.2 Malware detection

📖 6.2.1

Signature-based detection

Signature-based detection is one of the primary methods used in identifying malware. This approach involves comparing incoming files or packets with a database of known malware signatures. A "signature" in this context is a unique pattern of bytes associated with a specific piece of malicious code. When a signature

matches, the detection system can quickly identify the threat and take appropriate action. This method is highly effective at identifying known malware types, such as viruses and trojans, which have well-documented patterns.

However, signature-based detection has limitations. It struggles with detecting new and unknown threats, often called "zero-day" malware, because signatures for these emerging threats do not yet exist in the database. As a result, attackers who modify malware code slightly can sometimes bypass this detection method. This limitation makes signature-based detection less effective against sophisticated or novel threats.

6.2.2

What does signature-based detection primarily rely on to identify malware?

- A unique pattern of bytes
- Machine learning algorithms
- Random sampling
- User reports
- Heuristic rules

6.2.3

Signature-based detection relies on _____ of bytes that identify specific malware. However, it is less effective for _____ types of malware, which lack _____ in the database.

- patterns
- signatures
- new

6.2.4

Anomaly detection

Anomaly detection, also known as behavioral analysis, is an advanced method that identifies malware based on unusual software behavior. Unlike signature-based detection, which relies on known patterns, anomaly detection monitors how software behaves in real-time. It examines actions like system calls, file access patterns, and network communications, looking for deviations from expected behavior. Machine learning models can be trained on typical system behaviors, enabling them to recognize activities that fall outside these norms.

Anomaly detection is particularly effective at identifying new or modified malware because it doesn't rely on existing signatures. However, one downside is that it can be more prone to false positives, mistakenly identifying legitimate software as malicious due to unusual behavior. This approach works well for detecting evolving threats but may require careful tuning to reduce the occurrence of these false alerts.

6.2.5

Which of the following behaviors might anomaly detection monitor to identify malware?

- Network communications
- System calls
- File names
- Desktop settings

6.2.6

Anomaly detection focuses on _____ that deviates from typical behavior. This method can detect _____ threats but may produce _____ positives.

- behavior
- false
- new

6.2.7

Sandboxing

Sandboxing is a method that contains potentially malicious code in an isolated environment, known as a "sandbox." In this secure setting, the software can run without posing a risk to the main system. Analysts then monitor its behavior to determine if it performs any harmful actions, such as attempting to access sensitive files or communicate with external servers. If malicious behavior is detected, the software is flagged as malware and blocked from entering the main network.

This method is particularly useful for examining unknown files or programs without endangering system security. For instance, a suspicious email attachment can be opened in a sandbox to observe its behavior. However, sandboxing can be resource-intensive, as it requires a dedicated environment to run potentially harmful software. Despite this, it remains an effective tool for organizations seeking to protect against new malware.

6.2.8

Sandboxing runs suspicious code in the _____ environment where it can be safely _____. This method is useful for detecting _____ threats.

- isolated
- unknown
- observed

📖 6.2.9

Dynamic Binary Instrumentation (DBI)

Dynamic Binary Instrumentation (DBI) is an advanced technique that allows malware analysts to monitor and manipulate program execution in real-time. This process involves inserting additional instructions into the code to analyze its behavior in great detail. Through DBI, analysts can extract valuable insights about the program's structure and operations, making it possible to detect hidden malware routines and potential security threats.

DBI provides detailed information that helps to reveal malware that may evade simpler detection methods. Although DBI is highly effective for in-depth analysis, it requires significant computational resources and expertise. It's often used in specialized environments where detailed examination of malware behavior is necessary.

📝 6.2.10

DBI allows analysts to _____ program execution in detail, making it useful for uncovering _____ that simpler methods might miss. However, it demands high _____ resources.

- computational
- observe
- routines

📖 6.2.11

Network traffic analysis

Network traffic analysis is a method of detecting malware by monitoring data exchanged between devices and remote servers. Malware often communicates with external servers to download additional malicious components or transmit stolen information. By analyzing traffic patterns, such as unusual data transfers or connections to suspicious domains, network analysis tools can identify potentially harmful activity.

This approach helps to catch malware that operates covertly by spotting abnormal network behaviors. For instance, if a program on a user's system is unexpectedly communicating with unknown servers, this could indicate malware. Network traffic analysis is particularly effective for detecting malware in organizational settings, as it provides a broader view of security across all network-connected devices.

6.2.12

Network traffic analysis monitors _____ patterns to detect _____ activity, especially useful for identifying _____ threats.

- suspicious
- covert
- data

6.2.13

Heuristic analysis

Heuristic analysis is a proactive approach that uses a set of rules to identify potentially harmful code based on suspicious characteristics. Unlike signature-based detection, heuristic analysis doesn't require a predefined pattern. Instead, it evaluates code for unusual structures or operations that might indicate malicious intent. For example, a heuristic rule might flag any code that attempts to modify system files or disable security protocols.

This technique is especially useful when dealing with polymorphic malware, which frequently changes its form to avoid detection. Heuristic analysis is often used in combination with other methods to improve detection accuracy. However, it can sometimes lead to false positives, as legitimate software might occasionally exhibit similar behaviors.

6.2.14

What are characteristics of heuristic analysis in malware detection?

- Detects suspicious code structures
- Effective against polymorphic malware
- Uses predefined signatures
- Relies on user feedback

6.3 Signature based detection

6.3.1

Signature-based detection is one of the primary methods used to identify malicious data within computer networks or traffic. It works by comparing data packets or files against a database of known "signatures," which are unique patterns associated with previously identified malware or malicious activities. This approach is effective for detecting known threats, as it can quickly recognize patterns that match existing records, making it both resource-efficient and accurate. For example, if a new data packet matches the unique code pattern of a known virus, the system flags it as a threat.

A fundamental component of signature-based detection is its database of signatures, which includes patterns associated with various types of malware. This database is created by analyzing the characteristics of known malware, viruses, and other threats. Security experts constantly add new signatures based on updates in the cybersecurity field, which helps maintain the accuracy and effectiveness of signature-based detection. For instance, when a new virus is discovered, analysts study its unique code structure and add this information to the database, so future occurrences can be swiftly identified.

However, maintaining this database requires frequent updates. Without regular additions, the detection system may fail to recognize newer threats. In the fast-paced world of cybersecurity, staying updated is essential, as hackers regularly develop new versions of malware. The signature-based detection system's dependency on its database means that if updates lag, the system may become ineffective against recently developed threats.

6.3.2

Choose correct features of the signature database in malware detection:

- Requires frequent updates.
- Stores known malware patterns.
- Detects malware without updates.
- Analyzes unknown behavior.

6.3.3

When a data packet, file, or sequence of data enters a network, a signature-based detection system analyzes it for matches with the stored signatures in its database. This process, called traffic analysis, examines various aspects of incoming traffic, such as the byte patterns, file structure, and other identifying characteristics. If a match is found, the system flags the data as malicious. For instance, if a signature for a known ransomware strain is found within an email attachment, the system alerts administrators of a potential threat.

This method is efficient because it focuses on finding exact matches within incoming data, providing a reliable way to detect threats that are already known. However, the effectiveness of signature matching is limited to recognizing familiar threats. For unknown malware types, this method will not raise any alarms, which is why additional security measures are often necessary.

6.3.4

In signature-based detection, what is the purpose of traffic analysis?

- To compare incoming data with known signatures.
- To predict new types of malware.
- To identify unknown threats.

- To modify malware signatures.

📖 6.3.5

Once the signature-based detection system identifies a match between incoming data and a signature, it can trigger specific actions to mitigate the threat. Common actions include blocking suspicious traffic, quarantining or deleting a flagged file, or alerting system administrators. For example, if a signature match for a trojan is found in a network file, the system may automatically block further access to that file, quarantine it for further inspection, or delete it to prevent harm to the network.

These predefined actions help contain threats promptly, reducing the risk of malware spreading through the system. However, the detection system can only act on threats that it recognizes from its database, highlighting the importance of regular updates and the addition of complementary detection methods to safeguard against new or modified malware.

📝 6.3.6

Choose actions commonly taken by signature-based detection systems.

- Quarantine a detected file.
- Alert administrators.
- Ignore matched signatures.
- Automatically decrypt data.

📖 6.3.7

Key components of signature-based detection systems

Signature-based detection is an essential approach in cybersecurity that identifies malicious activities by comparing incoming data against known malware patterns, or "signatures." This method involves several critical components that together allow it to recognize threats effectively. Here’s a breakdown of the main parts of signature-based detection:

1. **Creation of a Signature Database**: Every signature-based detection system relies on a database of known signatures or patterns associated with malicious data. This database is created based on previously identified malware, viruses, and other threats. As new threats emerge, the database needs regular updates to stay effective. For instance, signatures may be generated from malware characteristics, such as unique byte sequences or specific code structures.
2. **Traffic Analysis**: When data—whether a packet, file, or other form of information—enters a network or computer, the detection system inspects it for any suspicious patterns. This analysis includes checking data against the signature database, scanning for known malicious traits. For example, a

suspicious attachment in an email or an unusual data packet from a network request might be flagged.
3. **Signature Matching**: The core function of this detection method involves comparing the characteristics of incoming data with the signatures in the database. If a match is found, it typically indicates the presence of a known threat. This is crucial because a high degree of accuracy in signature matching can prevent known malware from affecting the system.
4. **Alert or Response Action**: Once a signature match is detected, the system can perform predefined actions to address the threat. These actions may include blocking traffic, quarantining or deleting a suspicious file, notifying administrators, or even launching a comprehensive scan of the affected system. Such responses help contain the threat immediately, preventing it from spreading further.

While signature-based detection effectively targets known threats, it has limitations, such as its inability to identify new or "zero-day" threats and its vulnerability to obfuscation techniques. Consequently, it is often used alongside other detection methods to provide a more robust defense.

6.3.8

Signature-based detection systems rely on a database of known _____ to identify threats. This involves comparing incoming _____ with patterns in the database. When a match is detected, it signals the presence of a known _____.

- threats
- signatures
- data

6.3.9

Signature-based detection is a widely used approach in cybersecurity for identifying common malware and viruses. It operates by comparing data within a system to a database of known malware signatures, making it highly effective against familiar threats. This method provides an initial layer of protection for networks and computer systems, efficiently identifying and responding to well-known malicious activities.

However, signature-based detection also has significant limitations. Firstly, it cannot protect against unknown or "zero-day" threats, as new types of malware will lack existing signatures in the database. This limitation makes it challenging to detect emerging threats without delay. Additionally, to stay effective, signature-based systems require regular updates to keep pace with newly discovered malware. Without continuous updates, the system’s protection level decreases, leaving it vulnerable to new threats.

Another drawback of this approach is its susceptibility to obfuscation techniques. Cybercriminals often use tactics to modify the code of malware, altering its signature

without changing its functionality. Techniques like encryption, polymorphism, or "stealth" viruses can bypass signature-based detection, reducing its effectiveness. For these reasons, signature-based detection is typically combined with other methods, such as behavioral analysis or machine learning, to overcome these vulnerabilities and provide a more comprehensive security strategy.

📝 6.3.10

Signature-based detection is effective against known threats but requires regular _____ to maintain its accuracy. It is also vulnerable to _____ techniques used by attackers to alter the appearance of malicious _____.

- updates
- obfuscation
- viruses

6.4 Anomaly detection

🕮 6.4.1

Anomaly detection, also known as behavioral analysis, is a powerful malware detection technique that focuses on identifying unusual or suspicious behavior within a system. Unlike signature-based detection, which relies on known patterns of malware, anomaly detection is designed to recognize deviations from normal behavior that could indicate new, unknown threats. This approach is particularly useful for detecting malware that hasn't been identified by antivirus programs and for which no existing signature exists. Since anomaly detection focuses on identifying unusual activities, it is often applied as a secondary measure after signature-based detection, scanning "clean" data to identify potential new threats.

Anomaly detection can examine various types of data but typically focuses on activities that deviate significantly from expected behavior. For instance, certain types of suspicious actions, such as accessing memory that doesn’t belong to a process or attempting to modify system files, may be flagged. However, defining the criteria for what constitutes "abnormal" behavior is challenging because it requires a clear understanding of both expected and malicious actions in a wide range of scenarios. Consequently, anomaly detection is a more complex task than signature-based methods, as it has to account for legitimate actions that may vary by context.

Despite these complexities, anomaly detection plays a critical role in identifying emerging threats. By looking for patterns that deviate from normal operations, it can detect malware that has yet to be categorized. With its ability to discover unknown threats, anomaly detection is a vital layer of defense in cybersecurity. However, it is often most effective when combined with signature-based detection to reduce false positives and improve overall detection accuracy.

6.4.2

What is a primary advantage of using anomaly detection in malware detection?

- It detects new types of malware for which no signature exists.
- It relies on known malware signatures.
- It only works with predefined malware types.
-

6.4.3

Anomaly detection identifies malware by finding _____ in behavior that deviate from _____ norms, often applied as a second layer after _____-based detection.

- signature
- normal
- anomalies

6.4.4

Machine learning plays a pivotal role in enhancing the accuracy and adaptability of anomaly detection systems. By training algorithms on large datasets that include examples of both normal and malicious behavior, machine learning models learn to recognize subtle indicators of malware. Once trained, these models can analyze new data and detect anomalies that might signal the presence of malware. Machine learning's adaptability is crucial, as it enables anomaly detection to evolve with new types of threats that were previously undetectable using traditional methods.

Machine learning-based anomaly detection is particularly beneficial because it can adapt to new obfuscation techniques, in which malware tries to hide its presence by altering its appearance. The model's ability to learn from data allows it to spot suspicious patterns even when malware uses techniques like encryption or polymorphism to mask itself. This adaptability gives anomaly detection systems a significant advantage over signature-based systems, which struggle with these forms of evasion.

Using machine learning algorithms in anomaly detection reduces the likelihood of false alarms by distinguishing between genuinely malicious behavior and harmless deviations. By analyzing vast datasets, these systems can fine-tune their detection criteria and improve accuracy. As a result, machine learning-based anomaly detection offers a robust approach to identifying malware while minimizing the interruptions caused by false positives.

6.4.5

Which of the following are benefits of using machine learning in anomaly detection?

- Allows for adaptability to new threats
- Reduces false alarms
- Requires no training data
- Only detects known malware

6.4.6

Examples of anomalous behavior

Anomaly detection in malware detection involves identifying behaviors that are unusual or unexpected. Some examples of anomalous behavior include unauthorized access to memory, illegal API calls, and attempts to modify sensitive system files. For instance, accessing memory that does not belong to an existing process can be an indicator of malicious behavior, as this action is often used by malware to exploit system vulnerabilities. Similarly, unauthorized attempts to read or modify system files can signal an attempt to compromise the operating system.

Operating system API calls that fall outside normal parameters are another red flag for anomaly detection systems. An example is the use of specific API calls in Windows, such as Dynamic Data Exchange (DDE) in WinAPI, which can be exploited by malware to perform unauthorized actions. Monitoring these calls allows anomaly detection systems to spot and isolate potentially harmful activities before they cause damage.

By identifying such abnormal behaviors, anomaly detection provides a proactive approach to security. Instead of relying solely on known signatures, it monitors behavior and flags suspicious actions. These types of behavior-based detections offer an added layer of protection against emerging threats that may not yet have identifiable signatures.

6.4.7

Which of the following behaviors might be flagged by an anomaly detection system?

- Unauthorized memory access
- Routine file access
- Scheduled system backups
- Standard network traffic

6.4.8

Anomaly detection systems flag unusual behaviors such as _____ memory access, unauthorized _____ calls, and modifications to sensitive _____ files.

- API
- unauthorized
- system

6.4.9

Types of anomaly detection techniques

Anomaly detection methods can vary, but two primary techniques are widely used in cybersecurity: statistical and machine learning-based detection. Statistical methods involve monitoring for deviations from expected statistical characteristics in the data, such as unusual patterns in network traffic or irregular access times. By establishing a baseline of normal behavior, these methods can quickly detect outliers that may indicate malicious activities.

Machine learning-based detection, on the other hand, uses algorithms that learn from historical data. This method is dynamic, as it allows models to adjust to new threats by learning from recent instances of malware. Machine learning techniques are effective for identifying complex patterns that statistical methods might miss, providing a more robust solution for spotting advanced threats.

The choice between statistical and machine learning-based anomaly detection depends on the application and the required level of security. Statistical methods may be sufficient for straightforward systems, but complex environments benefit from the adaptability of machine learning. Combining both approaches offers a comprehensive detection strategy, as each method complements the other.

6.4.10

Which of the following are techniques used in anomaly detection?

- Statistical analysis
- Machine learning
- Fixed signature comparison
- Manual inspection

6.4.11

Combining signature-based and anomaly detection

Anomaly detection is an essential tool, but it is most effective when combined with signature-based detection. Signature-based detection excels at identifying known

threats, as it quickly matches data to an extensive database of malware signatures. However, it falls short against new threats or malware that has been altered to bypass signature recognition. Anomaly detection addresses this gap by monitoring for unusual behaviors that could signal new malware.

