DAC-01-2581\_PRS2023\_ITS

**UNIVERSITAS NEGERI SEMARANG**

**DAC-01-0074**

**Statstronomers**

**DAC 2023**

**DAC 2023**

**DAC 2023**

**CHAPTER I: Introduction**

The rapid advancement of information technology and the widespread proliferation of telecommunications networks have fundamentally reshaped how we communicate, access information, and conduct global business [1]. Within the context of Indonesia's telecommunications sector, a significant challenge lies in the enduring threat of network attacks. These encompass a spectrum of deliberate actions aimed at undermining the security and functionality of computer networks. These attacks can manifest in various forms, such as Distributed Denial of Service (DDoS) attacks, infiltration by malware, or unauthorized access to sensitive data [2]. The consequences of these attacks may involve disruption of communication services, theft of customer data, or even compromise of the underlying network infrastructure.

However, the research delves deeply into the pivotal endeavour of identifying these network attacks. This intricate process necessitates the utilization of specialized tools and methodologies, frequently revolving around the implementation of Intrusion Detection Systems (IDS) or Network Intrusion Detection Systems (NIDS) [3], [4]. The inherent challenge lies in the creation of mechanisms capable of promptly recognizing and reacting to unauthorized activities within the network in real-time, thereby minimizing potential risks and damage.

Moreover, this research introduces an innovative approach to address these challenges – the utilization of multi-layer classification. This method aims to provide a more comprehensive understanding of network activities by starting with a binary classification, distinguishing between anomalies and normal behaviour [2]. For detected anomalies, it goes a step further, utilized multiclass techniques to precisely identify the specific type of attack. By way of explanation, when an anomaly is initially identified as a Denial-of-Service (DoS) attack, a secondary classification process ensues to determine if it fits into subcategories like DoS, Neptune, or Smurf attacks. Similarly, for probe attacks, the system employs multiclass classification to determine if they can be categorized into subtypes such as Nmap, Portscan, Satan, or Ipsweep attacks [3], [5], [6].

This approach not only enhances our ability to detect network attacks but also provides a more detailed and nuanced view of the threat landscape [7]. It empowers us to respond more effectively by tailoring our actions to specific attack types and subcategories, ultimately bolstering our network security and resilience.

This multi-layered approach holds several advantages, particularly in feature selection. It begins with the outermost layer, distinguishing anomalies from normal activities, and gradually delves into the specifics of attack types [8]. This process aids in identifying which attributes are most relevant for accurate classification, enhancing the overall effectiveness of the system [6]. In essence, the research sets out to contribute by not only detecting network attacks but also by providing a comprehensive understanding of the attack landscape, enabling more informed and targeted cybersecurity efforts in Indonesia's telecommunications sector.

**CHAPTER II: Theoretical Framework**

1. Intrusion Detection Systems

The Intrusion Detection System (IDS) stands as a highly efficient security reinforcement tool, crucial for detecting and safeguarding against cyber-attacks within any network or host [2]. Its fundamental role is to identify and respond to suspicious activities, serving as a proactive measure to protect the network from potential threats and reduce the economic losses that can result from security breaches [9]. This capability makes IDS an integral component in ensuring the security and resilience of digital infrastructures, reinforcing the defence mechanisms against a wide range of cyber threats.

The Intrusion Detection System (IDS) serves as a vital security measure against network attacks and can be classified based on its deployment location: Network-based IDS (NIDS) or Host-based IDS (HIDS) [4].

HIDS operates on a single device within the network, monitoring that device's activities for signs of suspicious behaviour [4], [10]. However, HIDS can strain the resources of the host device and is better suited for protecting individual devices, making it less efficient for large-scale networks [4].

On the other hand, NIDS monitors the entire network and identifies potential threats to network devices. A typical NIDS operates in three key phases: monitoring, detection, and response [4]. During the monitoring phase, it collects statistical network features like packet counts and connections. These features are then used in the classification phase, where Machine Learning (ML) algorithms assess whether the observed characteristics indicate a potential network attack [4]. Based on the classification results, the system initiates suitable defensive actions during the response phase.

1. Machine Learning techniques for NIDS

Machine learning techniques play a pivotal role in fortifying the security of computer networks through the development of Network Intrusion Detection Systems (NIDS) within the cybersecurity domain [2]. Extensive research efforts have been dedicated to exploring various machine learning models, broadly categorized into traditional and advanced approaches, to enhance the effectiveness of NIDS [11].

In the realm of traditional machine learning algorithms, three prominent contenders have risen to prominence: K-Nearest Neighbours (KNN), Support Vector Machines (SVM), and Random Forest [11]. These algorithms are highly recognized for their proficiency in addressing the fundamental challenges of intrusion detection, excelling in classification tasks and feature selection [12]. Consequently, they serve as invaluable tools for constructing effective NIDSs.

In contrast, recent research endeavours have been focused on advanced machine learning techniques, including Multilayer Perceptron’s (MLP), Autoencoders, Gradient Boosting, CatBoost, and XGBoost [11]. These cutting-edge models stand out due to their exceptional ability to identify intricate and subtle patterns within network data [12]. As a result, they make substantial contributions to the development of NIDSs with enhanced capabilities, enabling the detection of even the most sophisticated and rapidly evolving network threats. These advanced techniques not only bolster NIDSs' accuracy but also enhance their adaptability to the rapidly changing threat landscape, ensuring the security and resilience of computer networks [12].

The integration of machine learning techniques, spanning both traditional and advanced approaches, is instrumental in steering the progression of NIDSs. These techniques drive the continuous evolution of NIDS, equipping them with the versatility and agility required to tackle the multifaceted challenges posed by the ever-evolving threat landscape within the cybersecurity domain. NIDS, serving as the first line of defense in safeguarding computer networks, heavily relies on these machine learning approaches to enhance its detection and response capabilities [2]. Traditional algorithms ensure that NIDS can accurately distinguish between normal network behaviour and potentially malicious activities [11], while advanced methods empower NIDSs to detect even the most sophisticated and rapidly evolving threats [12]. In essence, this integration empowers NIDSs to remain at the forefront of network security, contributing significantly to the security and resilience of computer networks.

