Comparative study of machine learning applications in malware forensics

Bayan Abduraimova[1,†], Sergiy Gnatyuk[2,†] and Albina Nurmukhanbetova[1,*,†]

[1] *L. N. Gumilyov Eurasian National University, 2 Satbayev str., 010000 Astana, Kazakhstan*
[2] *National Aviation University, 1 Liubomyra Huzara ave., 03058 Kyiv, Ukraine*

Abstract

In the digital world, one of the most significant threats is malicious software, known as malware, developed by cyber attackers to intentionally cause damage to computer systems or gain access to them. The form and behavior of malware have developed year by year at the same time methods of detection from malicious software have evolved to ensure security. Earlier for disclosure of malicious software some classical methods, such as signature-based, heuristic, and so on. Traditional methods for detecting malware have failed to defeat new generations of malware and their sophisticated obfuscation tactics. However, at present, the use of detection methods based on machine learning has been recognized as one of the most modern and prominent methods. Methods based on machine learning provide fast malware prediction with excellent detection and analysis rates for various types of malware. Therefore, this research work represents a comparison of various machine learning approaches, including neural networks, decision trees, the support vector machine, and ensemble methods applied to the analysis of behavioral and static characteristics of malware. The research paper analyzes the advantages and limitations of each method, their effectiveness in various types of malicious attacks, and the possibilities of their adaptation to changing threats. The current trends and directions of development in the use of machine learning in malware forensics, including the use of deep learning and big data analysis technologies, are also considered. The objective of this research is to conduct a comparative analysis of various machine learning techniques applied in malware forensics, focusing on their effectiveness in detecting, classifying, and analyzing malicious software. The study aims to identify the strengths and weaknesses of these techniques in real-world digital forensic scenarios, providing insights into their applicability and performance in enhancing malware investigation processes. The results of the study emphasize the prospects of using machine learning to improve malware detection and control, as well as the importance of further research to develop new models and methods of data analysis to protect information systems from modern cyber threats.

Keywords

cyber security, malicious software, malware forensics, machine learning algorithms, SVM, decision tree, random forest

1. Introduction

Malicious software, known as malware, is developed to harm digital devices on purpose. Nowadays, there are many types of malware, such as trojan horses, worms, viruses, bots and botnets, ransomware, adware, and spyware. Attackers targeted individual computers and networks, which led to an increase of holes in the system of security. The most significant issue in this situation is confidentiality and leakage of personal information. Moreover, financial damage caused by malware software all over the world increased proportionally. According to the report of McAfee about COVID-19 Threats and Malware Surges, the new malware samples averaged 648 new threats per minute [1]. Also, recent estimates of the AV-Test show that every day over 450,000 new malicious programs and potentially unwanted applications [2]. To ensure an answer to cyber threats, which are complex and unsafe, most of the farms developed different instruments for cyber security. Researchers used machine learning and deep learning algorithms to create effective models to solve these problems by conducting detailed research. Malware protection methods are also becoming more sophisticated as the variety of malware increases.

Earlier, malware was written by using simple codes and they were easily detected. Currently, creators of malicious software complicate the code, as a result even advanced methods cannot detect them. The next generation of malicious programs is hard to identify by comparison with traditional malicious programs designed to run in the kernel. These types of malware programs slip away easily

CPITS-II 2024: Workshop on Cybersecurity Providing in Information and Telecommunication Systems II, October 26, 2024, Kyiv, Ukraine
*Corresponding author.
† These authors contributed equally.
abduraimovabk@mail.ru (B. Abduraimova);
sergio.gnatyuk@gmail.com (S. Gnatyuk);
nuralbina9898@gmail.com (A. Nurmukhanbetova)

0000-0003-3913-1895 (B. Abduraimova);
0000-0003-4992-0564 (S. Gnatyuk);
0009-0009-7123-9522 (A. Nurmukhanbetova)

from the security system, such as firewalls and antiviruses. There are different approaches to malware detection, based on different functions, such as signature, heuristic, behavior, model verification, cloud, mobile devices, Internet of things, machine learning, and deep learning. At first, the detection of malware programs used signature-based detection. Despite its speed, it cannot identify complex malware. Traditional signature-based methods, such as pattern matching, do not meet the requirements for malware detection. So, in this case, the use of machine learning algorithms is suitable for the detection of malicious software now [3].