Combining these two methods enhances cybersecurity by providing a balanced approach. Signature-based detection serves as the first layer, swiftly identifying known threats. When no matches are found, anomaly detection takes over to scan for any suspicious activities that might indicate emerging threats. This dual-layer system reduces false positives, as only genuinely suspicious behavior triggers alerts in the absence of known malware signatures.

Incorporating both techniques provides comprehensive protection against a wide range of threats. While signature-based detection offers speed and accuracy for known threats, anomaly detection provides adaptability and resilience against evolving malware. Together, they create a robust defense mechanism for computer systems, networks, and sensitive data.

6.4.12

Why is anomaly detection often combined with signature-based detection?

- To detect rapidly both known and unknown threats
- To rely only on predefined malware signatures
- To avoid analyzing data behavior
- To replace the need for databases

AI in Malware Detection

Chapter 7

7.1 Role of AI

📖 7.1.1

Artificial Intelligence (AI) has become a pivotal force in enhancing malware detection and overall cybersecurity. By utilizing advanced algorithms and techniques, AI systems can analyze vast amounts of data, allowing them to identify patterns of normal behavior within various systems. This capability is crucial for establishing baselines of what is considered typical operation. Once these baselines are established, AI can effectively detect anomalies that deviate from them, potentially indicating the presence of malware. This approach is particularly effective for zero-day threats—new vulnerabilities that have not yet been documented or categorized.

AI's behavioral analysis is not only reactive but also proactive. By continuously monitoring for deviations, AI can identify suspicious activities that may signal an impending attack. This is especially important in today's cybersecurity landscape, where threats evolve rapidly, and traditional signature-based detection methods may fall short. As organizations increasingly rely on digital systems, AI's ability to maintain vigilance against emerging threats becomes invaluable.

Furthermore, the integration of AI into malware detection systems enhances the effectiveness of cybersecurity measures. As AI continues to learn from both benign and malicious behavior, it refines its ability to discern between normal and potentially harmful activities. This ongoing learning process equips organizations to better protect themselves against malware attacks, which are increasingly sophisticated.

📝 7.1.2

What is a primary benefit of AI in malware detection?

- It reduces the need for human intervention.
- It increases detection speed.
- It works only with known threats.
- It focuses solely on signature matching.

📝 7.1.3

AI's behavioral analysis is effective for detecting _____ threats that do not have known signatures. This method establishes _____ of normal operation and identifies _____ that deviate from these patterns.

- baselines
- anomalies
- zero-day

📖 7.1.4

Machine Learning (ML) algorithms are integral to the functionality of AI in malware detection. By training on extensive datasets that encompass examples of both harmless and malicious behavior, these algorithms can learn to recognize subtle indicators of malware. The training process involves feeding the algorithm large volumes of data, allowing it to identify complex relationships within the information. As a result, machine learning systems can adapt to new and evolving threats, including those that utilize obfuscation techniques to evade detection.

One significant advantage of using machine learning is its ability to improve detection accuracy over time. As the algorithms are exposed to more data, they refine their decision-making processes, becoming increasingly proficient at identifying potential threats. However, it is important to note that machine learning also has its limitations. For example, the effectiveness of supervised learning depends heavily on the quality and quantity of labeled data. If the dataset lacks comprehensive coverage of potential threats, the algorithm may struggle to recognize novel malware.

Additionally, unsupervised learning approaches, which do not rely on labeled data, can detect unexpected threats. However, they might misidentify benign activities as malicious, leading to false positives. The choice of machine learning technique significantly influences the effectiveness of malware detection, highlighting the need for organizations to consider their specific security requirements and the characteristics of their data when implementing AI solutions.

📝 7.1.5

Which machine learning technique is primarily used to improve detection accuracy over time?

- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning

📝 7.1.6

What are the characteristics of machine learning algorithms in malware detection?

- Require large datasets
- Recognize subtle indicators of malware
- Depend solely on labeled data
- Are inflexible and cannot adapt

📖 7.1.7

Automated threat intelligence represents a critical advancement in the realm of cybersecurity, enabling organizations to stay ahead of potential attacks. By leveraging AI capabilities, automated systems can gather and analyze vast amounts of information regarding vulnerabilities, attack vectors, and emerging threats in real time. This capability allows security teams to be proactive rather than reactive, ensuring that defenses are up-to-date with the latest intelligence.

The automation of threat intelligence significantly streamlines the process of responding to threats. Instead of relying solely on human analysts to sift through mountains of data, AI systems can rapidly assess new information, flagging relevant findings for further investigation. This expedites the identification of potential risks and enhances an organization's overall security posture. As threats continue to evolve, automated threat intelligence serves as a necessary tool for keeping security measures relevant and effective.

Additionally, the integration of automated threat intelligence with existing security systems can lead to more coordinated and efficient responses. By providing timely updates on vulnerabilities and threats, AI systems enable organizations to adjust their defenses promptly, mitigating potential risks before they escalate into serious incidents. This level of responsiveness is vital in a landscape where cyber threats are constantly changing.

📝 7.1.8

Which of the following statements are true regarding automated threat intelligence?

- It allows for proactive security measures.
- It helps keep security measures updated with new threats.
- It significantly reduces response times to potential risks.
- It operates independently of existing security systems.

📖 7.1.9

Reducing false positives with AI

One of the persistent challenges in malware detection is the occurrence of false positives—alerts that indicate a threat where none exists. These false alarms can lead to unnecessary investigations and resource allocation, ultimately detracting from the efficiency of security operations. AI plays a significant role in minimizing false positives by refining the criteria used to identify malicious activities. By learning from previous detections, AI systems can better distinguish between benign anomalies and genuine threats.

The ability of AI to reduce false positives enhances the effectiveness of malware detection strategies. For instance, when AI is trained on a comprehensive dataset that includes both normal and malicious behavior, it becomes adept at recognizing

patterns that signify actual threats. This learning process helps the AI to evolve and adapt, allowing it to provide more accurate alerts and, in turn, increasing trust in its detection capabilities.

Moreover, combining AI-driven detection with traditional methods, such as signature-based detection, can further enhance accuracy. By leveraging the strengths of both approaches, organizations can achieve a more balanced security strategy that not only identifies threats more effectively but also significantly reduces the noise created by false positives.

7.1.10

Which of the following are advantages of reducing false positives?

- Improved resource allocation
- Enhanced trust in detection capabilities
- Increased operational efficiency
- Increased alert fatigue

7.1.11

Predictive analytics and proactive defense

Predictive analytics is a powerful application of AI in cybersecurity that allows organizations to anticipate potential threats before they occur. By analyzing historical data and recognizing trends, predictive analytics can forecast future risks and vulnerabilities. This proactive approach enables organizations to strengthen their defenses ahead of time, rather than merely reacting to incidents after they happen.

Incorporating predictive analytics into a security strategy provides numerous advantages. For instance, it allows organizations to allocate resources effectively by focusing on areas identified as high-risk. This foresight is particularly beneficial in an ever-evolving threat landscape, where new attack vectors emerge regularly. By staying one step ahead, organizations can implement measures that preemptively mitigate risks.

Furthermore, predictive analytics can be integrated with machine learning algorithms, enhancing its effectiveness. As these algorithms continuously learn from new data, they refine their predictive capabilities, ensuring that organizations are equipped with the most relevant information for anticipating threats. This combination of predictive analytics and machine learning creates a robust defense mechanism that can adapt to the complexities of modern cybersecurity challenges.

7.1.12

Predictive analytics allows organizations to anticipate potential _____ before they occur by analyzing _____ data and recognizing _____.

- historical
- trends
- threats

7.2 Projects

7.2.1

Project: Malware detection

(by https://www.kaggle.com/code/maidaly/malware-detection-with-machine-learning/notebook)

Dataset

- original: https://www.kaggle.com/datasets/amauricio/pe-files-malwares
- local: https://priscilla.fitped.eu/data/cybersecurity/malware/dataset_malwares.csv

Analyze existing features and prepare a machine learning model.

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g.
pd.read_csv)
import pickle
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report,
confusion_matrix

data =
pd.read_csv('https://priscilla.fitped.eu/data/cybersecurity/ma
lware/dataset_malwares.csv')
print(data.head())
```

Program output:

```
                                                Name   e_magic
e_cblp  e_cp  e_crlc  \
```

```
0  VirusShare_a878ba26000edaac5c98eff4432723b3    23117
144     3       0
1  VirusShare_ef9130570fddc174b312b2047f5f4cf0    23117
144     3       0
2  VirusShare_ef84cdeba22be72a69b198213dada81a    23117
144     3       0
3  VirusShare_6bf3608e60ebc16cbcff6ed5467d469e    23117
144     3       0
4  VirusShare_2cc94d952b2efb13c7d6bbe0dd59d3fb    23117
144     3       0

   e_cparhdr  e_minalloc  e_maxalloc  e_ss  e_sp  ...
SectionMaxChar  \
0          4           0       65535     0   184  ...
3758096608
1          4           0       65535     0   184  ...
3791650880
2          4           0       65535     0   184  ...
3221225536
3          4           0       65535     0   184  ...
3224371328
4          4           0       65535     0   184  ...
3227516992

   SectionMainChar  DirectoryEntryImport
DirectoryEntryImportSize  \
0                0                     7
152
1                0                    16
311
2                0                     6
176
3                0                     8
155
4                0                     2
43

   DirectoryEntryExport  ImageDirectoryEntryExport
ImageDirectoryEntryImport  \
0                     0                           0
54440
1                     0                           0
262276
```

```
2                       0                             0
36864
3                       0                             0
356352
4                       0                             0
61440

   ImageDirectoryEntryResource  ImageDirectoryEntryException
\
0                         77824                         73728
1                        294912                             0
2                         40960                             0
3                       1003520                             0
4                         73728                             0

   ImageDirectoryEntrySecurity
0                            0
1                       346112
2                            0
3                     14109472
4                        90624

[5 rows x 79 columns]
```

```
data.info()
```

Program output:

```
RangeIndex: 19611 entries, 0 to 19610
Data columns (total 79 columns):
 #   Column                          Non-Null Count  Dtype
---  ------                          --------------  -----
 0   Name                            19611 non-null  object
 1   e_magic                         19611 non-null  int64
 2   e_cblp                          19611 non-null  int64
 3   e_cp                            19611 non-null  int64
 4   e_crlc                          19611 non-null  int64
 5   e_cparhdr                       19611 non-null  int64
 6   e_minalloc                      19611 non-null  int64
 7   e_maxalloc                      19611 non-null  int64
 8   e_ss                            19611 non-null  int64
 9   e_sp                            19611 non-null  int64
 10  e_csum                          19611 non-null  int64
 11  e_ip                            19611 non-null  int64
```

```
12  e_cs                        19611 non-null  int64
13  e_lfarlc                    19611 non-null  int64
14  e_ovno                      19611 non-null  int64
15  e_oemid                     19611 non-null  int64
16  e_oeminfo                   19611 non-null  int64
17  e_lfanew                    19611 non-null  int64
18  Machine                     19611 non-null  int64
19  NumberOfSections            19611 non-null  int64
20  TimeDateStamp               19611 non-null  int64
21  PointerToSymbolTable        19611 non-null  int64
22  NumberOfSymbols             19611 non-null  int64
23  SizeOfOptionalHeader        19611 non-null  int64
24  Characteristics             19611 non-null  int64
25  Magic                       19611 non-null  int64
26  MajorLinkerVersion          19611 non-null  int64
27  MinorLinkerVersion          19611 non-null  int64
28  SizeOfCode                  19611 non-null  int64
29  SizeOfInitializedData       19611 non-null  int64
30  SizeOfUninitializedData     19611 non-null  int64
31  AddressOfEntryPoint         19611 non-null  int64
32  BaseOfCode                  19611 non-null  int64
33  ImageBase                   19611 non-null  int64
34  SectionAlignment            19611 non-null  int64
35  FileAlignment               19611 non-null  int64
36  MajorOperatingSystemVersion 19611 non-null  int64
37  MinorOperatingSystemVersion 19611 non-null  int64
38  MajorImageVersion           19611 non-null  int64
39  MinorImageVersion           19611 non-null  int64
40  MajorSubsystemVersion       19611 non-null  int64
41  MinorSubsystemVersion       19611 non-null  int64
42  SizeOfHeaders               19611 non-null  int64
43  CheckSum                    19611 non-null  int64
44  SizeOfImage                 19611 non-null  int64
45  Subsystem                   19611 non-null  int64
46  DllCharacteristics          19611 non-null  int64
47  SizeOfStackReserve          19611 non-null  int64
48  SizeOfStackCommit           19611 non-null  int64
49  SizeOfHeapReserve           19611 non-null  int64
50  SizeOfHeapCommit            19611 non-null  int64
51  LoaderFlags                 19611 non-null  int64
52  NumberOfRvaAndSizes         19611 non-null  int64
53  Malware                     19611 non-null  int64
54  SuspiciousImportFunctions   19611 non-null  int64
55  SuspiciousNameSection       19611 non-null  int64
```

```
 56  SectionsLength                19611 non-null  int64
 57  SectionMinEntropy             19611 non-null  float64
 58  SectionMaxEntropy             19611 non-null  int64
 59  SectionMinRawsize             19611 non-null  int64
 60  SectionMaxRawsize             19611 non-null  int64
 61  SectionMinVirtualsize         19611 non-null  int64
 62  SectionMaxVirtualsize         19611 non-null  int64
 63  SectionMaxPhysical            19611 non-null  int64
 64  SectionMinPhysical            19611 non-null  int64
 65  SectionMaxVirtual             19611 non-null  int64
 66  SectionMinVirtual             19611 non-null  int64
 67  SectionMaxPointerData         19611 non-null  int64
 68  SectionMinPointerData         19611 non-null  int64
 69  SectionMaxChar                19611 non-null  int64
 70  SectionMainChar               19611 non-null  int64
 71  DirectoryEntryImport          19611 non-null  int64
 72  DirectoryEntryImportSize      19611 non-null  int64
 73  DirectoryEntryExport          19611 non-null  int64
 74  ImageDirectoryEntryExport     19611 non-null  int64
 75  ImageDirectoryEntryImport     19611 non-null  int64
 76  ImageDirectoryEntryResource   19611 non-null  int64
 77  ImageDirectoryEntryException  19611 non-null  int64
 78  ImageDirectoryEntrySecurity   19611 non-null  int64
dtypes: float64(1), int64(77), object(1)
memory usage: 11.8+ MB
```

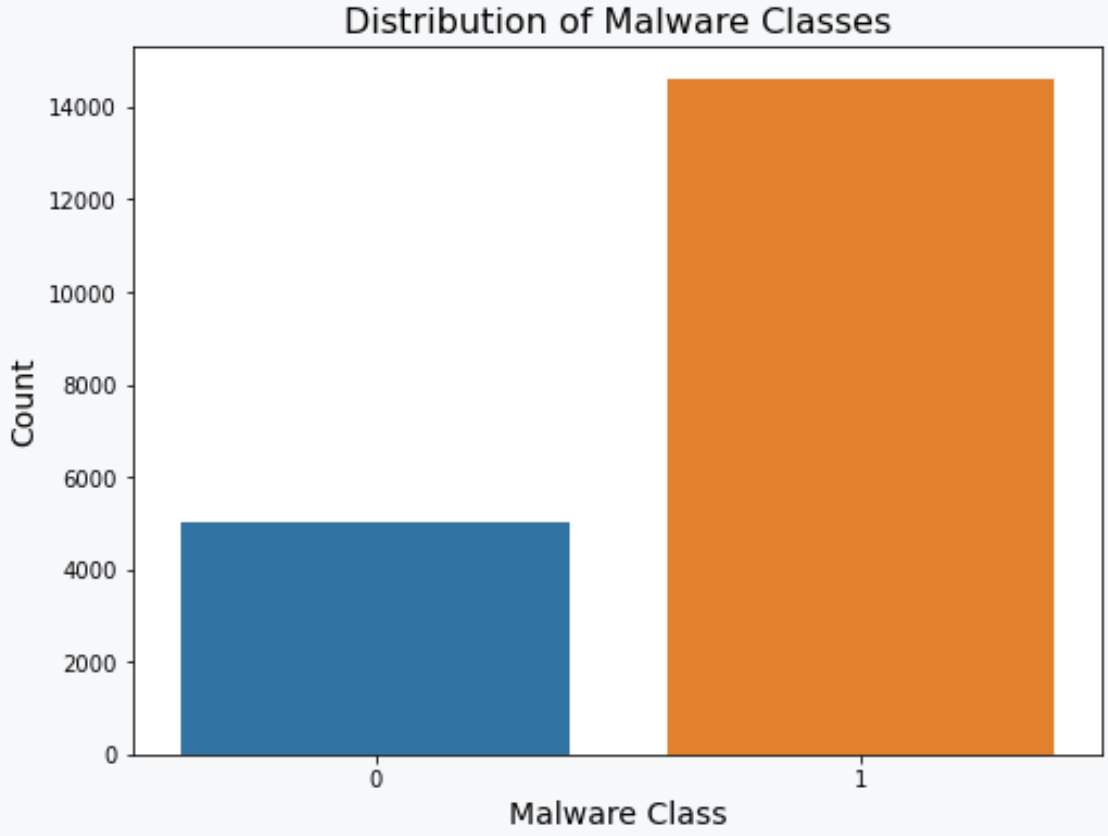
```
# Drop unnecessary columns from the dataset to focus on the
relevant features for analysis.
# The columns being removed are: 'Name', 'Machine',
'TimeDateStamp', and 'Malware'.
used_data = data.drop(['Name', 'Machine', 'TimeDateStamp',
'Malware'], axis=1)
```

Classes distribution