The integration of machine learning techniques, spanning both traditional and advanced approaches, along with the incorporation of the SMOTE-ENN sampling method, is instrumental in steering the progression of NIDSs. These techniques drive the continuous evolution of NIDS, equipping them with the versatility and agility required to tackle the multifaceted challenges posed by the ever-evolving threat landscape within the cybersecurity domain. NIDS, serving as the first line of defence in safeguarding computer networks, heavily relies on these machine learning approaches and innovative sampling techniques to enhance its detection and response capabilities [2], [11]. Traditional algorithms ensure that NIDS can accurately distinguish between normal network behaviour and potentially malicious activities [11], while advanced methods empower NIDSs to detect even the most sophisticated and rapidly evolving threats [12]. In essence, this integration empowers NIDSs to remain at the forefront of network security, contributing significantly to the security and resilience of computer networks.

**CHAPTER III: Analytical Steps**

1. Data Preparation and Data Preprocessing

Data preparation and preprocessing serve as the initial steps in the analytical process. This phase involves collecting and organizing the data required for the analysis. Raw data is cleaned, transformed, and made ready for further exploration and modelling. It encompasses tasks such as data cleaning to handle missing or erroneous values, data transformation to ensure uniform formats, and data integration to combine information from multiple sources. Proper data preparation and preprocessing lay the foundation for accurate and meaningful insights in subsequent stages.

1. Data Exploration

Data exploration is a critical step to gain a comprehensive understanding of the dataset. It involves the use of descriptive statistics, data visualization, and exploratory data analysis techniques. The primary objective is to uncover patterns, relationships, and potential outliers within the data. Exploratory data analysis helps in identifying trends, correlations, and insights that inform subsequent modelling decisions. It plays a crucial role in feature selection, which is pivotal for the effectiveness of machine learning models.

1. Machine Learning Model Selection

Machine learning model selection is a pivotal decision in the analytical process. It entails choosing the most suitable algorithms and models based on the nature of the problem, the dataset, and the desired outcomes. This step involves experimenting with various machine learning techniques, including traditional ones like K-Nearest Neighbours (KNN), Support Vector Machines (SVM), and Random Forest, as well as advanced models like Multilayer Perceptron’s (MLP), Autoencoders, Gradient Boosting, CatBoost, and XGBoost. The goal is to identify the models that exhibit the highest predictive accuracy and are well-aligned with the specific requirements of the intrusion detection task.

1. Multi-Layer Classification

Multi-layer classification is a distinctive approach employed in this research. It involves a hierarchical classification process that starts with a binary classification to distinguish between anomalies and normal network behaviour. For detected anomalies, it delves deeper into multiclass classification techniques to precisely identify the specific type of network attack. This multilayered approach allows for a nuanced understanding of network activities and facilitates more targeted responses to different attack types and subcategories.

1. Evaluation and Response

The final phase involves the evaluation of the selected machine learning models' performance. Various evaluation metrics are employed to assess their accuracy, precision, recall, F1-score, and other relevant measures. Additionally, the research emphasizes the importance of response mechanisms. When a network attack is detected, the system must initiate appropriate defensive actions promptly to mitigate potential risks and minimize damage. The effectiveness of the response mechanisms is a critical aspect of ensuring network security and resilience.

**CHAPTER IV: Analysis of Results**

(font 12) Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam eget urna nec arcu interdum ullamcorper [3]. Sed rhoncus sapien at justo vehicula, id interdum nisi ullamcorper. Quisque non tellus in neque vulputate dictum sit amet id justo. Sed et volutpat ipsum. Sed vel nunc eu metus blandit dictum. Vestibulum ut tincidunt ligula, in facilisis tortor [4]. Suspendisse ultricies semper justo, eget ultrices dolor pellentesque nec. In hac habitasse platea dictumst. Nullam vel magna urna. Sed tristique, arcu a bibendum lacinia, urna sapien iaculis libero, ut venenatis ligula purus id massa.. These results can serve as a reference to determine in the clustering which provinces have high and low stunting rates.

A graph with different colored bars

Description automatically generated

**(Font 12) Figure 4.1.** Barchart diagram

Nullam eget urna nec arcu interdum ullamcorper. Sed rhoncus sapien at justo vehicula, id interdum nisi ullamcorper. Quisque non tellus in neque vulputate dictum sit amet id justo. Sed et volutpat ipsum. Sed vel nunc eu metus blandit dictum.

Based on **Attachment 1,** it can be observed for provinces from the highest to the lowest average stunting rates. These results can serve as a reference to determine in the clustering which provinces have high and low stunting rates.

**CHAPTER V: Conclusion and Recommendation**

(Font 12) The conclusion should be concise and clear

# REFERENCES

[1] A. Li and S. Yi, “Intelligent Intrusion Detection Method of Industrial Internet of Things Based on CNN-BiLSTM,” *Security and Communication Networks*, vol. 2022, 2022, doi: 10.1155/2022/5448647.

[2] P. Toupas, D. Chamou, K. M. Giannoutakis, A. Drosou, and D. Tzovaras, “An intrusion detection system for multi-class classification based on deep neural networks,” in *Proceedings - 18th IEEE International Conference on Machine Learning and Applications, ICMLA 2019*, Institute of Electrical and Electronics Engineers Inc., Dec. 2019, pp. 1253–1258. doi: 10.1109/ICMLA.2019.00206.

[3] S. Mohammadi, H. Mirvaziri, M. Ghazizadeh-Ahsaee, and H. Karimipour, “Cyber intrusion detection by combined feature selection algorithm,” *Journal of Information Security and Applications*, vol. 44, pp. 80–88, Feb. 2019, doi: 10.1016/j.jisa.2018.11.007.

[4] K. He, D. D. Kim, and M. R. Asghar, “Adversarial Machine Learning for Network Intrusion Detection Systems: A Comprehensive Survey,” *IEEE Communications Surveys and Tutorials*, vol. 25, no. 1, pp. 538–566, 2023, doi: 10.1109/COMST.2022.3233793.

[5] C. Ambedkar and V. Kishore Babu, “Detection of Probe Attacks Using Machine Learning Techniques,” 2015. [Online]. Available: www.arcjournals.org

[6] S. Norwahidayah, Noraniah, N. Farahah, A. Amirah, N. Liyana, and N. Suhana, “Performances of Artificial Neural Network (ANN) and Particle Swarm Optimization (PSO) Using KDD Cup ’99 Dataset in Intrusion Detection System (IDS),” in *Journal of Physics: Conference Series*, IOP Publishing Ltd, Jun. 2021. doi: 10.1088/1742-6596/1874/1/012061.