This paper represents malware detection using machine learning algorithms and recommends each of them. In Sections 1 and 2 we review traditional and machine learning methods of detection of malicious software. Section 3 contains a detailed comparison table of machine learning algorithms and shows the practical part in Python. Finally, section 4 keeps the conclusion and final notes [4, 5].

2. Traditional method of detection of malicious software

Signature-based detection: In this approach, malware is detected based on certain signatures or file patterns. This traditional method allows the detection fastly known malicious software compared to other methods. Most of the antivirus programs are realized by using signature-based methods. Signature of malware generated and stored in the database of detection to verify the signature of an unknown file. If the signature of the file matches, then it is declared a malicious file, otherwise, it is a benign file. The main issue of this method is that the file signature changes even when one byte of the file is changed. Thus, for every modified and new malware must be generated a new signature. After that, the detector of malicious software can find a new signature. Also, the main disadvantage of this detection is that it takes a lot of time and effort to detect malware and extract different signatures for different types of malware [6].

Behavior-based malware detection: This method is based on program behavior monitoring and decision making whether it is malware or not. This method detected malicious software by its suspicious behavior difference from other typical programs. The behavior of malware determines its importance in this type, and sometimes different malware programs are found under the same signature [7]. The benefit compared to the traditional signature-based approach is that this method enables the detection of new malware without requiring human analysis and extraction of their signatures, as the behavior of new malware may align with known patterns of malicious behavior. However, it faces challenges related to time consumption and false positive rates [8].

Heuristic-based malware detection: This method studies the behavior of malware by using machine learning algorithms and intellectual data analysis. It solves many problems, which are existing in signature-based and behavior-based detection methods [9].

3. Machine learning methods and algorithms

Machine learning is modern technology, which is supposed to be learning machines based on experience. Machine learning can detect hidden patterns, which users can detect easily themselves [10]. The process begins by providing the machine learning algorithm with an input dataset. Then constructed a model, which can do accurate predictions for new datasets. Machine learning algorithms are divided into three main categories. Firstly, supervised learning, which includes machine learning by using a set of examples with their answers. The algorithm uses these examples to respond to any new input based on what it has learned. Secondly, an unsupervised model uses data sets without answers. Here, the algorithm classified input data based on the similarity between these values.

Analysis of malicious software includes the process of analyzing and understanding the malicious software for determination of its functionality, behavior, and potential impacts. Analysis of malware provides a clear view of malware in the code and byte string values, which is used by an attacker to steal information or modify the source code. In the process of analysis, various key features of the code are extracted, which provides information and functionality of malware activity in the system or network system [11].

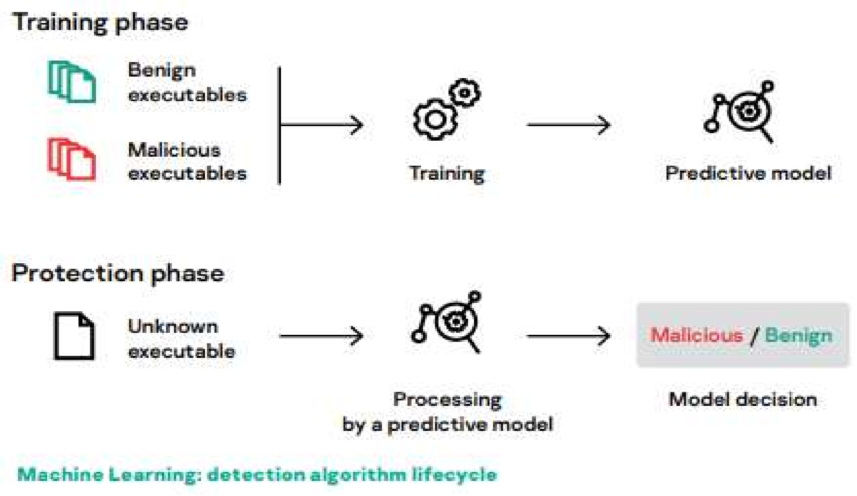


Figure 1: Machine learning: detection algorithm lifecycle

There are three types of the most frequently used methods [12], as shown in Fig. 2.