```
# Set up the figure size for the plot to ensure clarity and
proper display of the data.
data['Malware'] = data['Malware'].astype('category')

# Set the figure size for better visibility
plt.figure(figsize=(8, 6))

# Create a count plot to visualize the distribution of the
'Malware' classes
```

```
sns.countplot(x='Malware', data=data)

# Set the title and labels
plt.title('Distribution of Malware Classes', fontsize=16)
plt.xlabel('Malware Class', fontsize=14)
plt.ylabel('Count', fontsize=14)

# Display the plot
plt.show()
```

Program output:

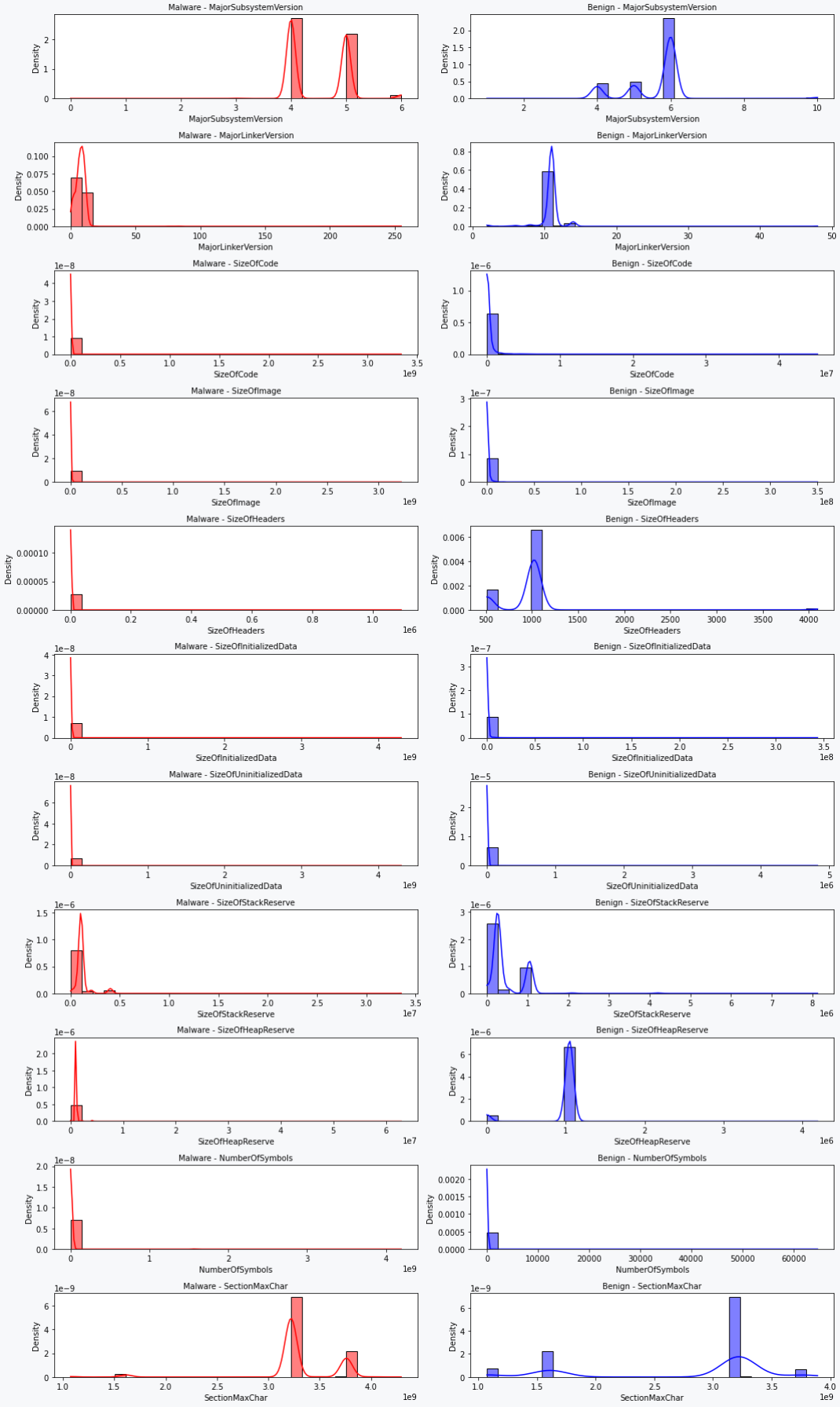


Features visualization

```
# Define the features to plot
features = ['MajorSubsystemVersion', 'MajorLinkerVersion',
'SizeOfCode', 'SizeOfImage',
            'SizeOfHeaders', 'SizeOfInitializedData',
'SizeOfUninitializedData',
            'SizeOfStackReserve', 'SizeOfHeapReserve',
'NumberOfSymbols', 'SectionMaxChar']

i = 1
```

```
# Create a figure for the subplots
plt.figure(figsize=(15, 25))

for feature in features:
    # Plot for malware samples
    ax1 = plt.subplot(len(features), 2, i)
    sns.histplot(data[data['Malware'] == 1][feature], ax=ax1,
kde=True, bins=30, color='red', stat='density')
    ax1.set_title(f'Malware - {feature}', fontsize=10)

    # Plot for benign samples
    ax2 = plt.subplot(len(features), 2, i + 1)
    sns.histplot(data[data['Malware'] == 0][feature], ax=ax2,
kde=True, bins=30, color='blue', stat='density')
    ax2.set_title(f'Benign - {feature}', fontsize=10)

    i += 2

# Adjust layout to prevent overlap
plt.tight_layout()
# Display the plots
plt.show()
```

Program output:



Splitting the data

```
X_train, X_test, y_train, y_test = train_test_split(used_data,
data['Malware'], test_size=0.2, random_state=0)

print(f'Number of used features is {X_train.shape[1]}')
```

Program output:

```
Number of used features is 75
```

Building the model

```
# Initialize the RandomForestClassifier
rfc = RandomForestClassifier(
    n_estimators=100,         # Number of trees in the forest
(100 trees)
    random_state=0,           # Seed for random number
generator to ensure reproducibility
    oob_score=True,           # Enable out-of-bag scoring to
assess model performance
    max_depth=16              # Maximum depth of each tree
(helps control overfitting)
)

# Fit the model on the training data
rfc.fit(X_train, y_train)     # Train the Random Forest model
using the training data

# Make predictions on the test set
y_pred = rfc.predict(X_test)  # Predict the class labels for
the test data
```

Classification report

```
print(classification_report(y_test, y_pred,
target_names=['Benign', 'Malware']))
```

Program output:

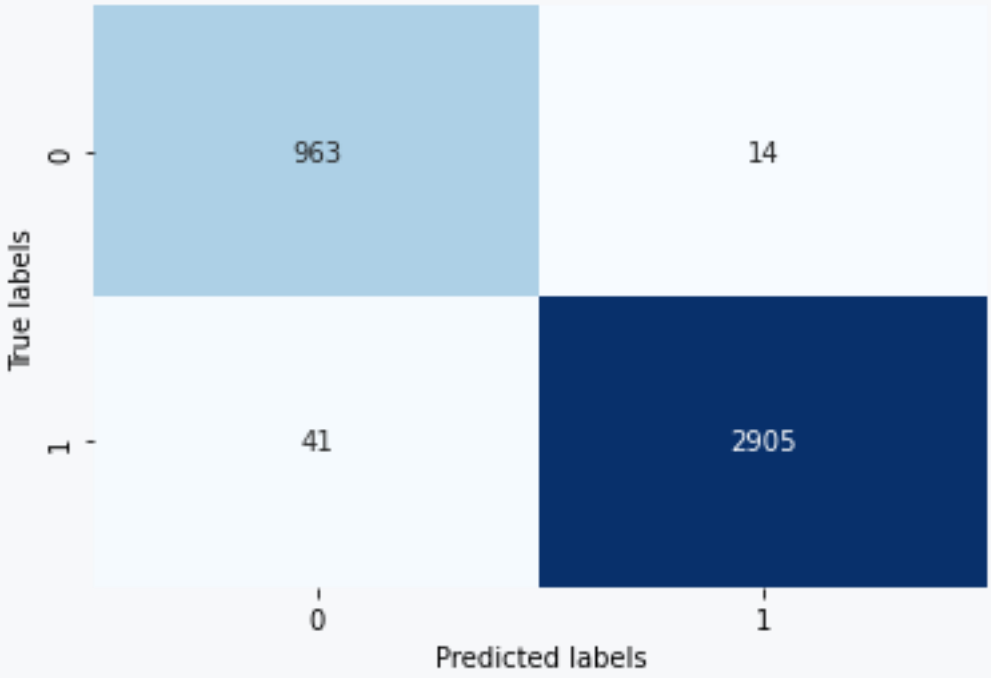
```
              precision    recall  f1-score   support

      Benign       0.99      0.96      0.97      1004
     Malware       0.99      1.00      0.99      2919

    accuracy                           0.99      3923
   macro avg       0.99      0.98      0.98      3923
```

```
weighted avg       0.99      0.99      0.99      3923
```

Confusion matrix

```
ax=sns.heatmap(confusion_matrix(y_pred, y_test), annot=True,
fmt="d", cmap=plt.cm.Blues, cbar=False)
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
```

Program output:

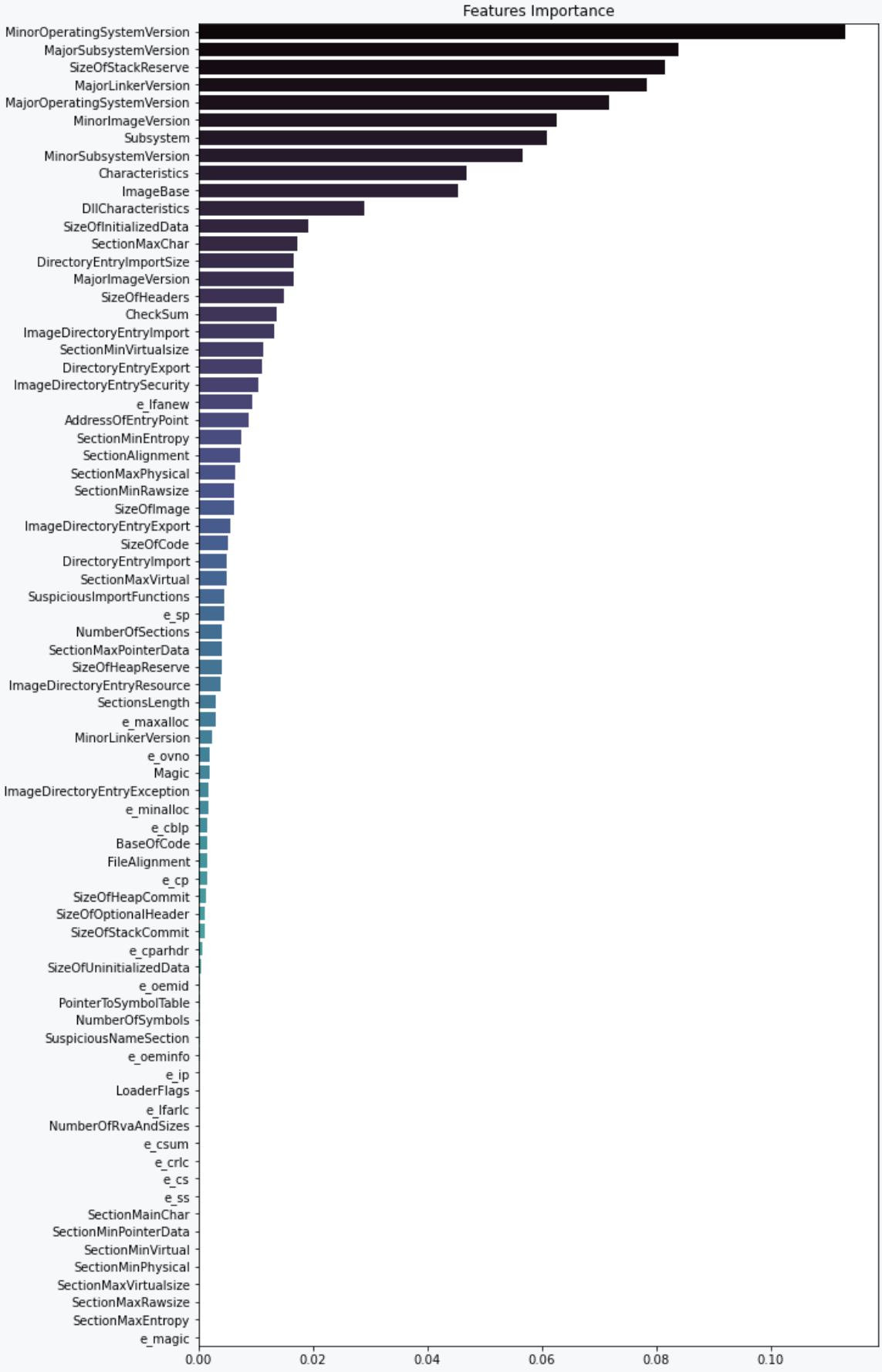


Features importance

```
# Extract feature importances from the trained Random Forest
model
importance = rfc.feature_importances_

# Create a dictionary mapping feature names to their
corresponding importance scores
importance_dict = {used_data.columns.values[i]: importance[i]
for i in range(len(importance))}

# Sort the dictionary by importance scores in ascending order
sorted_dict = {k: v for k, v in
sorted(importance_dict.items(), key=lambda item: item[1])}

# Set up the figure size for the plot
```

```
plt.figure(figsize=(10, 20))

# Create a horizontal bar plot to visualize feature importance
sns.barplot(y=list(sorted_dict.keys())[::-1],
x=list(sorted_dict.values())[::-1], palette='mako')

# Set the title for the plot
plt.title('Features Importance')
```

Program output:



7.2.2

Project: PCA, RFC, KNN

(by https://www.kaggle.com/code/singh2010nidhi/simple-machine-learning-antimalware)

Dataset

- original: https://www.kaggle.com/datasets/amauricio/pe-files-malwares
- local: https://priscilla.fitped.eu/data/cybersecurity/malware/dataset_malwares.csv

Build a simple machine learning based anti-malware system using the Benign & Malicious PE Files dataset.

The dataset contains Portable Executable (PE) files divided into two categories:

- Malicious PE Files: These files contain malware originating from VirusShare.
- Benign PE files: These are clean Windows server operating system files with no malicious content.

The dataset is created using the Python pefile library, which allows the extraction of various PE file properties such as headers, sections, and version information. These features will serve as features for the machine learning model.

```
#Import the libraries
import numpy as np
import pandas as pd
import seaborn as sns
import pickle as pck
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
%matplotlib inline
```

```
#Loading dataset from training
data =
pd.read_csv('https://priscilla.fitped.eu/data/cybersecurity/malware/dataset_malwares.csv', sep=',')

# Initialize a StandardScaler object to normalize the data
scaler = StandardScaler()
```

```
# Fit the scaler to the training data and transform it,
scaling each feature to have a mean of 0 and standard
deviation of 1
X_scaled = scaler.fit_transform(X_train)

# Create a new DataFrame for the scaled data, using the same
column names as the original data
X_new = pd.DataFrame(X_scaled, columns=X.columns)

# Display the first few rows of the scaled DataFrame to verify
the transformation
print(X_new.head())
```

Program output:

```
   e_magic    e_cblp      e_cp    e_crlc  e_cparhdr
e_minalloc  e_maxalloc  \
0      0.0 -0.038591 -0.050297 -0.041557  -0.040212   -
0.042419    0.148298
1      0.0 -0.038591 -0.050297 -0.041557  -0.040212   -
0.042419    0.148298
2      0.0 -0.038591 -0.050297 -0.041557  -0.040212   -
0.042419    0.148298
3      0.0 -0.038591 -0.050297 -0.041557  -0.040212   -
0.042419    0.148298
4      0.0 -0.038591 -0.050297 -0.041557  -0.040212   -
0.042419    0.148298

       e_ss      e_sp    e_csum  ...  SectionMaxChar
SectionMainChar  \
0 -0.016139 -0.036843 -0.031918  ...        1.076024
0.0
1 -0.016139 -0.036843 -0.031918  ...        0.097299
0.0
2 -0.016139 -0.036843 -0.031918  ...        0.097299
0.0
3 -0.016139 -0.036843 -0.031918  ...        0.097299
0.0
4 -0.016139 -0.036843 -0.031918  ...        0.097299
0.0

   DirectoryEntryImport  DirectoryEntryImportSize
DirectoryEntryExport  \
0              1.379922                 -0.623512
-0.087645
```

```
1              -0.656755                    0.249356
-0.087645
2               1.125337                    1.886949
-0.063126
3               0.106999                    0.434744
-0.087645
4              -0.274878                   -0.113695
-0.087645

   ImageDirectoryEntryExport  ImageDirectoryEntryImport  \
0                   0.000436                   -0.000677
1                  -0.016366                   -0.059942
2                  -0.011787                   -0.056269
3                  -0.016366                   -0.075943
4                  -0.016366                   -0.038952

   ImageDirectoryEntryResource  ImageDirectoryEntryException
\
0                    -0.067061                     -0.019125
1                    -0.060538                     -0.020494
2                    -0.059451                     -0.020494
3                    -0.045862                     -0.020494
4                    -0.045862                     -0.020494

   ImageDirectoryEntrySecurity
0                    -0.040622
1                    -0.040622
2                    -0.040622
3                     5.561297
4                    -0.006233

[5 rows x 77 columns]
```

Following code performs dimensionality reduction with PCA, retaining 55 principal components from the original dataset. The explained_variance_ratio_ shows the proportion of variance captured by each component, and the cumulative sum indicates the total variance retained across all 55 components.