[7] R. Dalʹnevostochnyĭ federalʹnyĭ universitet (Vladivostok and Institute of Electrical and Electronics Engineers, *Selecting Architecture and Parameters of Deep Neural Networks for Computer Attack Classification*.

[8] A. Halbouni, T. S. Gunawan, M. H. Habaebi, M. Halbouni, M. Kartiwi, and R. Ahmad, “CNN-LSTM: Hybrid Deep Neural Network for Network Intrusion Detection System,” *IEEE Access*, vol. 10, pp. 99837–99849, 2022, doi: 10.1109/ACCESS.2022.3206425.

[9] Y. Zhang, X. Chen, L. Jin, X. Wang, and D. Guo, “Network Intrusion Detection: Based on Deep Hierarchical Network and Original Flow Data,” *IEEE Access*, vol. 7, pp. 37004–37016, 2019, doi: 10.1109/ACCESS.2019.2905041.

[10] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, “Survey of intrusion detection systems: techniques, datasets and challenges,” *Cybersecurity*, vol. 2, no. 1, Dec. 2019, doi: 10.1186/s42400-019-0038-7.

[11] I. H. Sarker, “Machine Learning: Algorithms, Real-World Applications and Research Directions,” *SN Computer Science*, vol. 2, no. 3. Springer, May 01, 2021. doi: 10.1007/s42979-021-00592-x.

[12] P. Nancy, S. Muthurajkumar, S. Ganapathy, S. V. N. Santhosh Kumar, M. Selvi, and K. Arputharaj, “Intrusion detection using dynamic feature selection and fuzzy temporal decision tree classification for wireless sensor networks,” *IET Communications*, vol. 14, no. 5, pp. 888–895, Mar. 2020, doi: 10.1049/iet-com.2019.0172.

**ATTACHMENT**

Attachment 1: Diagram Logic

A screenshot of a computer screen

Description automatically generated

Attachment 2: Merging Logic

|  |
| --- |
| Load binary prediction save as Y |
| if Y == 1:  1. hapus kolom Y  2. modul prediksi multi-class attack  3. new\_Y  if new\_Y == 1:  1. hapus kolom new\_Y  2. model prediksi toa  3. new\_Y2 (ouput: dos, smurf, neptune)  elif new\_Y == 2:  1. hapus kolom new\_Y  2. model prediksi toa  3. new\_Y2 (ouput: nmap, ipsweep, portsweep, satan)  else:  0 -> normal  else -> normal |

Attachment 3: Total Number of Attributes Given in DAC – KDD Dataset

|  |  |  |  |
| --- | --- | --- | --- |
| Index | Feature Name | Index | Feature Name |
| 0 | duration | 21 | is\_guest\_login |
| 1 | protocol\_type | 22 | count |
| 2 | service | 23 | srv\_count |
| 3 | flag | 24 | serror\_rate |
| 4 | src\_bytes | 25 | srv\_serror\_rate |
| 5 | dst\_bytes | 26 | rerror\_rate |
| 6 | land | 27 | srv\_rerror\_rate |
| 7 | wrong\_fragment | 28 | same\_srv\_rate |
| 8 | urgent | 29 | diff\_srv\_rate |
| 9 | hot | 30 | srv\_diff\_host\_rate |
| 10 | num\_failed\_logins | 31 | dst\_host\_count |
| 11 | logged\_in | 32 | dst\_host\_srv\_count |
| 12 | num\_compromised | 33 | dst\_host\_same\_srv\_rate |
| 13 | root\_shell | 34 | dst\_host\_diff\_srv\_rate |
| 14 | su\_attempted | 35 | dst\_host\_same\_src\_port\_rate |
| 15 | num\_root | 36 | dst\_host\_srv\_diff\_host\_rate |
| 16 | num\_file\_creations | 37 | dst\_host\_serror\_rate |
| 17 | num\_shells | 38 | dst\_host\_srv\_serror\_rate |
| 18 | num\_access\_files | 39 | dst\_host\_rerror\_rate |
| 19 | num\_outbound\_cmds | 40 | dst\_host\_srv\_rerror\_rate |
| 20 | is\_host\_login | 41 | type\_of\_attack |

Attachment 4:

**SYNTAX**

# \*\*Notebook Purpose\*\*

\*The purpose of this notebook is to create a predictive model for attacks as part of a Network Intrusion Detection System (NIDS). The prediction is performed using a multi-layer classification approach.\*

# \*\*Import Library\*\*

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import plotly.express as px

import sklearn

from sklearn import preprocessing

from scipy.interpolate import PchipInterpolator

from imblearn.combine import SMOTEENN

from scipy.stats import spearmanr

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

from sklearn.feature\_selection import RFE

from sklearn.ensemble import RandomForestClassifier

import itertools

from sklearn.ensemble import GradientBoostingClassifier, GradientBoostingRegressor

from sklearn.metrics import roc\_auc\_score, r2\_score

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import LabelEncoder

from feature\_engine.selection import (

RecursiveFeatureElimination,

RecursiveFeatureAddition

)

# \*\*Load Dataset\*\*

# Path to training dataset

train\_PATH = 'D:\DAC-2023\Dataset\DataTrain\_Preliminary.csv'

pred\_PATH = 'D:\DAC-2023\Dataset\Data\_Prediction.csv'

# Load dataset

df = pd.read\_csv(train\_PATH, delimiter=';')

df\_pred = pd.read\_csv(pred\_PATH, delimiter=';')

df.head()

df.info()

Here we could see that the data type of every column is 'object' so we need to change the data type based on the value the column has.

print('Label distribution Training set:')

print(df['type\_of\_attack'].value\_counts())

We will use multi-layer classification to predict attacks.

1. If the network activity is classified as an anomaly:

- If it's a Denial-of-Service (DoS) attack:

- Perform multiclass classification to determine if it's a DoS, Neptune, or Smurf attack.

- If it's a probe attack:

- Perform multiclass classification to determine if it's Nmap, Portscan, Satan, or Ipsweep.