Static analysis: Static analysis of malicious software includes the study of malware or files without execution. This method of analysis allows users and analytics of security to understand structure, behavior, and potential threats, coming from malware, without the risk of infecting the system.

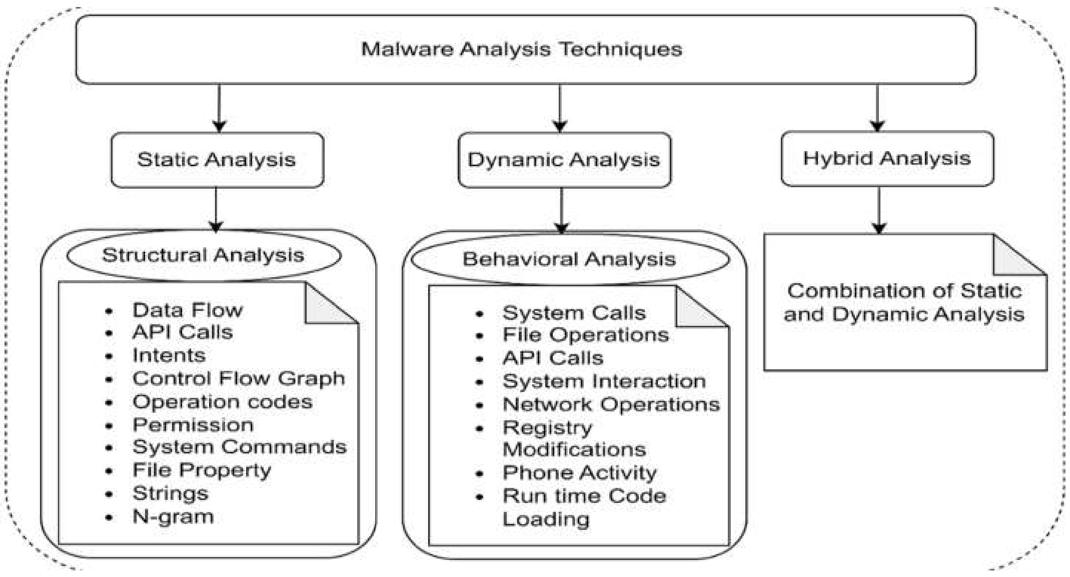


Figure 2: Taxonomy of Malware Analysis Techniques

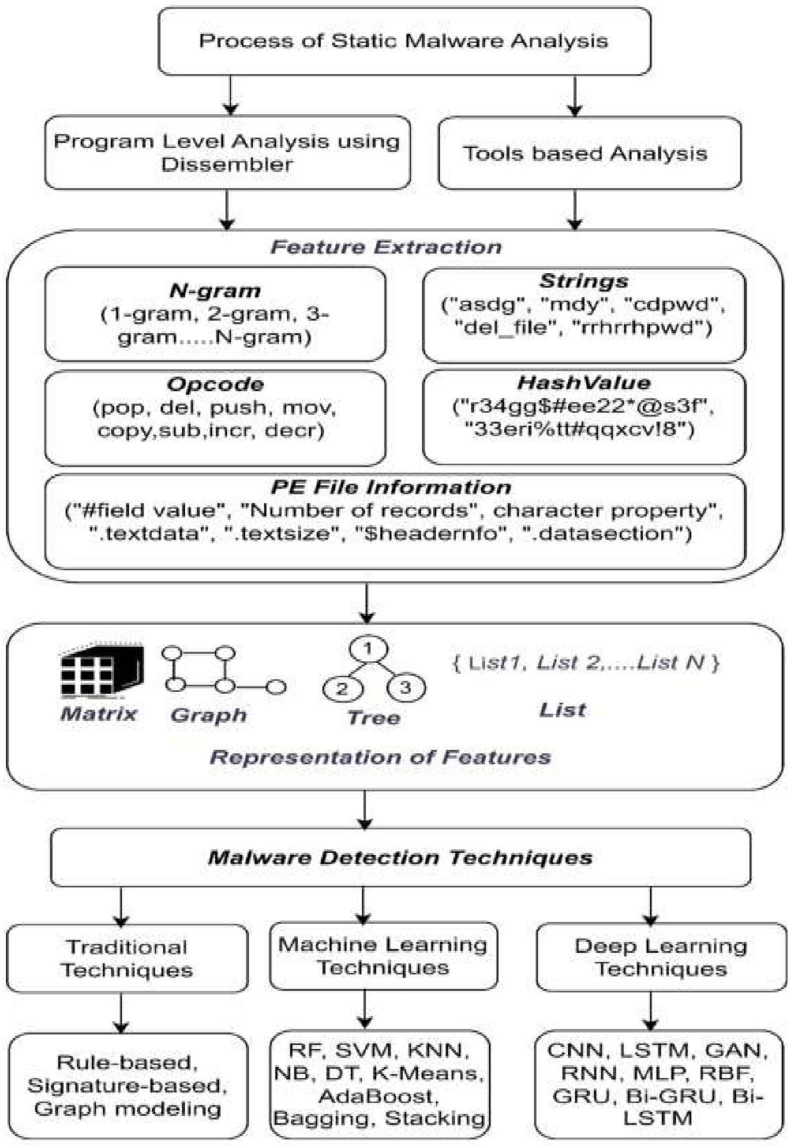
Static analysis can present valuable information about behavior, opportunities, and potential influence of malware on systems. This analysis is useful for the fast detection of known malware signatures and the extraction of important information from the code or files [13]. The functionality of malware can be analyzed by checking the internal malware code. It provides information about the identity of the malware, library, URL addresses, and programming languages as shown in Fig. 3. The process of static analysis executes faster and provides deeper knowledge about the malware execution path. However, it has drawbacks: it does not detect new variants of malware families or polymorphic malware specifically designed to evade static analysis [14].