```
# Initialize a PCA (Principal Component Analysis) object with
the number of components set to 55.
# PCA is used here to reduce the dimensionality of the data
while retaining as much variance as possible.
skpca = PCA(n_components=55)
```

```
# Fit the PCA model to the scaled data (X_new) and transform
it, reducing it to the top 55 principal components.
X_pca = skpca.fit_transform(X_new)

# Print the cumulative variance explained by the 55 components
to understand how much of the data's variance is retained.
print('Variance sum : ',
skpca.explained_variance_ratio_.cumsum()[-1])
```

Program output:

```
Variance sum :  0.9872673777501171
```

```
# Import the RandomForestClassifier class as RFC from scikit-
learn.
from sklearn.ensemble import RandomForestClassifier as RFC
# Import metrics for evaluation: classification_report and
confusion_matrix.
from sklearn.metrics import classification_report,
confusion_matrix

# Initialize the Random Forest Classifier with specific
hyperparameters:
# - n_estimators=100: The number of trees in the forest.
# - random_state=0: A fixed seed for reproducibility.
# - oob_score=True: Enables the out-of-bag error estimate.
# - max_depth=16: Limits the maximum depth of each tree,
controlling overfitting.
# - max_features='sqrt': Restricts the number of features
considered at each split to the square root of total features.
model = RFC(n_estimators=100, random_state=0,
                          oob_score = True,
                          max_depth = 16,
                          max_features = 'sqrt')

# Fit the model to the training data (X_pca, y_train), which
has been reduced in dimensions using PCA.
model.fit(X_pca, y_train)

# Scale the test data using the same scaler used on the
training set to ensure consistency.
X_test_scaled = scaler.transform(X_test)

# Convert scaled test data into a DataFrame for better column
management.
```

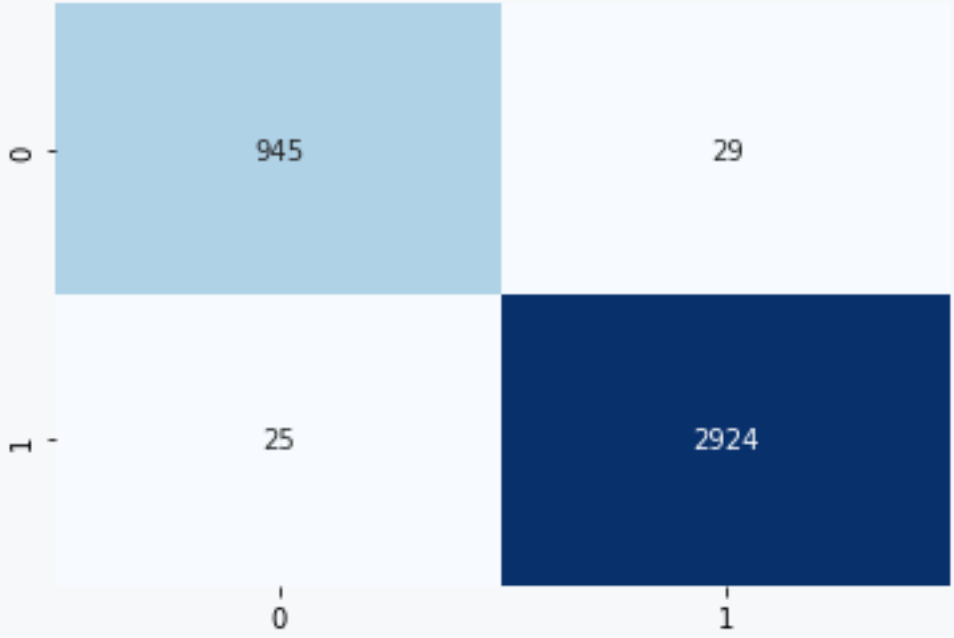
```
X_test_new = pd.DataFrame(X_test_scaled, columns=X.columns)

# Apply PCA transformation on the scaled test data using the
previously fitted PCA model.
X_test_pca = skpca.transform(X_test_new)

# Use the trained Random Forest model to predict on the PCA-
transformed test data.
y_pred = model.predict(X_test_pca)

# Print the classification report, which provides precision,
recall, f1-score, and support for each class.
print(classification_report(y_test, y_pred))
```

Program output:

```
              precision    recall  f1-score   support

           0       0.97      0.97      0.97       974
           1       0.99      0.99      0.99      2949

    accuracy                           0.99      3923
   macro avg       0.98      0.98      0.98      3923
weighted avg       0.99      0.99      0.99      3923
```

```
sns.heatmap(confusion_matrix(y_test,y_pred), annot=True,
fmt="d", cmap=plt.cm.Blues, cbar=False)
```

Program output:



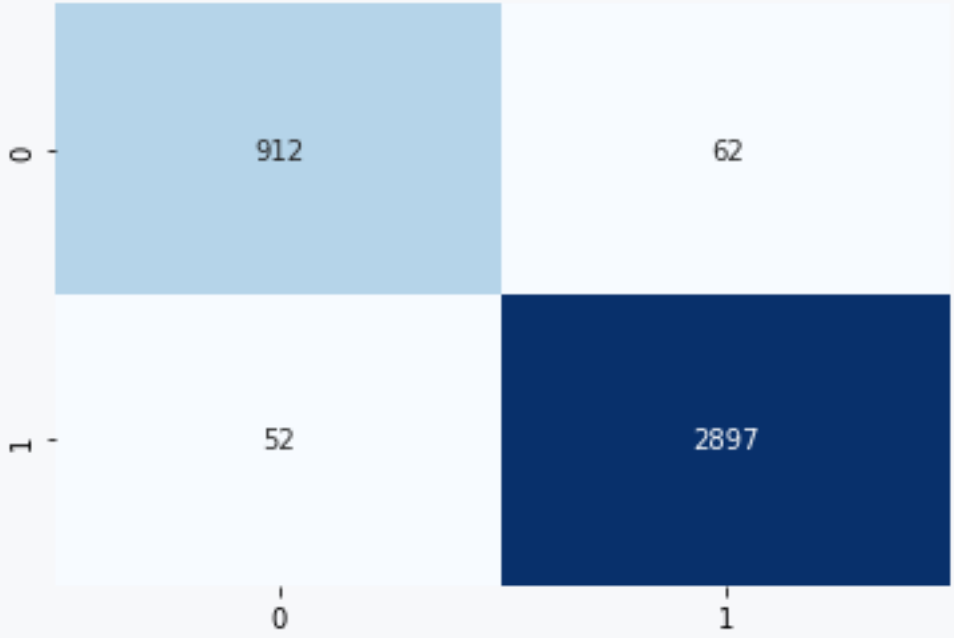
SVC

```
# Import the Support Vector Classifier (SVC) from scikit-
learn.
from sklearn.svm import SVC

# Initialize the Support Vector Classifier with default
hyperparameters.
model = SVC()

# Train (fit) the SVC model on the PCA-transformed training
data (X_pca) and corresponding labels (y_train).
model.fit(X_pca, y_train)
predictions = model.predict(X_test_pca)
print(classification_report(y_test,predictions))
```

Program output:

```
              precision    recall  f1-score   support

           0       0.95      0.94      0.94       974
           1       0.98      0.98      0.98      2949

    accuracy                           0.97      3923
   macro avg       0.96      0.96      0.96      3923
weighted avg       0.97      0.97      0.97      3923
sns.heatmap(confusion_matrix(y_test,predictions), annot=True,
fmt="d", cmap=plt.cm.Blues, cbar=False)
```

Program output:




```
from sklearn.model_selection import GridSearchCV

# Define the hyperparameters to search over for optimization
param_grid = {
    'C': [0.1, 1, 10, 100, 1000],      # Regularization
parameter
    'gamma': [1, 0.1, 0.01, 0.001, 0.0001]  # Kernel
coefficient for 'rbf', 'poly', and 'sigmoid'
}

# Initialize GridSearchCV with SVC and the parameter grid
# verbose=3 enables more detailed output during the search
process
grid = GridSearchCV(SVC(), param_grid, verbose=3)

# Fit the grid search on the training data to find the best
parameter combination
grid.fit(X_pca, y_train)
```

Program output:

```
[CV 5/5] END ..............C=1000, gamma=0.0001;, score=0.964
total time=   1.6s
```

```
print(grid.best_params_)
print(grid.best_estimator_)
```

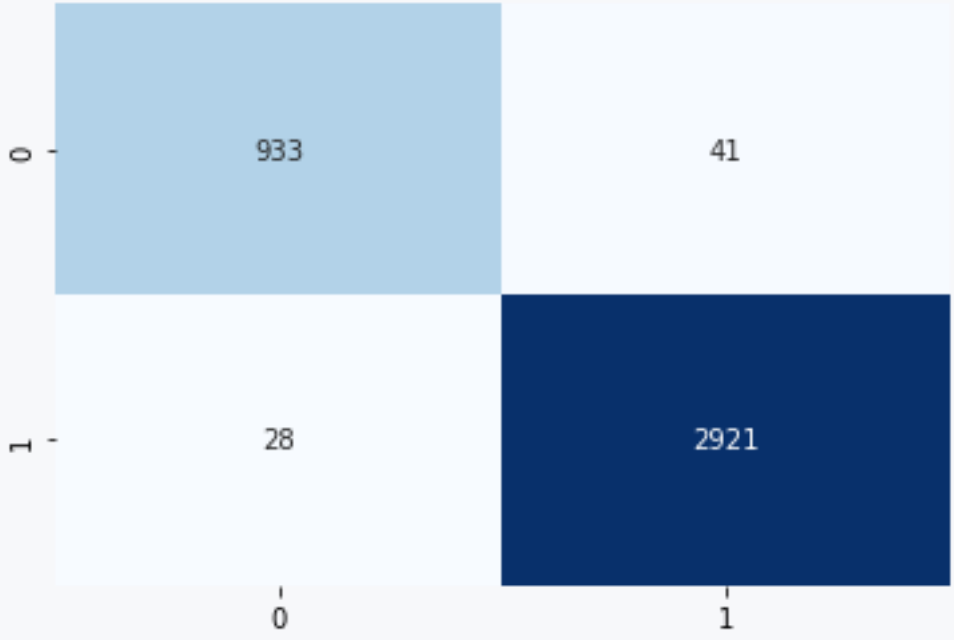
```
grid_predictions = grid.predict(X_test_pca)
print(classification_report(y_test,grid_predictions))
```

Program output:

```
{'C': 100, 'gamma': 0.1}
SVC(C=100, gamma=0.1)
              precision    recall  f1-score   support

           0       0.97      0.96      0.96       974
           1       0.99      0.99      0.99      2949

    accuracy                           0.98      3923
   macro avg       0.98      0.97      0.98      3923
weighted avg       0.98      0.98      0.98      3923
```

```
sns.heatmap(confusion_matrix(y_test,grid_predictions),
annot=True, fmt="d", cmap=plt.cm.Blues, cbar=False)
```

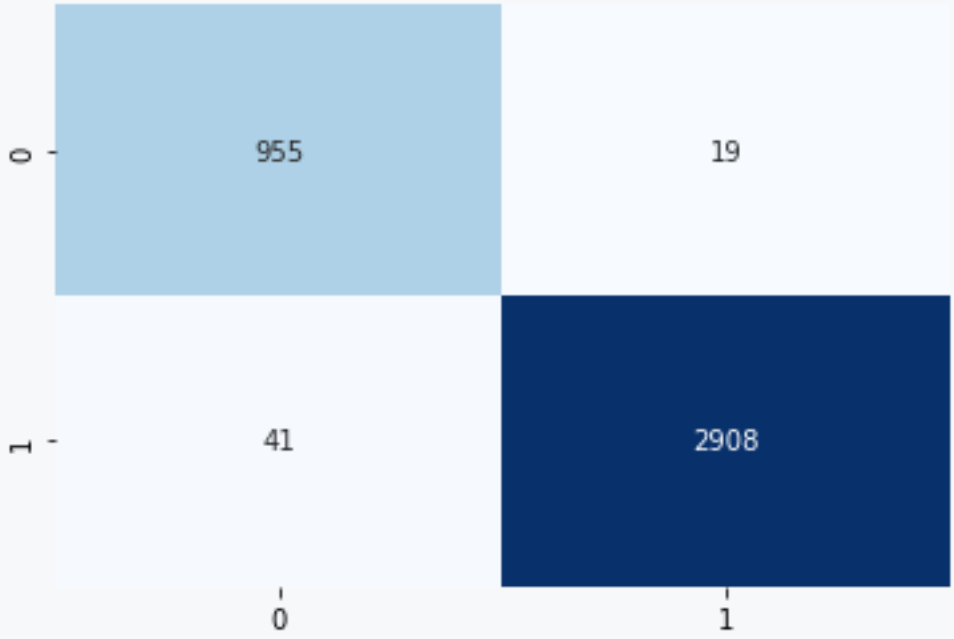
Program output:




```
from sklearn.neighbors import KNeighborsClassifier

# Initialize the KNeighborsClassifier with 1 neighbor
knn = KNeighborsClassifier(n_neighbors=1)

# Fit the model on the PCA-transformed training data
```

```
knn.fit(X_pca, y_train)
pred = knn.predict(X_test_pca)
sns.heatmap(confusion_matrix(y_test,pred), annot=True,
fmt="d", cmap=plt.cm.Blues, cbar=False)
```

Program output:




```
print(classification_report(y_test,pred))
```

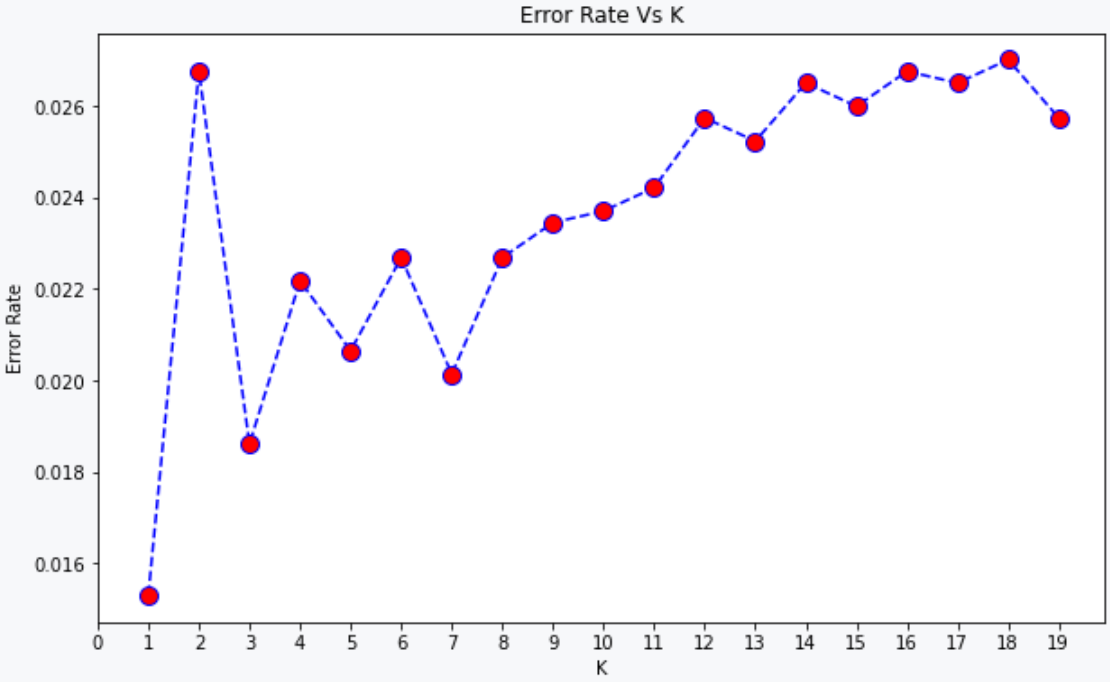
Program output:

```
              precision    recall  f1-score   support

           0       0.96      0.98      0.97       974
           1       0.99      0.99      0.99      2949

    accuracy                           0.98      3923
   macro avg       0.98      0.98      0.98      3923
weighted avg       0.98      0.98      0.98      3923
```

```
error_rate = []
for i in range(1,20):
    knn = KNeighborsClassifier(n_neighbors=i)
    knn.fit(X_pca,y_train)
    pred_i = knn.predict(X_test_pca)
    error_rate.append(np.mean(pred_i != y_test))
```