2. If the network activity is not classified as an anomaly, it is considered normal.

# Membuat kolom 'target' dengan nilai awal 'anomaly'

df['target'] = 'anomaly'

# Mengganti nilai kolom 'target' berdasarkan nilai kolom 'type\_of\_attack'

df.loc[df['type\_of\_attack'] == 'normal', 'target'] = 'normal'

# Sisipkan kolom 'target' tepat sebelum kolom terakhir

df.insert(df.columns.get\_loc('type\_of\_attack'), 'target', df.pop('target'))

# Fungsi untuk menentukan nilai di kolom "attack"

def determine\_attack\_type(row):

if row['type\_of\_attack'] == 'normal':

return 'normal'

elif row['type\_of\_attack'] in ['nmap', 'portsweep', 'satan', 'ipsweep']:

return 'probe'

elif row['type\_of\_attack'] in ['neptune', 'smurf', 'Denial of Service Attack']:

return 'dos'

else:

return 'unknown' # Jika jenis serangan tidak dikenali

# Buat kolom baru 'attack'

df['attack'] = df.apply(determine\_attack\_type, axis=1)

# Sisipkan kolom 'attack' tepat sebelum kolom terakhir

df.insert(len(df.columns) - 2, 'attack', df.pop('attack'))

df.head()

df.shape

# \*\*Data Preprocessing\*\*

## 1. Investigate Unique Value

# Initialize a dictionary to store missing value representations

missing\_value\_representations = ['', ' ', '-', 'NA', 'N/A', '999', '-999', None,

np.nan, '\*', '9999', '-9999', '99999', '-99999',

99999]

# Initialize a set to store unique types of missing values

unique\_missing\_value\_types = set()

# Iterate through columns and check for missing value representations

for column in df.columns:

for value in df[column]:

if value in missing\_value\_representations:

unique\_missing\_value\_types.add(value)

# Boolean indexing to filter rows with unique missing value types

filtered\_df = df[df.apply(lambda row: any(val in unique\_missing\_value\_types for val in row), axis=1)]

# Display the filtered DataFrame

print("Unique types of missing values found in the DataFrame:")

print(unique\_missing\_value\_types)

print("Rows containing unique missing value types:")

filtered\_df

## 2. Remove NaN Values

We will apply an imputation method to handle missing values in numeric columns. However, since imputation cannot be applied to object-type columns, we will remove rows where missing values exist in those object-type columns.

# Count rows with NaN values in specific columns

columns\_to\_check = ['protocol\_type', 'service', 'flag', 'type\_of\_attack']

nan\_rows\_count = df[columns\_to\_check].isnull().any(axis=1).sum()

print("Number of rows with NaN values in the specified columns:", nan\_rows\_count)

# Calculate the percentage of rows with missing values in column 'protocol\_type', 'service', 'flag', 'type\_of\_attack'

percentage\_rows\_with\_missing = (nan\_rows\_count / len(df)) \* 100

print(f"Persentage of rows with NaN value/s: {percentage\_rows\_with\_missing:.2f}%")

Because it is less then 5%, so we could delete it.

df = df.dropna(subset=columns\_to\_check)

df.shape

## 3. Imputation

### 3.1. Binary Imputation

# column that is binary (0=no, 1=yes)

bin\_df = df.copy()

bin\_df = bin\_df[['land','logged\_in','root\_shell','su\_attempted',

'is\_host\_login','is\_guest\_login','type\_of\_attack']]

# Function to filter df based on type of attack

def filtered\_by\_type\_of\_attack(df, values):

filtered = df[df['type\_of\_attack'] == values]

return filtered

# List of unique attack types

attack\_types = bin\_df['type\_of\_attack'].unique()

# Initialize a dictionary to store probabilities for each attack type

probability\_dict = {}

# Iterate through each attack type and store probabilities

for attack\_type in attack\_types:

filtered\_data = filtered\_by\_type\_of\_attack(bin\_df, attack\_type)

probability\_land = filtered\_data['land'].value\_counts(normalize=True).get('0', 0)

probability\_logged\_in = filtered\_data['logged\_in'].value\_counts(normalize=True).get('0', 0)

probability\_root\_shell = filtered\_data['root\_shell'].value\_counts(normalize=True).get('0', 0)

probability\_su\_attempted = filtered\_data['su\_attempted'].value\_counts(normalize=True).get('0', 0)

probability\_is\_host\_login = filtered\_data['is\_host\_login'].value\_counts(normalize=True).get('0', 0)

probability\_is\_guest\_login = filtered\_data['is\_guest\_login'].value\_counts(normalize=True).get('0', 0)

# Store probabilities in the dictionary

probability\_dict[attack\_type] = {

'land': probability\_land,

'logged\_in': probability\_logged\_in,

'root\_shell': probability\_root\_shell,

'su\_attempted': probability\_su\_attempted,

'is\_host\_login': probability\_is\_host\_login,

'is\_guest\_login': probability\_is\_guest\_login,

}

# Function to convert '\*' and '99999' based on probability

def convert\_value(value, probability):

if value == '\*' or value == '99999':

random\_number = np.random.rand()

if random\_number <= probability:

return '0'

else:

return '1'

else:

return value

for column in ['land', 'logged\_in', 'root\_shell', 'su\_attempted', 'is\_host\_login', 'is\_guest\_login']:

# Ambil probabilitas yang sesuai dari dictionary

probability = probability\_dict[attack\_type][column]

# Terapkan konversi ke seluruh kolom dalam DataFrame yang sesuai dengan jenis serangan saat ini

bin\_df.loc[bin\_df['type\_of\_attack'] == attack\_type, column] = bin\_df.loc[bin\_df['type\_of\_attack'] == attack\_type, column].apply(lambda x: convert\_value(x, probability))

# Delete 'type\_of\_attack' column

bin\_df = bin\_df.drop('type\_of\_attack', axis=1)

# Mengubah tipe data menjadi numerik

bin\_df = bin\_df.astype(int)

### 3.2. Integer Imputation

# column that has int value

int\_df = df.copy()

int\_df = int\_df[['duration','src\_bytes','dst\_bytes','wrong\_fragment',

'urgent','hot','num\_failed\_logins','num\_compromised',

'num\_root','num\_file\_creations','num\_shells','num\_access\_files',

'num\_outbound\_cmds','count','srv\_count','dst\_host\_count',

'dst\_host\_srv\_count','type\_of\_attack']]

# 1. konversi \* dan 99999 jadi nan value

int\_df.replace(['\*'], '99999', inplace=True)

# 2. ubah tipe data

# Mengubah tipe data menjadi numerik

columns\_to\_convert = [col for col in int\_df.columns if col != 'type\_of\_attack']

int\_df[columns\_to\_convert] = int\_df[columns\_to\_convert].astype(int)