Dynamic analysis: At the same time, malicious code is executed in a controlled environment to observe its behavior and understand its capabilities. Difference from the static analysis that researches the code without its execution [15]. Dynamic analysis provides real-time monitoring of how malware adapts and behaves in response to specific environmental conditions. Observing malicious software in action, analysts can gain insight into its purpose, distribution methods, and the potential damage it can cause [16]. However, dynamic analysis carries a certain risk, since malware is actively working and there is a possibility of unforeseen consequences. This is why, it is significant to carry out dynamic analysis in a controlled and isolated environment to minimize the potential impact on the host system and the network as a whole [17]. The process of dynamic analysis is shown in Fig. 4.

Hybrid malware analysis: Hybrid malware analysis, also known as combined analysis or integrated analysis, is an approach combining several methods, such as static analysis, dynamic analysis, and behavior analysis, to get a complete understanding of malware. Using the strong sides of different analysis methods, hybrid analysis is aimed at overcoming the limitations of individual methods and providing a more reliable and accurate assessment of the behavior, capabilities, and potential impact of malware [18].

Machine learning is the subset of AI, used for tasks of detection and classification in different areas. Algorithms of machine learning, such as RF, NB, KNN, SVM, and DT, are widely used for the detection and classification of malicious programs.

Naive Bayes (NB) is an algorithm of classification based on supervised learning. It works in the base of probability functions, in which each attribute belongs to a specific class. A strong assumption is required that the attributes are conditionally independent. An assumption, which is used in algorithm NB simplifies the probability calculation. It calculates whether a data point can fall into a certain class or not. It can accurately predict test data sets for binary and many class datasets.