```
plt.figure(figsize=(10,6))
plt.plot(range(1,20),error_rate,color='blue',linestyle="--
",marker="o", markerfacecolor='red',markersize=10)
plt.title("Error Rate Vs K")
plt.xlabel("K")
plt.xticks(np.arange(0, 20, step=1))
plt.ylabel("Error Rate")
```

Program output:



k=1 is the optimum value of k

7.3 Benefits and advantages

🕮 7.3.1

Integrating machine learning into malware detection systems provides numerous benefits that enhance security measures and improve operational efficiency. These advantages arise from the powerful capabilities of machine learning algorithms, which can analyze large volumes of data, adapt to emerging threats, and automate the detection process. The following section highlights the key benefits of using machine learning in malware detection:

- **High Accuracy**: Machine learning algorithms can effectively identify malware by learning from a vast dataset of both malicious and benign files. By training on these labeled examples, these algorithms reduce the occurrence of false positives - instances where benign files are incorrectly flagged as

harmful. This means that they not only enhance detection accuracy but also help ensure that genuine malware is prevented from entering the system.

- **Automation**: One of the significant advantages of machine learning in malware detection is its ability to automate the identification process. This automation saves valuable time and resources for security professionals, allowing them to focus on more complex tasks. This feature is particularly beneficial in large systems that experience high traffic volumes and face multiple potential threats, as it enables continuous monitoring without overwhelming human analysts.
- **Adaptability**: Machine learning algorithms are designed to adapt to new threats and learn from past experiences. By continuously updating and retraining these models with new data, they can identify novel malware that may not have been previously encountered. This adaptability is crucial in a constantly evolving cybersecurity landscape, where new types of malware emerge regularly

7.3.2

Which of the following statements is true regarding high accuracy in machine learning algorithms?

- It prevents false positives.
- It only detects known threats.
- It is irrelevant to malware detection.
- It requires no training data.

7.3.3

Which of the following are advantages of automation in malware detection?

- Saves time and resources.
- Allows continuous monitoring of systems.
- Increases the workload of security experts.
- Limits detection to known malware only.

7.3.4

Machine learning can significantly enhance the speed, accuracy, adaptability, and scalability of malware identification processes. These improvements are crucial for preventing malware infections and addressing various security challenges. The integration of machine learning techniques allows security systems to analyze large volumes of data efficiently, adapt to emerging threats, and automate detection mechanisms. This proactive approach not only strengthens cybersecurity but also ensures organizations can respond swiftly to new vulnerabilities.

Several common machine learning techniques contribute to improved accuracy and speed in detecting malware. **Feature extraction** is one such technique, where algorithms efficiently identify and extract relevant characteristics related to malware,

including file size, type, and behavior. By thoroughly analyzing these attributes, machine learning algorithms can recognize patterns and trends associated with malware, thus enhancing both detection accuracy and speed. Another technique is **pattern recognition**, which focuses on identifying trends in malware behavior that may go unnoticed by human analysts. By scrutinizing extensive datasets, these algorithms can detect specific file types, unusual network traffic patterns, and behavioral anomalies, allowing for quicker and more precise identification of potential threats.

Moreover, machine learning systems excel in **learning from experience**, continuously improving their detection capabilities by identifying patterns within vast datasets. This ongoing learning process enables algorithms to discern subtle trends in malware behavior that human analysts might miss, thereby increasing the system's accuracy and responsiveness. **Advanced analysis** techniques facilitate the rapid examination of large data sets, allowing algorithms to identify and respond to threats in real time, which is essential for preventing security incidents. Finally, the **automation** capabilities of machine learning algorithms can significantly reduce the workload on security experts by automating the malware detection process. This leads to swift analysis and rapid identification of threats, ultimately enhancing an organization’s overall security posture. Through the use of feature extraction, pattern recognition, learning from experience, real-time analysis, and automation, machine learning algorithms can effectively identify threats, preventing malware infections and other security events.

7.3.5

Which of the following techniques is used to enhance malware detection by identifying relevant characteristics such as size and behavior?

- Feature extraction
- Pattern recognition
- Random sampling
- Mutual analysis

7.3.6

Which of the following statements are true regarding machine learning techniques in malware detection?

- Learning from experience allows algorithms to identify patterns human analysts might miss.
- Advanced analysis enables real-time detection of malware.
- Automation reduces the workload on security experts.
- Pattern recognition only works with labeled data.

Access Attacks Detection

Chapter 8

8.1 Network traffic analysis

📖 8.1.1

In our interconnected digital landscape, safeguarding network security is critical for individuals and organizations alike, as cyber threats continuously evolve in sophistication. Network traffic analysis plays a key role in this defense, as it involves monitoring data flows within a network to detect unusual behavior. By closely examining data packets moving through the network, administrators can identify potential security breaches, operational issues, or other irregularities that signal cyber threats.

Traditional network security methods, such as firewalls and signature-based intrusion detection, rely on established rules to detect suspicious activities. While these methods have proven useful, they often fall short in identifying complex or evolving threats. They can also generate false positives or miss subtle indicators of compromise, resulting in a security gap that could leave networks vulnerable.

As network environments grow more complex with cloud computing, Internet of Things (IoT) devices, and remote access requirements, the need for advanced traffic analysis increases. Modern cyber threats can bypass conventional defenses by exploiting specific vulnerabilities in network protocols or targeting applications and users directly. Thus, a proactive, adaptive approach to traffic analysis is essential in today's cybersecurity landscape.

📝 8.1.2

What is the primary purpose of network traffic analysis in cybersecurity?

- Detecting abnormal network behavior
- Improving website load speed
- Deleting unused files
- Simplifying network infrastructure

📝 8.1.3

Which are benefits of network traffic analysis in modern cybersecurity?

- Detecting suspicious behavior
- Preventing operational issues
- Reducing email spam
- Automatically deleting viruses

📖 8.1.4

The integration of Artificial Intelligence (AI) in network traffic analysis has revolutionized cybersecurity by enhancing the ability to detect anomalies and

potential intrusions. Unlike traditional, rule-based methods, AI-powered systems use machine learning algorithms to adapt and improve detection over time. By analyzing patterns in network data, these systems can identify deviations from normal behavior that might indicate malicious activity. This adaptive nature makes AI tools more effective in detecting new or evolving cyber threats.

AI-based systems work by training on historical data, learning what "normal" network behavior looks like, and then recognizing deviations. This approach reduces reliance on static rules, which can become outdated as new threats emerge. AI systems can also reduce false positives, focusing alerts on genuine threats and giving network administrators more accurate information to work with.

The AI-driven approach represents a critical shift in cybersecurity, allowing network monitoring to be proactive rather than reactive. With the flexibility to analyze large volumes of data quickly and recognize subtle changes, AI provides a more dynamic solution to network security, enabling organizations to detect and respond to attacks more effectively.

8.1.5

How does AI improve network traffic analysis over traditional methods?

- By adapting to changing network conditions
- By creating new network protocols
- By storing user passwords
- By increasing the speed of data transfers

8.1.6

AI-driven network traffic analysis involves several essential steps to ensure accurate detection and response. The first step, **Data Collection**, involves gathering network traffic data from multiple sources, including routers, firewalls, and intrusion detection systems. This data, which may include packet headers and flow records, forms the raw input that AI algorithms will analyze.

Once collected, the data undergoes **Preprocessing** to ensure it's suitable for analysis. Preprocessing tasks involve extracting relevant features, normalizing data, and decoding protocols, which helps highlight important patterns in network activity. **Feature Extraction** follows, in which AI algorithms identify meaningful features, such as packet size, inter-packet arrival times, and communication patterns, that will help the model differentiate normal from suspicious activity.

Finally, **Model Training** and **Anomaly Detection** enable the AI system to learn from labeled datasets, distinguishing between normal and malicious network behavior. Once trained, the AI model can detect anomalies in real time, flagging unusual patterns as potential security threats. When such an anomaly is detected, an **Alert** is generated to notify network administrators, allowing for a rapid and informed response.

8.1.7

Which of the following are steps involved in AI-driven network traffic analysis?

- Data Collection
- Preprocessing
- Feature Extraction
- Network layout design

8.2 Benefits and chalenges

8.2.1

The integration of artificial intelligence (AI) into network traffic analysis brings several notable benefits that enhance cybersecurity measures.

- **Improved Accuracy:** AI algorithms excel at detecting subtle patterns that may indicate malicious activities. By learning from vast datasets, these algorithms can identify previously unseen threats, leading to a reduction in false positives. This enhanced detection accuracy allows security teams to focus on genuine threats rather than sifting through numerous alerts.
- **Real-Time Monitoring:** One of the key advantages of AI-powered systems is their capability for real-time analysis of network traffic. This allows for the swift identification of threats as they arise, enabling immediate responses to mitigate potential damage. The ability to act quickly is crucial in minimizing the impact of cyberattacks.
- **Scalability:** AI models are designed to handle large volumes of network traffic, making them highly suitable for enterprise-scale deployments. This scalability ensures that organizations can maintain robust security measures even as their network environments grow and become more complex.
- **Adaptability:** AI algorithms can continuously learn and adapt to evolving threats and changing network conditions without the need for manual updates. This adaptability is vital for maintaining ongoing protection against new cyber risks that may emerge over time.
- **Enhanced Threat Intelligence:** By analyzing historical data and identifying attack patterns, AI-driven network traffic analysis contributes valuable insights for threat intelligence. These insights can inform proactive defense strategies, helping organizations to stay ahead of potential threats.

8.2.2

What is the key advantage of AI in network traffic analysis?

- Improved detection accuracy
- Simplified network design
- Reduced user access
- Slower response times

8.2.3

Which of the following are benefits of AI-driven network traffic analysis?

- Scalability
- Real-time monitoring
- Increased manual intervention
- Higher false positive rates

8.2.4

While the integration of AI in network traffic analysis holds significant promise for enhancing cybersecurity, it also presents several challenges and considerations that must be addressed to ensure its effective implementation.

- **Data Privacy:** One of the foremost concerns with analyzing network traffic is the potential infringement on user privacy. When monitoring communications or sensitive information, organizations must tread carefully to ensure compliance with data privacy regulations. This involves implementing strict data handling practices and ensuring that personal data is protected during analysis. Balancing the need for security with respect for individual privacy is essential to maintaining user trust.
- **False Positives:** Although AI algorithms have made strides in reducing false positives compared to traditional security methods, they are not entirely free from errors. False alarms can lead to unnecessary alerts, causing security teams to waste valuable resources investigating benign activities. To enhance the effectiveness of AI-driven systems, continuous fine-tuning and validation of the algorithms are necessary. This iterative process helps minimize false positives and improves overall detection accuracy.
- **Adversarial Attacks:** Malicious actors are continually developing sophisticated tactics to evade detection, including crafting attacks specifically designed to confuse AI models. As a result, it is critical to implement robust testing and adversarial training techniques that bolster the resilience of AI-driven security systems. By preparing for potential attacks that exploit AI vulnerabilities, organizations can enhance their defenses against evolving threats.
- **Interpretability:** The interpretability of AI models is crucial for effective threat response and decision-making. Security analysts must understand how AI systems arrive at their conclusions to trust their recommendations. Therefore, employing transparent and interpretable AI techniques is vital for fostering confidence in the analysis results. Enhancing the explainability of AI decisions can lead to more informed security practices and better incident management.

8.2.5

What is a significant challenge associated with AI-driven network traffic analysis?

- Data privacy concerns
- Faster data processing
- Increased network speeds
- Reduced need for monitoring

8.2.6

Which of the following are challenges faced by AI in network traffic analysis?

- False positives
- Adversarial attacks
- Interpretability of AI models
- Data security improvements

8.3 Projects

8.3.1

Project: ML Classification of network traffic

(by https://github.com/sinanw/ml-classification-malicious-network-traffic/tree/main/data)

Dataset:

- original: https://github.com/sinanw/ml-classification-malicious-network-traffic/tree/main/data
- local: https://priscilla.fitped.eu/data/cybersecurity/network/network2_dataset.csv

This project focuses on analyzing and classifying a real network traffic dataset to identify and differentiate between malicious and benign traffic records. The goal is to compare and fine-tune various machine learning algorithms, ensuring the highest possible accuracy while minimizing false positive and false negative rates. By achieving these objectives, the project aims to contribute valuable insights into network security management and intrusion detection systems.

Dataset Overview

The dataset employed in this analysis is the CTU-IoT-Malware-Capture-34-1, which is part of the Aposemat IoT-23 dataset. This labeled dataset consists of both malicious

and benign IoT network traffic, making it ideal for supervised learning tasks in the realm of cybersecurity. The dataset was developed within the Avast AIC laboratory and is supported by funding from Avast Software, highlighting its relevance and applicability in real-world scenarios.

Key features of the dataset

The dataset contains comprehensive records of network traffic, which can be utilized for various machine learning tasks, including:

- Network Intrusion Detection: Identifying unauthorized access attempts or anomalies within the network.
- Traffic Classification: Differentiating between various types of network traffic, including benign and malicious activities.
- Anomaly Detection: Recognizing patterns that deviate from expected behavior, which could indicate security threats.

The dataset includes features such as source and destination IP addresses, timestamps, protocols, and other metrics that allow for in-depth analysis of network behavior.

1. Initial data cleaning

```
# Import necessary libraries and modules
import pandas as pd
import numpy as np

# Column names are included in the file in a commented line,
so we need to read the corresponding line separately and
remove the first description word.
data_columns =
pd.read_csv('https://priscilla.fitped.eu/data/cybersecurity/ne
twork/network2_dataset.csv', sep='\t', skiprows=6, nrows=1,
header=None).iloc[0][1:]

# Read the actual dataset
data_df =
pd.read_csv('https://priscilla.fitped.eu/data/cybersecurity/ne
twork/network2_dataset.csv', sep='\t', comment="#",
header=None)

# Set column names
data_df.columns = data_columns

# Check dataset shape
print(data_df.shape)
```

```
# Check dataset head
print(data_df.head())
```

Program output:

```
(23145, 21)
0            ts                 uid      id.orig_h  id.orig_p
id.resp_h  \
0  1.545404e+09   CrDn63WjJEmrWGjqf  192.168.1.195      41040
185.244.25.235
1  1.545404e+09  CY9lJW3gh1Eje4usP6  192.168.1.195      41040
185.244.25.235
2  1.545404e+09   CcFXLynukEDnUlvgl  192.168.1.195      41040
185.244.25.235
3  1.545404e+09   CDrkrSobGYxHhYfth  192.168.1.195      41040
185.244.25.235
4  1.545404e+09  CTWZQf2oJSvq6zmPAc  192.168.1.195      41042
185.244.25.235

0  id.resp_p proto service  duration orig_bytes  ...
conn_state local_orig  \
0         80   tcp       -  3.139211          0  ...
S0           -
1         80   tcp       -         -          -  ...
S0           -
2         80   tcp       -         -          -  ...
S0           -
3         80   tcp    http  1.477656        149  ...
SF           -
4         80   tcp       -  3.147116          0  ...
S0           -

0 local_resp missed_bytes      history orig_pkts
orig_ip_bytes  resp_pkts  \
0          -            0            S         3
180          0
1          -            0            S         1
60           0
2          -            0            S         1
60           0
3          -         2896  ShADadttcfF        94
5525          96
4          -            0            S         3
180          0
```

```
0  resp_ip_bytes  tunnel_parents   label   detailed-label
0              0                       -   Benign    -
1              0                       -   Benign    -
2              0                       -   Benign    -
3         139044                       -   Benign    -
4              0                       -   Benign    -

[5 rows x 21 columns]
```

There are a couple of issues that need to be fixed in this phase:

- We notice here that the last column contains several values, this is due to an unmatched delimiter in the original dataset file.
- We also notice some fields with '-', which means the field is unset according to the dataset documentation.

```
# Check dataset summary
data_df.info()
```

Program output:

```
RangeIndex: 23145 entries, 0 to 23144
Data columns (total 21 columns):
 #   Column                                    Non-Null Count
Dtype
---  ------                                    --------------
-----
 0   ts                                        23145 non-null
float64
 1   uid                                       23145 non-null
object
 2   id.orig_h                                 23145 non-null
object
 3   id.orig_p                                 23145 non-null
int64
 4   id.resp_h                                 23145 non-null
object
 5   id.resp_p                                 23145 non-null
int64
 6   proto                                     23145 non-null
object
```

```
 7   service                                         23145 non-null
object
 8   duration                                        23145 non-null
object
 9   orig_bytes                                      23145 non-null
object
 10  resp_bytes                                      23145 non-null
object
 11  conn_state                                      23145 non-null
object
 12  local_orig                                      23145 non-null
object
 13  local_resp                                      23145 non-null
object
 14  missed_bytes                                    23145 non-null
int64
 15  history                                         23145 non-null
object
 16  orig_pkts                                       23145 non-null
int64
 17  orig_ip_bytes                                   23145 non-null
int64
 18  resp_pkts                                       23145 non-null
int64
 19  resp_ip_bytes                                   23145 non-null
int64
 20  tunnel_parents   label   detailed-label  23145 non-null
object
dtypes: float64(1), int64(7), object(13)
memory usage: 3.7+ MB
```

- This summary says there are no missing values which is inaccurate (due to the unset fields with '-' values).
- Some numerical fields are misidentified as "object" (strings) which will also be fixed later.

2. Data cleaning

- The last column in the dataset contains three separate values and needs to be unpacked into three corresponding columns. This is due to unmatched separators in the original data file.