# 3. parting based on type of attack

nmap = filtered\_by\_type\_of\_attack(int\_df, 'nmap')

neptune = filtered\_by\_type\_of\_attack(int\_df, 'neptune')

normal = filtered\_by\_type\_of\_attack(int\_df, 'normal')

dos = filtered\_by\_type\_of\_attack(int\_df, 'Denial of Service Attack')

portsweep = filtered\_by\_type\_of\_attack(int\_df, 'portsweep')

satan = filtered\_by\_type\_of\_attack(int\_df, 'satan')

ipsweep = filtered\_by\_type\_of\_attack(int\_df, 'ipsweep')

smurf = filtered\_by\_type\_of\_attack(int\_df, 'smurf')

# stored df in list

filtered\_dfs = [nmap, neptune, normal, dos, portsweep, satan, ipsweep, smurf]

# Loop through each filtered DataFrame and its corresponding index in the list

for filtered\_df, attack\_type in zip(filtered\_dfs, ['nmap', 'neptune', 'normal', 'Denial of Service Attack', 'portsweep', 'satan', 'ipsweep', 'smurf']):

for column in filtered\_df.columns:

if pd.api.types.is\_numeric\_dtype(filtered\_df[column]):

# Check if the column contains numeric data

missing\_mask = filtered\_df[column] == 99999

if missing\_mask.any():

# If there are missing values, interpolate

x = filtered\_df.index[~missing\_mask]

y = filtered\_df[column][~missing\_mask]

pchip = PchipInterpolator(x, y, extrapolate='periodic')

# Replace NaN values with interpolated values in the same DataFrame (filtered\_df)

interpolated\_values = pchip(filtered\_df.index)

# Ensure the interpolated values are non-negative

filtered\_df[column].loc[missing\_mask] = np.maximum(0, interpolated\_values[missing\_mask])

# Copy the interpolated values back to the corresponding rows in the original int\_df

int\_df.loc[int\_df['type\_of\_attack'] == attack\_type, filtered\_df.columns] = filtered\_df

# Mengubah tipe data menjadi numerik

columns\_to\_convert = [col for col in int\_df.columns if col != 'type\_of\_attack']

int\_df[columns\_to\_convert] = int\_df[columns\_to\_convert].astype(int)

# Delete 'type\_of\_attack' column

int\_df = int\_df.drop('type\_of\_attack', axis=1)

### 3.3. Float Imputation

# column that has float value

float\_df = df.copy()

float\_df = float\_df[['serror\_rate','srv\_serror\_rate','rerror\_rate',

'srv\_rerror\_rate','same\_srv\_rate','diff\_srv\_rate',

'srv\_diff\_host\_rate','dst\_host\_same\_srv\_rate',

'dst\_host\_diff\_srv\_rate','dst\_host\_same\_src\_port\_rate',

'dst\_host\_srv\_diff\_host\_rate','dst\_host\_serror\_rate',

'dst\_host\_srv\_serror\_rate','dst\_host\_rerror\_rate',

'dst\_host\_srv\_rerror\_rate','type\_of\_attack']]

# 1. konversi \* dan 99999 jadi nan value

float\_df.replace(['\*','99999',99999], np.nan, inplace=True)

# 2. ubah tipe data

# Mengubah tipe data menjadi numerik

columns\_to\_convert = [col for col in float\_df.columns if col != 'type\_of\_attack']

float\_df[columns\_to\_convert] = float\_df[columns\_to\_convert].astype(float)

# 3. parting based on type of attack

nmap = filtered\_by\_type\_of\_attack(float\_df, 'nmap')

neptune = filtered\_by\_type\_of\_attack(float\_df, 'neptune')

normal = filtered\_by\_type\_of\_attack(float\_df, 'normal')

dos = filtered\_by\_type\_of\_attack(float\_df, 'Denial of Service Attack')

portsweep = filtered\_by\_type\_of\_attack(float\_df, 'portsweep')

satan = filtered\_by\_type\_of\_attack(float\_df, 'satan')

ipsweep = filtered\_by\_type\_of\_attack(float\_df, 'ipsweep')

smurf = filtered\_by\_type\_of\_attack(float\_df, 'smurf')

# stored df in list

filtered\_dfs = [nmap, neptune, normal, dos, portsweep, satan, ipsweep, smurf]

# Loop through each filtered DataFrame and its corresponding index in the list

for filtered\_df, attack\_type in zip(filtered\_dfs, ['nmap', 'neptune', 'normal', 'Denial of Service Attack', 'portsweep', 'satan', 'ipsweep', 'smurf']):

for column in filtered\_df.columns:

if pd.api.types.is\_numeric\_dtype(filtered\_df[column]):

# Check if the column contains numeric data

missing\_mask = filtered\_df[column].isnull()

if missing\_mask.any():

# If there are missing values, interpolate

x = filtered\_df.index[~missing\_mask]

y = filtered\_df[column][~missing\_mask]

pchip = PchipInterpolator(x, y, extrapolate='periodic')

# Replace NaN values with interpolated values in the same DataFrame (filtered\_df)

interpolated\_values = pchip(filtered\_df.index)

# Ensure the interpolated values are non-negative

filtered\_df[column].loc[missing\_mask] = np.maximum(0, interpolated\_values[missing\_mask])

# Copy the interpolated values back to the corresponding rows in the original float\_df

float\_df.loc[float\_df['type\_of\_attack'] == attack\_type, filtered\_df.columns] = filtered\_df

# Delete 'type\_of\_attack' column

float\_df = float\_df.drop('type\_of\_attack', axis=1)