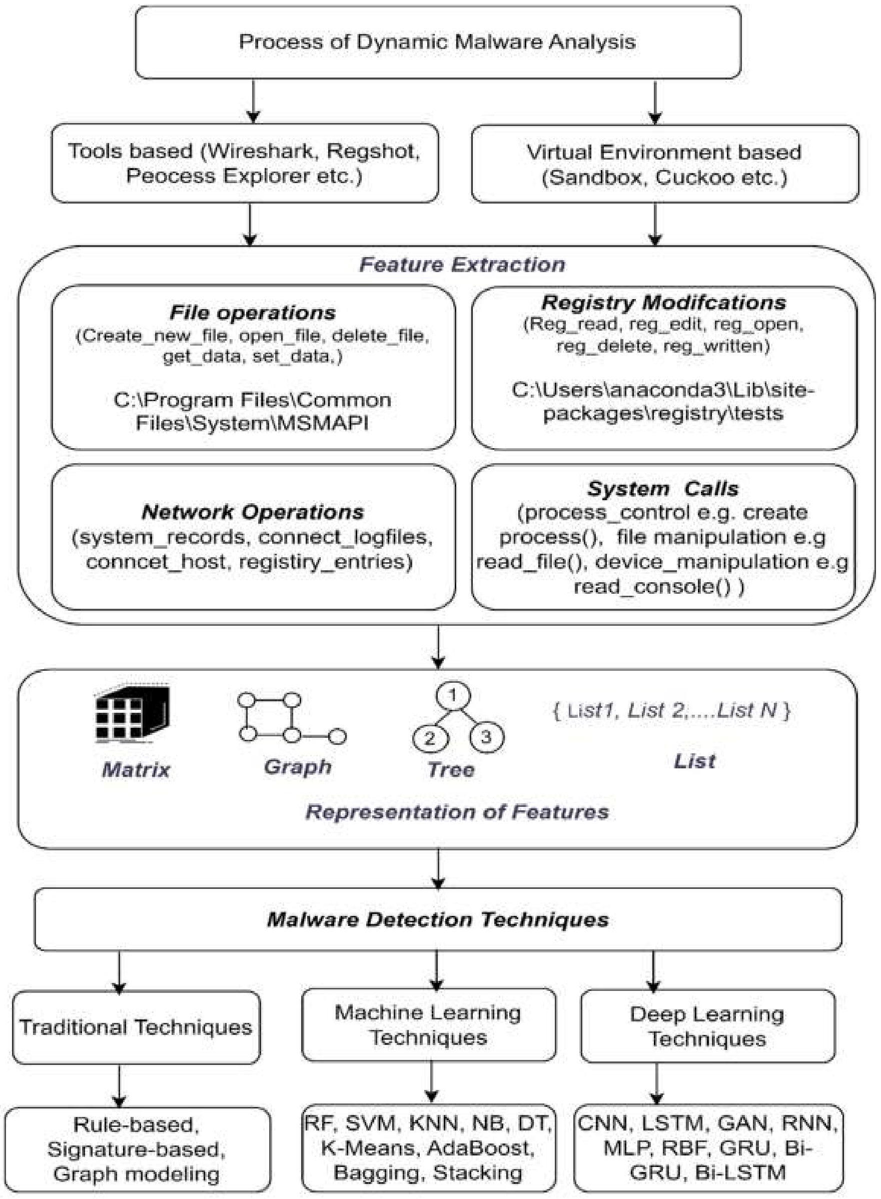
Process of Static Malware Analysis
Program Level Analysis using Dissembler
Tools based Analysis
Feature Extraction
N-gram (1-gram, 2-gram, 3-gram.....N-gram)
Strings ("asdg", "mdy", "cdpwd", "del_file", "rrhrrhpwd")
Opcode (pop, del, push, mov, copy,sub,incr, decr)
HashValue ("r34gg$#ee22*@s3f", "33eri%tt#qqxcv!8")
PE File Information ("#field value", "Number of records", character property", ".textdata", ".textsize", "$headernfo", ".datasection")
1
2
3
{ List1, List 2,....List N }
Matrix
Graph
Tree
List
Representation of Features
Malware Detection Techniques
Traditional Techniques
Machine Learning Techniques
Deep Learning Techniques
Rule-based, Signature-based, Graph modeling
RF, SVM, KNN, NB, DT, K-Means, AdaBoost, Bagging, Stacking
CNN, LSTM, GAN, RNN, MLP, RBF, GRU, Bi-GRU, Bi-LSTM

Figure 3: Schematic diagram of static analysis for malware detection [19]

It is effective and scales well, therefore, it works well for small irrelevant data sets. It used the NB theorem for the decomposition of conditional probability. However, NB gives poor performance because attributes are highly correlated with each other, and it is believed that attributes should be independent.

K-Nearest Neighbors is a nonparametric supervised algorithm, which means that it doesn't make any assumptions about the underlying distribution of the data. It works based on the proximity of the similarity of objects or searching for Nearest Neighbors in the specified set and uses the majority of votes to classify a new data point. KNN defines the similar characteristics of new points on the base of previously saved data points, using the Euclidean distance between two data points. This algorithm is known as lazy learning because it does not require parameter adjustment and works without training the model. Instead of training the model, it takes all the data points at the time of forecasting. However, it has some disadvantages, such as high cost, low speed, and less scalability for large data sets.