```
# Split the last combined column into three ones
tunnel_parents_column = data_df.iloc[:,-1].apply(lambda x:
x.split()[0])
```

```
label_column = data_df.iloc[:,-1].apply(lambda x:
x.split()[1])
detailed_label_column = data_df.iloc[:,-1].apply(lambda x:
x.split()[2])

# Drop the combined column
data_df.drop(["tunnel_parents   label   detailed-label"],
axis=1, inplace=True)

# Add newly created columns to the dataset
data_df["tunnel_parents"] = tunnel_parents_column
data_df["label"] = label_column
data_df["detailed_label"] = detailed_label_column

# Check the dataset
print(data_df.head())
```

Program output:

```
0            ts                 uid      id.orig_h  id.orig_p
id.resp_h  \
0  1.545404e+09   CrDn63WjJEmrWGjqf  192.168.1.195      41040
185.244.25.235
1  1.545404e+09  CY9lJW3gh1Eje4usP6  192.168.1.195      41040
185.244.25.235
2  1.545404e+09   CcFXLynukEDnUlvgl  192.168.1.195      41040
185.244.25.235
3  1.545404e+09   CDrkrSobGYxHhYfth  192.168.1.195      41040
185.244.25.235
4  1.545404e+09  CTWZQf2oJSvq6zmPAc  192.168.1.195      41042
185.244.25.235

0  id.resp_p proto service  duration orig_bytes  ...
local_resp missed_bytes  \
0         80   tcp       -  3.139211          0  ...
-            0
1         80   tcp       -         -          -  ...
-            0
2         80   tcp       -         -          -  ...
-            0
3         80   tcp    http  1.477656        149  ...
-         2896
4         80   tcp       -  3.147116          0  ...
-            0
```

```
0      history orig_pkts  orig_ip_bytes resp_pkts
resp_ip_bytes  \
0            S         3            180         0
0
1            S         1             60         0
0
2            S         1             60         0
0
3  ShADadttcfF        94           5525        96
139044
4            S         3            180         0
0

0  tunnel_parents   label  detailed_label
0               -  Benign               -
1               -  Benign               -
2               -  Benign               -
3               -  Benign               -
4               -  Benign               -

[5 rows x 23 columns]
```

- the combined column was replaced with three separate ones: "tunnel_parents", "label", and "detailed_label".

Drop irrelevant columns

- Drop extra columns that don't contribute to the data analysis and predictions (ex. ids, columns with only unique values, columns with just one value, ip addresses, ...).

```
# Check the number of unique values in each column
print(data_df.nunique().sort_values(ascending=False))
```

Program output:

```
0
ts                23145
uid               23145
duration           4654
id.orig_p          4383
orig_ip_bytes       108
resp_ip_bytes        62
orig_pkts            53
id.resp_h            49
resp_bytes           44
```

```
orig_bytes           29
resp_pkts            28
history              26
id.resp_p            10
conn_state            6
service               5
detailed_label        4
missed_bytes          3
proto                 2
id.orig_h             2
label                 2
local_resp            1
local_orig            1
tunnel_parents        1
dtype: int64
```

```
# Two columns have only unique values and three columns have
only one value, so we should drop them.
data_df.drop(columns=["ts","uid","local_resp","local_orig","tu
nnel_parents"], inplace=True)

# IP addresses might introduce bias in the predictions so it's
recommended to drop them
data_df.drop(columns=["id.orig_h","id.resp_h"], inplace=True)

# The "detailed_label" column provides more information about
the "label" column itself, so it doesn't contribute to the
data analysis.
data_df.drop(columns="detailed_label", inplace=True)

# Check the dataset
print(data_df.head())
```

Fix unset values and validate the data types

- According to the dataset documentation, empty and unset values are represented as '-' and "(empty)". This is why info() method didn't show any missing values. Fixing these values is essential to correct the types of numeric attributes that were misinterpreted as strings.

```
# Replace all occurrences of empty/unset cells with null
values
data_df.replace({'-':np.nan, "(empty)":np.nan}, inplace=True)
```

```
# Fix data types of the misinterpreted columns
dtype_convert_dict = {
    "duration": float,
    "orig_bytes": float,
    "resp_bytes": float
}
data_df = data_df.astype(dtype_convert_dict)
print(data_df.info())
```

Program output:

```
RangeIndex: 23145 entries, 0 to 23144
Data columns (total 15 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   id.orig_p       23145 non-null  int64
 1   id.resp_p       23145 non-null  int64
 2   proto           23145 non-null  object
 3   service         1847 non-null   object
 4   duration        5321 non-null   float64
 5   orig_bytes      5321 non-null   float64
 6   resp_bytes      5321 non-null   float64
 7   conn_state      23145 non-null  object
 8   missed_bytes    23145 non-null  int64
 9   history         23145 non-null  object
 10  orig_pkts       23145 non-null  int64
 11  orig_ip_bytes   23145 non-null  int64
 12  resp_pkts       23145 non-null  int64
 13  resp_ip_bytes   23145 non-null  int64
 14  label           23145 non-null  object
dtypes: float64(3), int64(7), object(5)
memory usage: 2.6+ MB
None
```

3. Data preprocessing

```
import seaborn as sns
import ipaddress
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score,
classification_report
from sklearn.impute import KNNImputer
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.preprocessing import MinMaxScaler
```

```
# Check null values in the target attribute
print(data_df["label"].isna().sum())
```

Program output:

```
0
```

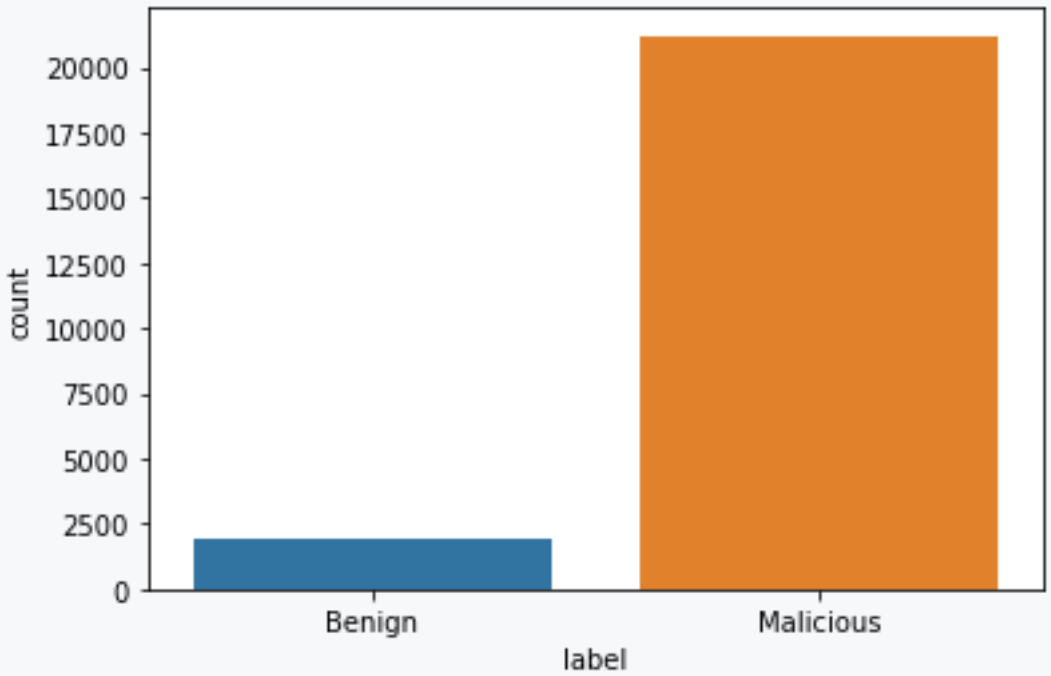
```
# Check values distribution
print(data_df["label"].value_counts())
```

Program output:

```
Malicious    21222
Benign        1923
Name: label, dtype: int64
```

```
# Plot target attribute on a count plot
sns.countplot(data=data_df, x="label")
```

Program output:



- As we can see from these statistics, the target attribute is highly imbalanced.
- We will maintain the original distribution for now until we explore the models' performance.
- Suitable techniques will be applied to re-balance the labels if we couldn't achieve an acceptable prediction accuracy.

```
# Initialize encoder with default parameters
```

```
target_le = LabelEncoder()

# Fit the encoder to the target attribute
encoded_attribute = target_le.fit_transform(data_df["label"])

# Replace target attribute with encoded values
data_df["label"] = encoded_attribute

# Check mapped labels
print(dict(zip(target_le.classes_,
target_le.transform(target_le.classes_))))
```

Program output:

```
{0: 0, 1: 1}
```

Handling outliers

```
# Use describe() method to obtain general statistics about the
numerical features
numerical_features = ["duration", "orig_bytes",
      "resp_bytes", "missed_bytes", "orig_pkts",
      "orig_ip_bytes", "resp_pkts", "resp_ip_bytes"]
print(data_df[numerical_features].describe())
```

Program output:

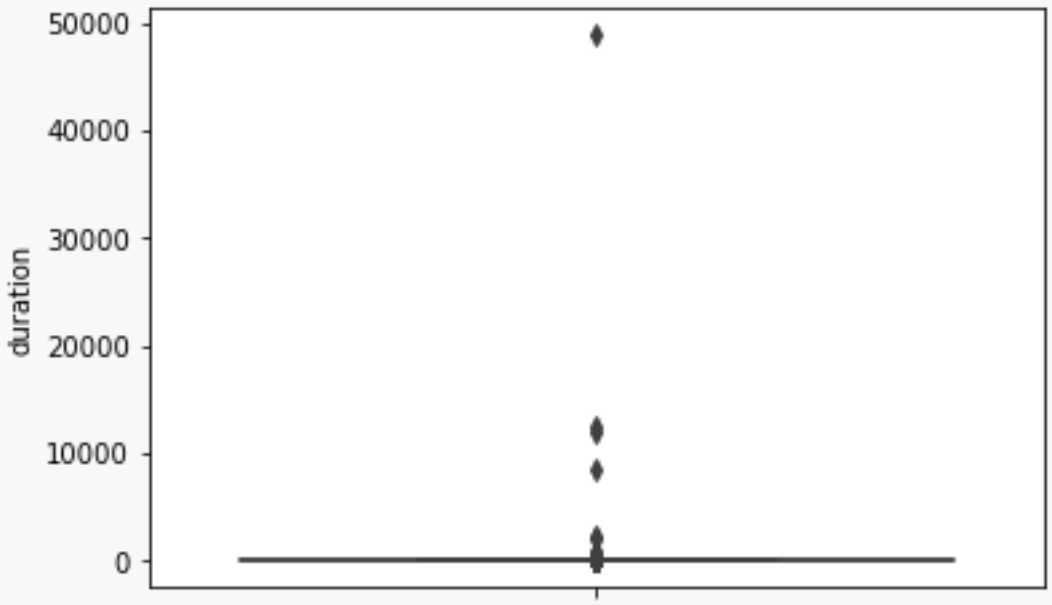
```
0           duration    orig_bytes     resp_bytes  missed_bytes
orig_pkts  \
count   5321.000000  5.321000e+03    5321.000000  23145.000000
23145.000000
mean      22.806503  1.478868e+04     350.429431      2.127112
6.375157
std      722.522302  1.036441e+06    5378.262771    102.490787
178.548725
min        0.000497  0.000000e+00       0.000000      0.000000
0.000000
25%        2.075814  0.000000e+00       0.000000      0.000000
0.000000
50%        3.110974  0.000000e+00       0.000000      0.000000
0.000000
75%        3.153695  6.200000e+01     243.000000      0.000000
1.000000
max    48976.819063  7.554662e+07  164266.000000   5792.000000
18444.000000

0      orig_ip_bytes     resp_pkts  resp_ip_bytes
```

```
count    2.314500e+04  23145.000000   23145.000000
mean     3.664312e+03      0.611017     111.218967
std      5.003762e+05      8.305898    2713.082822
min      0.000000e+00      0.000000       0.000000
25%      0.000000e+00      0.000000       0.000000
50%      0.000000e+00      0.000000       0.000000
75%      7.600000e+01      0.000000       0.000000
max      7.606306e+07   1070.000000  168910.000000
```

Upon a brief review of the statistical summary, we observed several features exhibiting values that could be classified as outliers. However, confirming these values and understanding their underlying causes is challenging without expert input from the relevant domain. For the purposes of this demonstration, we will focus specifically on the "duration" feature as having genuine outliers, as it is the only feature that presents a substantial number of unique values.

```
# Plot "duration" feature on a boxplot
sns.boxplot(data=data_df, y="duration")
```

Program output:




```
# Replace outliers using IQR (Inter-quartile Range)
outliers_columns = ['duration']
for col_name in outliers_columns:
    # Calculate first and third quartiles
    q1, q3 = np.nanpercentile(data_df[col_name],[25,75])

    # Calculate the inter-quartile range
    intr_qr = q3-q1
```

```
    # Calculate lower and higher bounds
    iqr_min_val = q1-(1.5*intr_qr)
    iqr_max_val = q3+(1.5*intr_qr)
    print(f"(min,max) bounds for \"{col_name}\":
({iqr_min_val},{iqr_max_val})")

    # Replace values that are less than min or larger then max
with np.nan
    data_df.loc[data_df[col_name] < iqr_min_val, col_name] =
np.nan
    data_df.loc[data_df[col_name] > iqr_max_val, col_name] =
np.nan

# Reevaluate the new distribution of values
print(data_df["duration"].describe())
```

Program output:

```
(min,max) bounds for "duration":
(0.4589924999999997,4.7705165)
count    3718.000000
mean        2.848130
std         0.806614
min         0.553685
25%         3.085004
50%         3.110717
75%         3.140398
max         4.723553
Name: duration, dtype: float64
```

```
# Check the number of null values in each column
print(data_df.isnull().sum().sort_values(ascending=False))
```

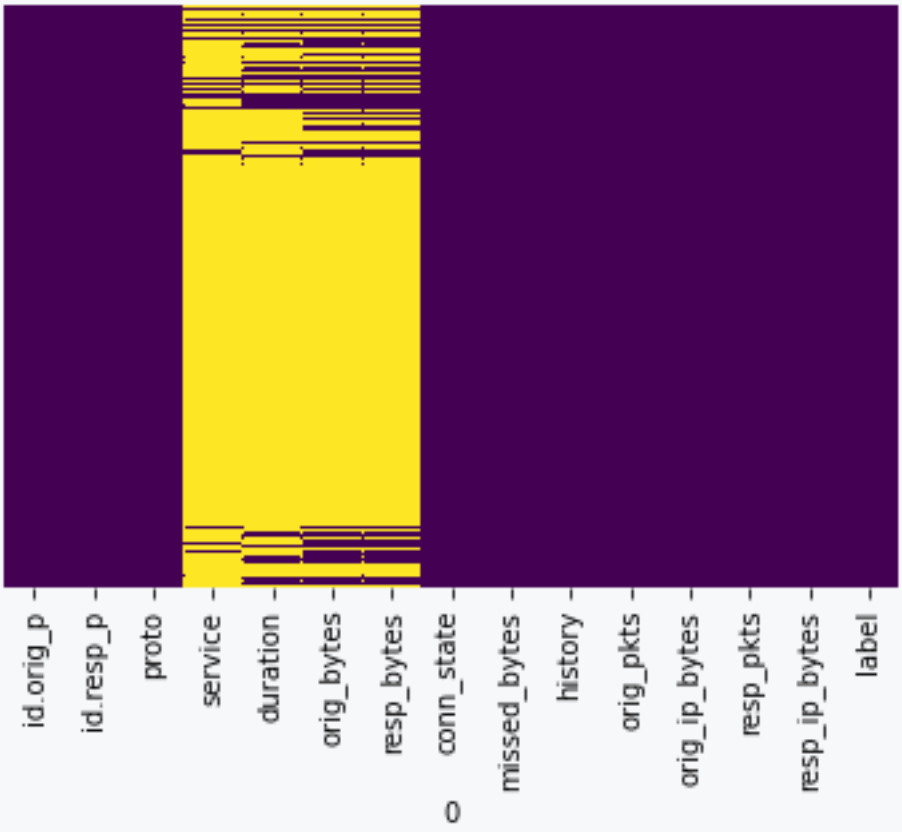
Program output:

```
0
service          21298
duration         19427
orig_bytes       17824
resp_bytes       17824
id.orig_p            0
id.resp_p            0
proto                0
conn_state           0
missed_bytes         0
history              0
```

```
orig_pkts           0
orig_ip_bytes       0
resp_pkts           0
resp_ip_bytes       0
label               0
dtype: int64
```