### 3.4. Numeric Data Replacement

# Create a dictionary where keys are column names and values are corresponding DataFrames

replacement\_data = {

'land': bin\_df,

'logged\_in': bin\_df,

'root\_shell': bin\_df,

'su\_attempted': bin\_df,

'is\_host\_login': bin\_df,

'is\_guest\_login': bin\_df,

'serror\_rate': float\_df,

'srv\_serror\_rate': float\_df,

'rerror\_rate': float\_df,

'srv\_rerror\_rate': float\_df,

'same\_srv\_rate': float\_df,

'diff\_srv\_rate': float\_df,

'srv\_diff\_host\_rate': float\_df,

'duration': int\_df,

'src\_bytes': int\_df,

'dst\_bytes': int\_df,

'wrong\_fragment': int\_df,

'urgent': int\_df,

'hot': int\_df,

'num\_failed\_logins': int\_df,

'num\_compromised': int\_df,

'num\_root': int\_df,

'num\_file\_creations': int\_df,

'num\_shells': int\_df,

'num\_access\_files': int\_df,

'num\_outbound\_cmds': int\_df,

'count': int\_df,

'srv\_count': int\_df,

'dst\_host\_count': int\_df,

'dst\_host\_srv\_count': int\_df,

'dst\_host\_same\_srv\_rate': float\_df,

'dst\_host\_diff\_srv\_rate': float\_df,

'dst\_host\_same\_src\_port\_rate': float\_df,

'dst\_host\_srv\_diff\_host\_rate': float\_df,

'dst\_host\_serror\_rate': float\_df,

'dst\_host\_srv\_serror\_rate': float\_df,

'dst\_host\_rerror\_rate': float\_df,

'dst\_host\_srv\_rerror\_rate': float\_df,

}

# Iterate over the columns and replace data in 'df'

for column\_name, source\_df in replacement\_data.items():

df[column\_name] = source\_df[column\_name]

df.dtypes

## 4. Remove Constant, Quasi-Constand and Duplicated Features

# remove constant, quasi-constand and duplicated features

quasi\_constant\_feat = []

# iterate over every feature

for feature in df.columns:

# find the predominant value, that is the value that is shared

# by most observations

predominant = df[feature].value\_counts(

normalize=True).sort\_values(ascending=False).values[0]

# evaluate the predominant feature: do more than 99% of the observations

# show 1 value?

if predominant > 0.998:

# if yes, add the variable to the list

quasi\_constant\_feat.append(feature)

df.drop(labels=quasi\_constant\_feat, axis=1, inplace=True)

df.shape

quasi\_constant\_feat

## 5. Remove Duplicate Rows

df = df.drop\_duplicates()

df.shape

# \*\*Exploratory Data Analysis\*\*

## 1. Statistical Measure

df.describe()

df.describe(include='object')

## 2. Label Distribution

class\_distribution = df['target'].value\_counts().reset\_index()

class\_distribution.columns = ['target', 'count']

fig = px.treemap(class\_distribution,

path=['target'],

values='count',

title='Distribution of Target')

fig.show()

class\_distribution = df['attack'].value\_counts().reset\_index()

class\_distribution.columns = ['attack', 'count']

fig = px.treemap(class\_distribution,

path=['attack'],

values='count',

title='Distribution of Attack')

fig.show()

class\_distribution = df['type\_of\_attack'].value\_counts().reset\_index()

class\_distribution.columns = ['type\_of\_attack', 'count']

fig = px.treemap(class\_distribution,

path=['type\_of\_attack'],

values='count',

title='Distribution of Type of Attacks')

fig.show()

## 3. Data Normalization

from sklearn import preprocessing

from sklearn.preprocessing import StandardScaler

# selecting numeric attributes columns from df

numeric\_col = df.select\_dtypes(include='number').columns

# using standard scaler for normalizing

std\_scaler = StandardScaler()

def normalization(df,col):

for i in col:

arr = df[i]

arr = np.array(arr)

df[i] = std\_scaler.fit\_transform(arr.reshape(len(arr),1))

return df

# data before normalization

df.head()

# calling the normalization() function

df = normalization(df, numeric\_col)

# data after normalization

df.head()

## 4. Correlation

# feature

numeric\_col = df.select\_dtypes(include='number').columns

X = df[numeric\_col]

# label

y = df[['target','attack','type\_of\_attack']]

# Calculate the Spearman correlation matrix

corr = X.corr(method='spearman')

# Change the main diagonal (correlation of columns with themselves) to NaN

for i in range(corr.shape[0]):

corr.iloc[i, i] = np.nan

# Find columns with correlations > 0.95 or < -0.95

high\_corr\_cols = (corr.abs() > 0.95).any()

# Get the column names with high correlations and their correlation values

high\_corr\_columns = X.columns[high\_corr\_cols]

# Create a dictionary to store the correlated columns and their correlation values

correlation\_data = {}

# Calculate and store the correlated columns and their correlation values

for col in high\_corr\_columns:

correlated\_cols = corr.index[corr[col].abs() > 0.8].tolist()

if col in correlated\_cols:

correlated\_cols.remove(col) # Remove itself from the list if present

correlated\_data = {}

for correlated\_col in correlated\_cols:

correlation\_value, \_ = spearmanr(X[col], X[correlated\_col])

correlated\_data[correlated\_col] = correlation\_value

correlation\_data[col] = correlated\_data

# Define the file path where you want to save the output

output\_file\_path = 'high\_correlation.txt'

# Open the file for writing

with open(output\_file\_path, 'w') as file:

# Iterate through the correlation data dictionary and write it to the file

for col, correlated\_data in correlation\_data.items():

file.write(f"Kolom '{col}' berkorelasi dengan kolom-kolom berikut:\n")

for correlated\_col, correlation\_value in correlated\_data.items():

file.write(f"{correlated\_col}: {correlation\_value:.2f}\n")

file.write('\n')

high\_corr\_columns = ['serror\_rate','srv\_serror\_rate','dst\_host\_srv\_serror\_rate']

# drop features with high correlation

X.drop(high\_corr\_columns, axis=1, inplace=True)

df.drop(high\_corr\_columns, axis=1, inplace=True)

X.shape, y.shape, df.shape

# Check heat map with blue colormap

corr\_mat = X.corr(method='spearman')

f, ax = plt.subplots(figsize=(7, 6))

sns.heatmap(corr\_mat, vmax=.8, square=True, cmap='Blues', ax=ax)

plt.show()

# \*\*Feature Engineering\*\*

## 1. For Binary Classification

bin\_df = df.copy()

bin\_df = bin\_df.drop(['attack','type\_of\_attack'],axis=1)

bin\_df.head()

### 1.1. Encoding

#### Categorical Features to One Hot Encoding

# explore categorical features

print('Training Data:')

for col\_name in bin\_df.columns:

if col\_name not in ['target', 'attack', 'type\_of\_attack'] and bin\_df[col\_name].dtype == 'object':

unique\_cat = len(bin\_df[col\_name].unique())

print(f"Feature '{col\_name}' has {unique\_cat} categories")

# selecting categorical data attributes

cat\_col = ['protocol\_type','service','flag']

# creating a dataframe with only categorical attributes

cat\_df = bin\_df[cat\_col]

cat\_df.head()

# one-hot-encoding categorical attributes using pandas.get\_dummies() function

cat\_df = pd.get\_dummies(cat\_df, columns=cat\_col, dtype=int)

cat\_df.head()

Join encoded categorical dataframe with the non-categorical dataframe.