```
# Check null values using heatmap
sns.heatmap(data=data_df.isnull(), yticklabels=False,
cbar=False, cmap="viridis")
```

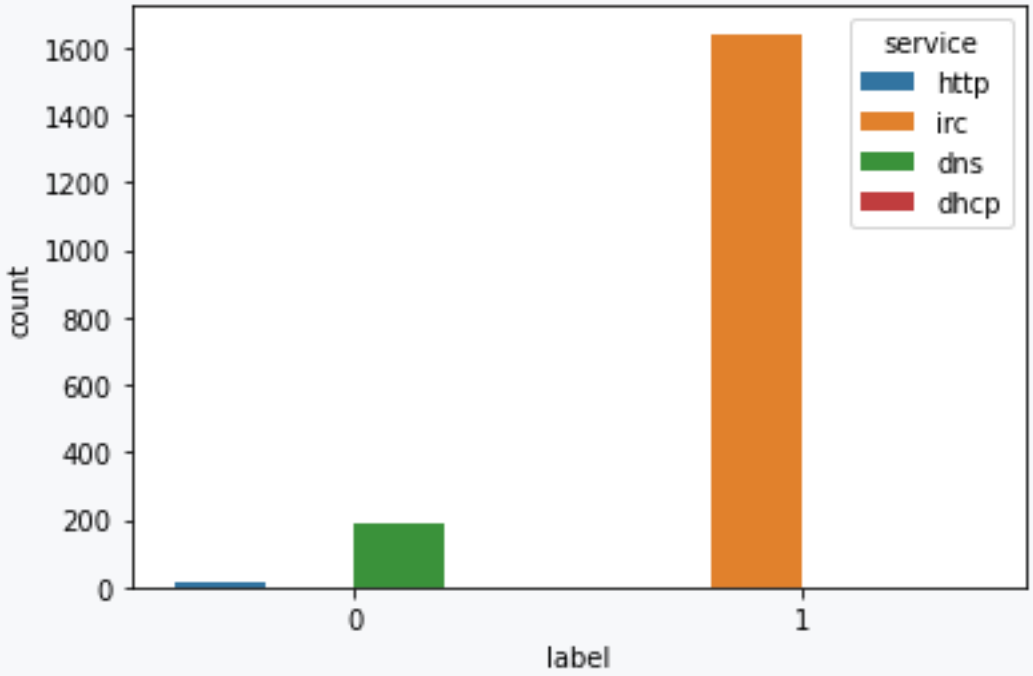
Program output:



There are four columns in the dataset that contain a significant number of missing values: one categorical column ("service") and three numerical columns ("duration," "orig_bytes," and "resp_bytes"). A common approach might be to remove these columns entirely from the dataset; however, we will take a different route. To retain any potentially valuable information that may be hidden within these features, we will attempt to impute their missing values

```
# Check the relationship between the "service" and the target
attribute
sns.countplot(data=data_df, x="label", hue="service")
```

Program output:



From the visualization, it is evident that nearly all malicious observations are associated with a particular service type, specifically "irc." In contrast, the other three service types predominantly represent benign samples. This observation highlights that, despite the high number of missing values in the "service" feature, it maintains a strong correlation with the target attribute, which is crucial for our analysis. Given its significance, we have decided to retain the "service" attribute in the dataset.

To address the missing values in this feature, we will utilize a classifier for imputation. By leveraging the relationships between the "service" feature and other available features, we can better estimate the missing values. This approach not only preserves the valuable information contained in the "service" attribute but also enhances the dataset's overall predictive power, ultimately contributing to improved performance in our machine learning models.

```
# Select specific columns to be used for the classification,
here we initially select the numerical attributes with no
missing values.
srv_training_columns =
["id.orig_p","id.resp_p","missed_bytes","orig_pkts","orig_ip_b
ytes","resp_pkts","resp_ip_bytes"]

# Split the rows into two datasets containing rows
with/without "service"
data_df_with_service = data_df[data_df["service"].notna()]
data_df_no_service = data_df[data_df["service"].isna()]
```

```
# Split the service dataset into dependent and independent
features
srv_X = data_df_with_service[srv_training_columns]
srv_y = data_df_with_service["service"].values

# Split into train/test subsets
srv_X_train, srv_X_test, srv_y_train, srv_y_test =
train_test_split(srv_X, srv_y, test_size=0.2, random_state=0)

# Create KNN estimator and fit it
srv_knn = KNeighborsClassifier(n_neighbors=3)
srv_knn.fit(srv_X_train, srv_y_train)

# Predict missing values
srv_y_pred = srv_knn.predict(srv_X_test)

# Check predictions accuracy
srv_accuracy_test = accuracy_score(srv_y_test, srv_y_pred)
print(f"Prediction accuracy for 'service' is:
{srv_accuracy_test}")
print("Classification report:")
print(classification_report(srv_y_test, srv_y_pred))
```

Program output:

```
Prediction accuracy for 'service' is: 1.0
Classification report:
              precision    recall  f1-score   support

         dns       1.00      1.00      1.00        41
        http       1.00      1.00      1.00         5
         irc       1.00      1.00      1.00       324

    accuracy                           1.00       370
   macro avg       1.00      1.00      1.00       370
weighted avg       1.00      1.00      1.00       370
```

The classification model achieved an accuracy of 100%, which means all samples in the test subset were correctly predicted. Now we can use this model to predict missing "service" fields.

```
# Predict "service" for missing values
srv_predictions =
srv_knn.predict(data_df_no_service[srv_training_columns])
```

```
# Update the original data set with predicted "service" values
data_df.loc[data_df["service"].isna(), "service"] =
srv_predictions

# To preserve hidden correlations with other features in the
dataset, we will use a KNN imputer to estimate the missing
values based on relationships with other numerical features.
numerical_features = data_df.drop("label",
axis=1).select_dtypes(include="number").columns
knn_imputer = KNNImputer()
data_df_after_imputing =
knn_imputer.fit_transform(data_df[numerical_features])

# Update original data set to fill missing values with imputed
ones
data_df[numerical_features] = data_df_after_imputing

# Confirm all missing values were successfully imputed
print(data_df.isnull().sum().sort_values(ascending=False))
```

Program output:

```
0
id.orig_p         0
id.resp_p         0
proto             0
service           0
duration          0
orig_bytes        0
resp_bytes        0
conn_state        0
missed_bytes      0
history           0
orig_pkts         0
orig_ip_bytes     0
resp_pkts         0
resp_ip_bytes     0
label             0
dtype: int64
ValueError
Must have equal len keys and value when setting with an
iterable
```

Scaling numerical attributes

- As we aim to compare several classifiers, and since some of them rely on distance-based comparisons, we will scale the numerical features to have them represented in a unified distribution.
- Since most attributes have no normal distribution, it's more suitable to apply a normalization (between 0 and 1, using MinMaxScaler) instead of standardization.

```
# Check statistics for numerical features
numerical_features = ["id.orig_p", "id.resp_p", "duration",
"orig_bytes", "resp_bytes", "missed_bytes", "orig_pkts",
"orig_ip_bytes",   "resp_pkts", "resp_ip_bytes"]
# Initialize and apply MinMaxScaler scaler
min_max_scaler = MinMaxScaler()
data_df[numerical_features] =
min_max_scaler.fit_transform(data_df[numerical_features])

# Check statistics for scaled features
print(data_df[numerical_features].describe())
```

Program output:

```
0         id.orig_p     id.resp_p      duration    orig_bytes
resp_bytes  \
count  23145.000000  23145.000000  23145.000000  2.314500e+04
23145.000000
mean       0.864132      0.036421      0.591380  4.548098e-05
0.000533
std        0.263126      0.084062      0.086854  6.578095e-03
0.015722
min        0.000000      0.000000      0.000000  0.000000e+00
0.000000
25%        0.821428      0.001238      0.609168  6.353692e-07
0.000047
50%        1.000000      0.001238      0.609168  7.227325e-07
0.000047
75%        1.000000      0.104488      0.609168  7.227325e-07
0.000047
max        1.000000      1.000000      1.000000  1.000000e+00
1.000000

0      missed_bytes     orig_pkts  orig_ip_bytes     resp_pkts
resp_ip_bytes
count  23145.000000  23145.000000   2.314500e+04  23145.000000
23145.000000
```

```
mean        0.000367      0.000346   4.817466e-05      0.000571
0.000658
std         0.017695      0.009681   6.578439e-03      0.007763
0.016062
min         0.000000      0.000000   0.000000e+00      0.000000
0.000000
25%         0.000000      0.000000   0.000000e+00      0.000000
0.000000
50%         0.000000      0.000000   0.000000e+00      0.000000
0.000000
75%         0.000000      0.000054   9.991710e-07      0.000000
0.000000
max         1.000000      1.000000   1.000000e+00      1.000000
1.000000
```

Encoding categorical features

- Since all categorical features don't imply an ordered relationship between their values, they can be encoded using One-Hot Encoding.
- We need first to check features with rare values and map them to "other", in order to avoid sparse columns with statistically negligible impact and higher computational effect.

```
# Check the number of unique values in each feature
categorical_features =
["proto","service","conn_state","history"]
for c in categorical_features:
    print(f"Column ({c}) has ({data_df[c].nunique()}) distinct
values.")
```

Program output:

```
Column (proto) has (2) distinct values.
Column (service) has (4) distinct values.
Column (conn_state) has (6) distinct values.
Column (history) has (26) distinct values.
```

```
# Check values of "history" because it has too many unique
values.
history_val_counts = data_df["history"].value_counts()
print(history_val_counts)
```

Program output:

```
C               14252
S                5417
ShAdDaf          1477
```

```
D                  978
Dd                 836
ShAdDaft           102
ShAdfDr             48
CCCC                 6
ShADadtcfF           3
ShADadttcfF          3
ShAdDatfr            2
CCC                  2
ShADadf              2
ShDadAf              2
ShAfdtDr             2
ShADacdtfF           2
ShADadtctfF          2
ShAdDatf             1
ShADadttfF           1
ShAdD                1
ShADadtctfFR         1
ShAdDfr              1
ShAD                 1
DdAtaFf              1
ShADad               1
ShAdDa               1
Name: history, dtype: int64
```

```
# Map values to their frequencies
history_freq_map = data_df["history"].map(history_val_counts)

# Replace low frequent values in "history" with "Other" using
the corresponding frequency map
data_df["history"] = data_df["history"].mask(history_freq_map
< 10, "Other")

# Check "history" values after mapping
print(data_df["history"].value_counts())
```

Program output:

```
C           14252
S            5417
ShAdDaf      1477
D             978
Dd            836
ShAdDaft      102
ShAdfDr        48
```

```
Other          35
Name: history, dtype: int64
```

Encoding categorical features: apply one-hot encoder

```
# Initialize the encoder with its default parameters
ohe = OneHotEncoder()

# Fit the encoder to categorical features in the dataset
encoded_features =
ohe.fit_transform(data_df[categorical_features])

# Create a dataframe of encoded features
encoded_features_df = pd.DataFrame(encoded_features.toarray(),
columns=ohe.get_feature_names_out())

# Merge encoded features with the dataset and drop original
columns
data_df = pd.concat([data_df, encoded_features_df],
axis=1).drop(categorical_features, axis=1)
```

4. Model training

```
# Import necessary libraries and modules
import pandas as pd
from sklearn.model_selection import StratifiedKFold,
GridSearchCV
from sklearn.naive_bayes import ComplementNB
from sklearn.metrics import precision_score, confusion_matrix,
recall_score, accuracy_score, f1_score
from statistics import mean
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
import xgboost as xgb
from joblib import dump
```

```
# Split data into independent and dependent variables
data_X = data_df.drop("label", axis=1)
data_y = data_df["label"]
```

To compare the performance of several models, we choose a set of the most popular machine learning algorithms for classification tasks.

```
# Initialize classification models
classifiers = [
    # Since we have unbalanced labels, we use the Complement
version of Naive Bayes which is particularly suited for
imbalanced data sets.
    ("Naive Bayes", ComplementNB()),

    # We use the Decision Tree with its default parameters,
including the "Gini Impurity" to measure the quality of splits
and ccp_alpha=0 (no pruning is performed).
    ("Decision Tree", DecisionTreeClassifier()),

    # Logistic Regression model to help discovering linearity
separation in the data set.
    ("Logistic Regression", LogisticRegression()),

    # The efficient Random Forest model with a default base
estimators of 100.
    ("Random Forest", RandomForestClassifier()),

    # The classifier version of Support Vector Machine model.
    ("Support Vector Classifier", SVC()),

    # The distance-based KNN classifier with a default
n_neighbors=5.
    ("K-Nearest Neighbors", KNeighborsClassifier()),

    # The most powerful ensemble model of XGBoost with some
initially tuned hyperparameters.
    ("XGBoost", xgb.XGBClassifier(objective =
"binary:logistic", alpha = 10)),
]
```

To achieve more reliable performance metrics for each model across multiple iterations, we will implement a cross-validation technique instead of the conventional train/test split. Cross-validation allows us to evaluate the model's performance on different subsets of the data, leading to a better understanding of its generalization capabilities.

Given that we are working with imbalanced class distributions in our dataset, we will employ Stratified K-Folds cross-validation. This method ensures that each fold maintains the same proportion of both classes as the entire dataset, thereby preserving the distribution of labels in each subset.

```
# Initialize the cross-validator with 5 splits and sample
shuffling activated
skf_cv = StratifiedKFold(n_splits=5, shuffle=True,
random_state=0)
print("Model Training Started!")
# Initialize the results summary
classification_results = pd.DataFrame(index=[c[0] for c in
classifiers], columns=["Accuracy", "TN", "FP", "FN", "TP",
"Recall", "Precision", "F1"])

# Iterate over the estimators
for est_name, est_object in classifiers:

    print(f"### [{est_name}]: Processing ...")

    # Initialize the results for each classifier
    accuracy_scores = []
    confusion_matrices = []
    recall_scores = []
    precision_scores = []
    f1_scores = []

    # Initialize best model object to be saved
    models_path = "..\\models"
    best_model = None
    best_f1 = -1

    # Iterate over the obtained folds
    for train_index, test_index in skf_cv.split(data_X,
data_y):

        # Get train and test samples from the cross-validation
model
        X_train, X_test = data_X.iloc[train_index],
data_X.iloc[test_index]
        y_train, y_test = data_y.iloc[train_index],
data_y.iloc[test_index]

        # Train the model
        est_object.fit(X_train.values, y_train.values)

        # Predict the test samples
        y_pred = est_object.predict(X_test.values)
```

```
        # Calculate and register accuracy metrics
        accuracy_scores.append(accuracy_score(y_test, y_pred))
        confusion_matrices.append(confusion_matrix(y_test,
y_pred))
        recall_scores.append(recall_score(y_test, y_pred))
        precision_scores.append(precision_score(y_test,
y_pred))
        est_f1_score = f1_score(y_test, y_pred)
        f1_scores.append(est_f1_score)

        # Compare with best performing model
        if best_f1 < est_f1_score:
            best_model = est_object
            best_f1 = est_f1_score

    # Summarize the results for all folds for each classifier
    tn, fp, fn, tp = sum(confusion_matrices).ravel()
    classification_results.loc[est_name] =
[mean(accuracy_scores),tn,fp,fn,tp,mean(recall_scores),mean(pr
ecision_scores),mean(f1_scores)]

    # Save the best performing model
    if best_model:
        model_name = est_name.replace(' ', '_').replace('-',
'_').lower()
        model_file = model_name + ".pkl"
        dump(best_model, models_path + "\\" + model_file)

print("Model Training Finished!")
```

Program output:

```
Model Training Started!
### [Naive Bayes]: Processing ...
### [Decision Tree]: Processing ...
### [Logistic Regression]: Processing ...
### [Random Forest]: Processing ...
### [Support Vector Classifier]: Processing ...
### [K-Nearest Neighbors]: Processing ...
### [XGBoost]: Processing ...
Model Training Finished!
```

```
# Check the results
print(classification_results)
```

Program output:

```
                          Accuracy    TN  FP  FN     TP
Recall Precision  \
Naive Bayes               0.994772  1838  85  36  21186
0.998304  0.996004
Decision Tree             0.999914  1923   0   2  21220
0.999906       1.0
Logistic Regression       0.994772  1830  93  28  21194
0.998681  0.995631
Random Forest              0.99987  1923   0   3  21219
0.999859       1.0
Support Vector Classifier 0.995636  1824  99   2  21220
0.999906  0.995356
K-Nearest Neighbors        0.99771  1880  43  10  21212
0.999529  0.997977
XGBoost                    0.99987  1923   0   3  21219
0.999859       1.0

                                F1
Naive Bayes               0.997152
Decision Tree             0.999953
Logistic Regression       0.997154
Random Forest             0.999929
Support Vector Classifier 0.997626
K-Nearest Neighbors       0.998752
XGBoost                   0.999929
```

5. Result analysis

Overall, all the models are performing very well with very high accuracy, precision, recall, and F1 scores. The Decision Tree, Random Forest, and XGBoost models are achieving near-perfect performance.

Models evaluation:

- Naive Bayes achieved relatively good overall accuracy although the labels are not evenly distributed.
- Decision Tree delivered one of the highest prediction accuracies, benefiting from its algorithmic resilience to imbalanced labels.
- Logistic Regression also achieved good results, though it yielded a higher number of incorrect predictions, suggesting some linearity in the dataset.
- Random Forest as anticipated, demonstrated superior performance as one of the most efficient prediction methods. However, given the strong performance of the Decision Tree, there was no significant improvement noticed when using Random Forest.

- Support Vector Classifier also produced relatively good results with slightly higher False Positive rates.
- KNN model likewise performed well, with a minimal number of incorrect predictions, which can be attributed to the dataset's normalization between 0 and 1.
- XGBoost was expectedly among the best estimators since it's arguably the most powerful machine learning algorithm these days.

Overall observations:

- Remarkably accurate predictions were generated by most models, considering that the numbers of False Positives/Negatives are cumulative results from five separate iterations.
- Out of the seven estimators, four achieved relatively lower accuracy, but these could potentially be improved with further model tuning.
- Regardless of the model used, there were consistently some False Negative predictions, which might be attributed to anomalies or outliers in the original dataset.
- Lower accuracy models tend to produce errors primarily in the form of False Positives, largely because the majority of the population is labeled as "Malicious".
- Based on their performance, models can be categorized into two distinct groups with quite similar behavior: one group exhibits significantly high accuracy, including DT, RF, and XGB, while the second group shows relatively good performance, comprising NB, KNN, LogR, and SVC.

8.3.2

Project: Analyzing Network Traffic Dataset

(by https://www.kaggle.com/datasets/ravikumargattu/network-traffic-dataset)

Dataset:

- original: https://www.kaggle.com/datasets/ravikumargattu/network-traffic-dataset
- local: https://priscilla.fitped.eu/data/cybersecurity/network/network1_dataset.csv

This project involves inspecting and analyzing a network traffic dataset obtained from a Kali Machine at the University of Cincinnati. By utilizing machine learning techniques, students will explore various applications, including network intrusion detection, traffic classification, and anomaly detection.

Dataset overview

The dataset comprises **394,137 instances** captured over one hour on the evening of October 9th, 2023, using **Wireshark**. It is stored in a CSV (Comma Separated Values) format and includes seven features that provide detailed information about network traffic. The primary features are:

1. **No**: Instance number.
2. **Timestamp**: The time at which the network traffic instance was recorded.
3. **Source IP**: The IP address of the device sending the data.
4. **Destination IP**: The IP address of the device receiving the data.
5. **Protocol**: The protocol used for the network communication (e.g., TCP, UDP).
6. **Length**: The size of the network packet.
7. **Info**: Additional information related to the traffic instance.

Each instance captures essential aspects of network activity, enabling students to conduct thorough analyses related to network performance and security.

Data Exploration:

- Load the dataset into your preferred programming environment (e.g., Python with Pandas).
- Examine the first few rows of the dataset to understand its structure.
- Generate summary statistics and visualizations (e.g., histograms, bar charts) to explore the distribution of numerical features like packet length.

```
# write your code
```

Preprocessing:

- Clean the dataset by handling missing values or erroneous data entries.
- Convert the timestamp into a suitable format for analysis.
- Encode categorical features if necessary (e.g., protocols).

```
# write your code
```

Machine Learning Applications:

- **Network Intrusion Detection**: Implement a classification model to identify potentially malicious traffic. Use techniques like logistic regression, decision trees, or support vector machines.
- **Traffic Classification**: Explore clustering methods (e.g., K-means) to categorize different types of network traffic based on features such as source IP, destination IP, and protocol.
- **Anomaly Detection**: Use techniques such as isolation forests or autoencoders to detect unusual patterns in network traffic.

```
# write your code
```

Reporting:

- Document your findings and methodologies in a report.
- Include visualizations to support your analysis and highlight key insights.
- Discuss the implications of your results for network security management and performance monitoring.

```
# write your code and / or report
```

Appendix

Chapter 9

9.1 Packet filtering firewalls

9.1.1

Despite their simplicity, such packet filtering firewalls have been successful for a long period, and they remain being a key component in contemporary Next-Generation firewalls.

As an example, unix-based packet filtering firewall was chosen. Traditionally, iptables was used as a main firewall for most unix distributions. Here it is described.

Iptables is a powerful and quite flexible firewall tool using packet-filtering approach. It is employed to configure the Linux kernel's built-in packet filtering system, Netfilter. The primary purpose of iptables is to filter and manipulate network packets before they reach their destination.

Iptables employs several components, namely Tables (3 predefined), Chains (whose number is not limited), and Rules. The **tables** are the following:

- **filter**: This is the default table and is used for packet filtering. It is responsible for deciding whether to allow or deny a packet.
- **nat**: This table is used for network address translation. It is crucial for configuring source or destination address manipulation, which is often used in scenarios like setting up a NAT gateway.
- **mangle**: The mangle table is used for specialized packet modifications, e.g. changing the Time-to-Live (TTL) field or Quality of Service (QoS) settings.

Chains are predefined sets of rules that are applied to packets. The main chains are:

- INPUT: Packets destined for the local system.
- OUTPUT: Packets generated by the local system.
- FORWARD: Packets routed through the system.

Other chains may be created.

Rules define what should be done with packets that match specific criteria. They consist of matching criteria and the target action (ACCEPT, DROP, REJECT, etc.).

Basic iptables actions and corresponding commands include:

```
iptables -A  -p  --dport  -j
```

It appends a rule to the end of a chain.

Deleting Rules:

```
iptables -D
```

It deletes a rule from a chain by its rule number.

Displaying Rules:

```
iptables -L
```

Lists all rules.

Deleting Rules:

```
iptables -F ## Deletes (flushes) all rules.
iptables -F  ## Flushes rules for a specific chain.
```

Default Policy Configuration:

```
iptables -P
```

Sets the default policy for a chain.

📖 9.1.2

Saving/Restoring Rules:

IP tables are not persistent by default, so they have to be saved. Moreover, they can be deleted during restart so the corresponding commands (for defining them) should be added e.g. to a .profile file so that they are renewed after a reboot. Commands iptables-save and iptables-restore are used to save and restore a ruleset, respectively. More detailed information can be found in manual pages or the specific operating system administration guide.

Nowadays, however, iptables are not used so frequently because of some limitations. Among the iptables limitations, the fact that the iptables syntax that seems to be a bit complex for large rulesets, is often mentioned. In addition, the iptables limitations became more and more apparent as more and more networks apply dual-stack approach where both IP version 4 and version 6 network protocols are active, and both must be controlled by a firewall. Namely in IPv6 rule definition, there is a main limitation of iptables. Therefore, iptables have been gradually replaced by nftables.

Nftables consolidates the packet filtering rule syntax for controlling IPv4 and IPv6 packets (previously the task of iptables and ipv6tables) as well as Ethernet frames (that was previously done by arptables and ebtables).

Key features and concepts of nftables are listed as follows:

- Tables: Like iptables, nftables uses the concept of tables to organize rules. Tables can be specific to an address family (e.g., IPv4, IPv6) or generic, covering multiple address families.
- Chains: Chains define sequences of rules within a table. They can be applied to various packet types and serve specific purposes, such as INPUT, OUTPUT, FORWARD, or user-defined chains.
- Rules: Rules in nftables consist of matching criteria and an associated action (target). The syntax for rules is more straightforward and expressive compared to its predecessors.
- Expressions: nftables introduces the concept of expressions that allow even more fine-grained packet manipulation. Expressions provide a modular way to perform various actions on packets, such as modifying packet headers.

Additional nftables features include set and map data structures, allowing more complex matching conditions. Sets can be used to define groups of IP addresses, ports, or other elements, while maps allow for more advanced lookups and transformations. In addition, nftables allows to do more in regard to stateful packet filtering. This is achieved through the seamless integration with the kernel's connection tracking framework, providing stateful packet filtering. This makes it easier to write rules that take into account the state of established connections in filtering.

Atomic Rule Replacement is another new feature of nftables. This ensures that a set of rules is replaced in its entirety, reducing the risk of inconsistent or partial rule application. Thanks to nftables modular architecture, its filtering performance is also better comparing to iptables.

9.1.3

Basic nftables command examples

Creating a Table:

```
nft add table inet filter
```

Adding a Chain (into already existing filter table):

```
nft add chain inet filter input { type filter hook input
priority 0 ; }
```

Inserting a Rule:

```
nft add rule inet filter input tcp dport 22 accept
```

Listing Rules:

```
nft list ruleset
```

Example Rule for Set:

```
nft add rule ip filter input ip saddr { 192.168.1.2,
192.168.1.3 } drop
```

Nevertheless, because in packet filtering firewalls, most rules are fixed, opportunities for application of artificial intelligence is limited and it is not used for such tasks at present.

9.2 Bibliography and sources

🕮 9.2.1

Bibliography and sources:

1. D. Glăvan, C. Răcuciu, R. Moinescu and N. F. Antonie, Scientific Bulletin of Naval Academy, Vol. XXII 2019, pg. 134-143.
2. DJENNA, Amir, et al. Artificial intelligence-based malware detection, analysis, and mitigation. Symmetry, 2023, 15.3: 677.
3. FARUK, Md Jobair Hossain, et al. Malware detection and prevention using artificial intelligence techniques. In: 2021 IEEE international conference on big data (big data). IEEE, 2021. p. 5369-5377.
4. FRITSCH, Lothar; JABER, Aws; YAZIDI, Anis. An overview of artificial intelligence used in malware. In: Symposium of the Norwegian AI Society. Cham: Springer International Publishing, 2022. p. 41-51.
5. https://blog.logrocket.com/email-spam-detector-python-machine-learning/
6. https://blogs.blackberry.com/en/2021/05/the-role-of-artificial-intelligence-and-machine-learning-in-threat-detection
7. https://brightsec.com/blog/security-testing/
8. https://colab.research.google.com/github/ElizaLo/ML-using-Jupiter-Notebook-and-Google-Colab/blob/master/Spam%20Detection/Spam_Detection.ipynb
9. https://colab.research.google.com/github/love4684/Detection-of-Phishing-Websites-using-an-Efficient-Machine-Learning-Framework/blob/main/4_Finding_Best_Model/Finding_Best_Model.ipynb
10. https://cyberspecs.medium.com/security-implementations-at-different-layers-of-the-osi-model-426df664a766
11. https://dev.to/oluwadamisisamuel1/how-to-build-a-logistic-regression-model-a-spam-filter-tutorial-261b
12. https://dotsecurity.com/insights/blog-types-of-network-security-measures
13. https://github.com/Apaulgithub/oibsip_taskno4
14. https://github.com/cheese-hub/ddos-classification/blob/master/ddos-classification.ipynb
15. https://github.com/Kiinitix/Malware-Detection-using-Machine-learning
16. https://github.com/MakrandBhandari/Spam-Detection-using-Multinomial-Naive-Bayes-

Classifier/blob/main/Multinomial%20Naive%20Bayes%20Classifier%20-%20Spam%20Detection.ipynb
17. https://github.com/milindsoorya/Spam-Classifier-in-python
18. https://github.com/shreyagopal/Phishing-Website-Detection-by-Machine-Learning-Techniques
19. https://github.com/sinanw/ml-classification-malicious-network-traffic/tree/main/data
20. https://hpbn.co/transport-layer-security-tls/
21. https://hussnain-akbar.medium.com/understanding-and-implementing-na%C3%AFve-bayes-algorithm-for-email-spam-detection-85a14b330fc6
22. https://cheapsslsecurity.com/blog/what-is-transport-layer-security-in-cyber-security/
23. https://cheq.ai/blog/osi-model-threats-layers-pt-2/
24. https://infosecwriteups.com/using-python-for-malware-analysis-a-beginners-guide-8432377df2c4
25. https://mailchimp.com/resources/most-common-spam-filter-triggers/
26. https://mailtrap.io/blog/spam-filters/
27. https://mawgoud.medium.com/machine-learning-in-malware-detection-concept-techniques-and-use-cases-1067d99208ac
28. https://medium.com/@azimkhan8018/email-spam-detection-with-machine-learning-a-comprehensive-guide-b65c6936678b
29. https://medium.com/@Coursesteach/spam-detection-using-machine-learning-methods-dd5dbc799b6b
30. https://medium.com/@varun.tyagi83/introducing-the-spam-detection-model-with-pre-trained-llm-3eb1f8186ba1
31. https://megasisnetwork.medium.com/ai-driven-network-traffic-analysis-uncovering-anomalies-and-intrusions-e0e11056d7d1
32. https://mind-core.com/blogs/cybersecurity/5-types-of-cyber-security/
33. https://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.04162018.pdf
34. https://securitygate.io/blog/osi-model-session-layer/
35. https://securitygate.io/blog/osi-model-transport-layer/
36. https://securityintelligence.com/articles/osi-model-stopping-threats-session-layer/
37. https://shaikhshahid.com/blog/email-spam-detection-model-python/118/
38. https://stevenzych.medium.com/patterns-are-power-a-beginners-guide-to-spam-detection-6bc0e9f4d68
39. https://towardsdatascience.com/an-approach-to-detect-ddos-attack-with-a-i-15a768998cf7
40. https://towardsdatascience.com/email-spam-detection-1-2-b0e06a5c0472
41. https://towardsdatascience.com/phishing-classification-with-an-ensemble-model-d4b15919c2d7
42. https://www.acronis.com/en-eu/blog/posts/ai-email-security/
43. https://www.broadcom.com/topics/application-security
44. https://www.coursera.org/articles/ai-in-cybersecurity
45. https://www.dashlane.com/blog/benefits-ai-cybersecurity
46. https://www.f5.com/glossary/application-layer-security
47. https://www.geeksforgeeks.org/detecting-spam-emails-using-tensorflow-in-python/

48. https://www.geeksforgeeks.org/session-layer-in-osi-model/
49. https://www.indeed.com/career-advice/career-development/presentation-layer
50. https://www.iso.org/standard/20269.html
51. https://www.kaggle.com/code/agustnluengo/logistic-reg-in-phising-cybersec
52. https://www.kaggle.com/code/akashkr/phishing-url-eda-and-modelling
53. https://www.kaggle.com/code/karthikapadmanaban/malware-detection-using-random-forest
54. https://www.kaggle.com/code/kirollosashraf/phishing-email-detection-using-deep-learning/notebook
55. https://www.kaggle.com/code/maidaly/malware-detection-with-machine-learning/notebook
56. https://www.kaggle.com/code/msvasan/jsm-jamd-phishing-site-classification-pred
57. https://www.kaggle.com/code/pmeasa/phishing-ml
58. https://www.kaggle.com/code/singh2010nidhi/simple-machine-learning-antimalware
59. https://www.kaggle.com/code/yasserh/email-spam-detection-comparing-best-ml-models
60. https://www.kaggle.com/datasets/charlottehall/phishing-email-data-by-type?select=phishing_data_by_type.csv
61. https://www.kaggle.com/datasets/merahulk/phishing-dataset
62. https://www.kaggle.com/datasets/shashwatwork/phishing-dataset-for-machine-learning
63. https://www.kaggle.com/datasets/subhajournal/phishingemails
64. https://www.milindsoorya.co.uk/blog/build-a-spam-classifier-in-python
65. https://www.stickmancyber.com/cybersecurity-blog/iso-27001-controls-resolve-organisational-challenges
66. https://www.trimbox.io/blog/ai-based-spam-detection
67. https://www.tutorialspoint.com/network_security/network_security_application_layer.htm
68. Jihyeon Song, Sunoh Choi, Jungtae Kim, Kyungmin Park, Cheolhee Park, Jonghyun Kim, Ikkyun Kim, A study of the relationship of malware detection mechanisms using Artificial Intelligence, ICT Express, Volume 10, Issue 3, 2024, Pages 632-649, ISSN 2405-595, https://doi.org/10.1016/j.icte.2024.03.005
69. Matthew G. Gaber, Mohiuddin Ahmed, and Helge Janicke. 2024. Malware Detection with Artificial Intelligence: A Systematic Literature Review. ACM Comput. Surv. 56, 6, Article 148 (June 2024), 33 pages. https://doi.org/10.1145/3638552
70. Onih, Valentine. (2024). Phishing Detection Using Machine Learning: A Model Development and Integration. International Journal of Scientific and Management Research. 07. https://doi.org/10.37502/IJSMR.2024.7403
71. Patel, Ripalkumar & Mavani, Chirag & Mistry, Hirenkumar & Goswami, Amit. (2024). APPLICATION LAYER SECURITY FOR CLOUD. 30. 1193–1198.
72. PFEFFER, Avi, et al. Artificial intelligence based malware analysis. arXiv preprint arXiv:1704.08716, 2017.

73. Raza, M.S.; Sheikh, M.N.A.; Hwang, I.-S.; Ab-Rahman, M.S. Feature-Selection-Based DDoS Attack Detection Using AI Algorithms. Telecom 2024, 5, 333-346. https://doi.org/10.3390/telecom5020017

📖 9.2.2

Statement regarding the use of Artificial Intelligence in content creation

This content has been developed with the assistance of artificial intelligence tools, specifically ChatGPT, Gemini, and Notebook LM. These AI technologies were utilized to enhance the text by providing suggestions for rephrasing, improving clarity, and ensuring coherence throughout the material. The integration of these AI tools has enabled a more efficient content creation process while maintaining high standards of quality and accuracy.

The use of AI in this context adheres to all relevant guidelines and ethical considerations associated with the deployment of such technologies. We acknowledge the importance of transparency in the content creation process and aim to provide a clear understanding of how artificial intelligence has contributed to the final product.

PRISCILLA

priscilla.fitped.eu