# Simpan kolom 'target' dalam variabel terpisah

target\_col = bin\_df['target']

# Hapus kolom 'target' dan categorical column sebelum di-encoding dari DataFrame

bin\_df.drop('target', axis=1, inplace=True)

bin\_df.drop('flag', axis=1, inplace=True)

bin\_df.drop('protocol\_type', axis=1, inplace=True)

bin\_df.drop('service', axis=1, inplace=True)

# Gabungkan DataFrame cat\_df ke dalam bin\_df

bin\_df = bin\_df.join(cat\_df)

# Sisipkan kolom 'target' kembali ke paling kanan DataFrame

bin\_df['target'] = target\_col

bin\_df.shape

bin\_df

#### Label to Label Encoding

# creating a dataframe with binary labels (normal, anomaly)

bin\_df['target\_label'] = bin\_df['target'].replace({'normal': 0, 'anomaly': 1})

bin\_df

# Menghitung jumlah 'normal' dan 'abnormal' dalam kolom 'label'

label\_counts = bin\_df['target'].value\_counts()

# Membuat bar chart

plt.figure(figsize=(3, 2))

plt.bar(label\_counts.index, label\_counts.values)

plt.xlabel('Label')

plt.ylabel('Count')

plt.title('Bar Chart Distribution of Normal and Anomaly Labels')

plt.show()

# # label encoder

# def label\_encoder(df):

# columns\_to\_encode = ['target', 'attack', 'type\_of\_attack']

# for col in columns\_to\_encode:

# if col in df.columns:

# label\_encoder = LabelEncoder()

# df[f"{col}\_ohe"]

# df[f"{col}\_ohe"] = label\_encoder.fit\_transform(df[col])

# # Membuat kolom 'target' dengan nilai awal 'anomaly'

# df['target'] = 'anomaly'

# # Mengganti nilai kolom 'target' berdasarkan nilai kolom 'type\_of\_attack'

# df.loc[df['type\_of\_attack'] == 'normal', 'target'] = 'normal'

# # Sisipkan kolom 'target' tepat sebelum kolom terakhir

# df.insert(df.columns.get\_loc('type\_of\_attack'), 'target', df.pop('target'))

# label\_encoder(bin\_df1)

### 1.2. Feature Scaling

# feature

X\_bin = bin\_df.drop(columns=['target','target\_label'])

# label

y\_bin = bin\_df[['target','target\_label']]

X\_bin.shape, y\_bin.shape

from sklearn.preprocessing import StandardScaler

# Inisialisasi objek StandardScaler

scaler = StandardScaler()

# Fit dan transformasi scaler pada DataFrame X\_bin

X\_bin\_scaled = scaler.fit\_transform(X\_bin)

# Menghasilkan DataFrame baru dengan data yang telah diubah skala

X\_bin = pd.DataFrame(X\_bin\_scaled, columns=X\_bin.columns)

# Tampilkan DataFrame yang telah diubah skala

X\_bin

### 1.3. Save Binary Data

# Menggabungkan X\_bin\_scaled\_df dengan y\_bin

binary\_df = pd.concat([X\_bin, y\_bin], axis=1)

# Simpan sebagai file CSV

binary\_df.to\_csv('D:\DAC-2023\Dataset\\binary.csv', index=False)

#### \*\*SMOTEENN Method\*\*

# # Define the list of target columns in order

# target\_columns = ['target','attack','type\_of\_attack'] # Replace with your actual target column names

# # Create an empty DataFrame to store the resampled data

# resampled\_df = pd.DataFrame()

# # Loop through each target column

# for target\_col in target\_columns:

# # Separate features and target for the current target

# X\_target = X # Features

# y\_target = y[target\_col] # Target

# # Apply SMOTE-ENN to balance the current target

# smote\_enn = SMOTEENN(sampling\_strategy='auto', random\_state=42)

# X\_resampled, y\_resampled = smote\_enn.fit\_resample(X\_target, y\_target)

# # Combine the resampled data with the resampled\_df

# resampled\_data = pd.DataFrame(data=X\_resampled, columns=X\_target.columns)

# resampled\_data[target\_col] = y\_resampled

# resampled\_df = pd.concat([resampled\_df, resampled\_data], axis=0)

# # At this point, 'resampled\_df' should contain balanced data for all target columns

# # Path to save the resampled data

# file\_path = 'resampled\_data.csv'

# # Save the resampled DataFrame to a CSV file

# resampled\_df.to\_csv(file\_path, index=False)

# # Load resampled data

# df = pd.read\_csv('D:\DAC-2023\\resampled\_data.csv')

# df

# class\_distribution = df['target'].value\_counts().reset\_index()

# class\_distribution.columns = ['target', 'count']

# fig = px.treemap(class\_distribution,

# path=['target'],

# values='count',

# title='Distribution of Target')

# fig.show()

## 2. For Multiclass Classification

attack\_df = df.copy()

attack\_df = attack\_df.drop(['target','type\_of\_attack'],axis=1)

attack\_df.head()

toa\_df = df.copy()

toa\_df = toa\_df.drop(['target','attack'],axis=1)

toa\_df.head()

### 2.1. Encoding

#### Encoding for attack\_df

# creating a dataframe with labels (normal, dos, probe)

attack\_df['attack\_label'] = attack\_df['attack'].replace({'normal': 0, 'dos':1, 'probe':2})

attack\_df

# Menghitung jumlah 'normal' dan 'abnormal' dalam kolom 'label'

label\_counts = attack\_df['attack'].value\_counts()

# Membuat bar chart

plt.figure(figsize=(3, 2))

plt.bar(label\_counts.index, label\_counts.values)

plt.xlabel('Label')

plt.ylabel('Count')

plt.title('Distribution of Attack Labels')

plt.show()

#### Encoding for type\_of\_attack

toa\_df.type\_of\_attack.unique()

# creating a dataframe with labels (normal, dos, probe)

toa\_df['toa\_label'] = toa\_df['type\_of\_attack'].replace({'normal': 0, 'Denial of Service Attack':1,

'neptune':2, 'smurf':3, 'nmap':4,

'portsweep':5, 'satan':6, 'ipsweep':7})

toa\_df

# Menghitung jumlah 'normal' dan 'abnormal' dalam kolom 'label'

label\_counts = toa\_df['type\_of\_attack'].value\_counts()

# Membuat bar chart

plt.figure(figsize=(3, 2))

plt.bar(label\_counts.index, label\_counts.values)

plt.xlabel('Label')

plt.ylabel('Count')

plt.title('Distribution of Type of Attack Labels')

plt.xticks(rotation=90)

plt.show()

### 2.2. Save Multiclass Data

#### attack.csv

# Feature

X\_bin

# Target

y\_attack = attack\_df[['attack','attack\_label']]

# Menggabungkan X\_bin dengan y\_attack

attack\_df = pd.concat([X\_bin, y\_attack], axis=1)

# Simpan sebagai file CSV

attack\_df.to\_csv('D:\DAC-2023\Dataset\\attack.csv', index=False)

#### toa.csv

# Feature

X\_bin

# Target

y\_toa = toa\_df[['type\_of\_attack','toa\_label']]

# Menggabungkan X\_bin dengan y\_toa

toa\_df = pd.concat([X\_bin, y\_attack], axis=1)

# Simpan sebagai file CSV

toa\_df.to\_csv('D:\DAC-2023\Dataset\\toa.csv', index=False)

# \*\*Feature Selection\*\*

### 1. Train and Test Sets

# separate train and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X,

y['target'],

test\_size=0.2,

random\_state=5)

X\_train.shape, X\_test.shape, y\_train.shape, y\_test.shape

### 2. RFE - Random Forest

# Initialize the RandomForestClassifier

rfc = RandomForestClassifier()

# Initialize the RFE with the RandomForestClassifier

rfe = RFE(rfc, n\_features\_to\_select=10)

# Fit the RFE to the training data

rfe = rfe.fit(X\_train, y\_train)

# Get the support from RFE, indicating which features are selected

support = rfe.support\_

# Get the ranking from RFE, indicating the importance ranking of features

ranking = rfe.ranking\_

# Initialize a list to store selected features

selected\_features = []

# Initialize a list to store AUC scores at each iteration

auc\_scores = []

# Loop through the selected features and compute AUC at each iteration

for i, (is\_selected, feature) in enumerate(zip(support, X\_train.columns)):

if is\_selected:

selected\_features.append(feature)

# Train a model with the selected features

rfc.fit(X\_train[selected\_features], y\_train)

# Make predictions on the test set

y\_pred = rfc.predict\_proba(X\_test[selected\_features])[:, 1]

# Calculate the AUC score

auc = roc\_auc\_score(y\_test, y\_pred)

auc\_scores.append(auc)

print(f"Iteration {i + 1}: AUC = {auc:.4f}")

# Now 'selected\_features' contains the final selected features

# 'auc\_scores' contains the AUC scores at each iteration

selected\_features

### 3. RFA - Random Forest

# Initialize the RandomForestClassifier

rfc = RandomForestClassifier()

# Initialize empty lists to store selected features and AUC scores

selected\_features = []

auc\_scores = []

# Initialize a list of all features

all\_features = X\_train.columns.tolist()

# Initialize a variable to store the best AUC score

best\_auc = 0

# Loop through the features and add them one by one based on AUC improvement

while len(all\_features) > 0:

best\_feature = None

for feature in all\_features:

# Try adding the feature to the selected features

current\_features = selected\_features + [feature]

# Train a model with the current selected features

rfc.fit(X\_train[current\_features], y\_train)

# Make predictions on the test set

y\_pred = rfc.predict\_proba(X\_test[current\_features])[:, 1]

# Calculate the AUC score

auc = roc\_auc\_score(y\_test, y\_pred)

# Check if the AUC improved

if auc > best\_auc:

best\_auc = auc

best\_feature = feature

if best\_feature is not None:

selected\_features.append(best\_feature)

auc\_scores.append(best\_auc)

all\_features.remove(best\_feature)

print(f"Selected Feature: {best\_feature}, AUC = {best\_auc:.4f}")

else:

break

# Now 'selected\_features' contains the final selected features

# 'auc\_scores' contains the AUC scores at each iteration

selected\_features

### 4. RFE - Gradient Boosting

# the ML model for which we want to select features

model = GradientBoostingClassifier(

n\_estimators=10,

max\_depth=2,

random\_state=10

)

# Setup the RFE selector

sel = RecursiveFeatureElimination(

variables=None, # automatically evaluate all numerical variables

estimator = model, # the ML model

scoring = 'roc\_auc', # the metric we want to evalute

threshold = 0.0005, # the maximum performance drop allowed to remove a feature

cv=2, # cross-validation

)

# this may take quite a while, because

# we are building a lot of models with cross-validation

sel.fit(X\_train, y\_train)

# performance of model trained using all features

sel.initial\_model\_performance\_

all\_columns = X\_train.columns.tolist()

# List of columns to be dropped

columns\_to\_drop = sel.features\_to\_drop\_

# Find columns that are not in the list of columns to drop

columns\_to\_keep = [col for col in all\_columns if col not in columns\_to\_drop]

# Now, 'columns\_to\_keep' contains the columns that are not in the list of columns to drop

print("Columns to keep:")

print(columns\_to\_keep)

### 4. RFA - Gradient Boosting

# Setup the RFA selector

rfa = RecursiveFeatureAddition(

variables=None, # automatically evaluate all numerical variables

estimator=model, # the ML model

scoring='roc\_auc', # the metric we want to evalute

threshold=0.0001, # the minimum performance increase needed to select a feature

cv=2, # cross-validation

)

rfa.fit(X\_train, y\_train)

# performance of model trained using all features

rfa.initial\_model\_performance\_

all\_columns = X\_train.columns.tolist()

# List of columns to be dropped

columns\_to\_drop = sel.features\_to\_drop\_

# Find columns that are not in the list of columns to drop

columns\_to\_keep = [col for col in all\_columns if col not in columns\_to\_drop]

# Now, 'columns\_to\_keep' contains the columns that are not in the list of columns to drop

print("Columns to keep:")

print(columns\_to\_keep)