Decision Tree is a classification of supervised learning, used for tasks of classification and regression. It can also work with numbers and category data. It follows the tree model approach, in which the data points are divided into two branches and a conclusion about existing features in each node of the tree [20].



Process of Dynamic Malware Analysis
Tools based (Wireshark, Regshot, Peocess Explorer etc.)
Virtual Environment based (Sandbox, Cuckoo etc.)
Feature Extraction
File operations
(Create_new_file, open_file, delete_file, get_data, set_data,)
C:\Program Files\Common Files\System\MSMAPI
Registry Modifcations
(Reg_read, reg_edit, reg_open, reg_delete, reg_written)
C:\Users\anaconda3\Lib\site-packages\registry\tests
Network Operations
(system_records, connect_logfiles, conncet_host, registiry_entries)
System Calls
(process_control e.g. create process(), file manipulation e.g read_file(), device_manipulation e.g read_console())
Matrix
Graph
Tree
{ List1, List 2,....List N }
List
Representation of Features
Malware Detection Techniques
Traditional Techniques
Machine Learning Techniques
Deep Learning Techniques
Rule-based, Signature-based, Graph modeling
RF, SVM, KNN, NB, DT, K-Means, AdaBoost, Bagging, Stacking
CNN, LSTM, GAN, RNN, MLP, RBF, GRU, Bi-GRU, Bi-LSTM

Figure 4. Schematic diagram of dynamic analysis for malware detection [19]

Table 1
Comparison of malware analysis approaches.

Methods	Pros	Cons
Static analysis	Fast and safe. High accuracy. Low memory consumption.	It is impossible to analyze confusing and encrypting malware. Unknown malware cannot be detected.
Dynamic analysis	Can analyze confusing and encrypting malware. Also, it can detect known and unknown malware.	Consumes a high number of resources and is unsafe and slow.
Hybrid analysis	The result is more accurate than static and dynamic analysis.	Higher complexity and more time are used.

A decision tree is designed for the creation of a learning flowchart structure, which can be used for the classification of classes or values targeted variables on the base of decision-making rules, which are extracted from previous data. It can work well with huge and noisy data sets compared to models of KNN and SVM [21].

Table 2
Comparison of machine learning algorithms for malware analysis

Machine learning methods	Advantages	Disadvantages	Application
NB	Easy to implement, fast operation, good performance on small amounts of data	It is impossible to analyze confusing and encrypting malware. Unknown malware cannot be detected.	Suitable for text classification and spam analysis
K-Nearest Neighbors	Simplicity and intuitiveness, no assumptions about data distribution	Consumes a high number of resources and is unsafe and slow.	Good for small to medium datasets with clear classes
Decision Tree	Easy to interpret and visualize, no need to scale data	Higher complexity and more time are used.	Suitable for classification and regression, as well as data preprocessing
Support Vector Machine	High precision, effective in high-dimensional spaces	Long training time on large datasets, difficulty in parameter tuning	Good for classification and regression problems, especially with clear boundaries between classes
Random Forest	High resistance to overfitting, good accuracy, ability to work with missing values	More complex interpretation of results, memory consumption, and training time	Widely used in classification and regression, especially when there are a large number of features.

Support Vector Machine is a powerful supervised model, which is used as a task of classification so it is for regression. The goal of SVM is to create the best hyperplane, which exactly divides data sets of one class from another class of data sets of learning and testing. In the data sets of SVM a set of data points separated by a line called a hyperplane is specified, which is used for classification of data sets classes. This is an effective classifier for effectively processing a large set of data. It is used for problem decisions of linear functions in multidimensional feature spaces. However, this does not work well in the case of big data sets with noisy data sets and requires more time for learning [21].

Random Forest is a classifier based on ensemble learning, which consists of several decision trees, that work parallel, a forest that is constructed and is called decision tree ensemble of the decision tree. Each tree in the ensemble model contains data samples taken from the training data with replacement, known as a bootstrap sample, which is a set of learning models to improve the overall results of the models [20]. The decision about targeted classes was adopted by a majority of votes in Random Forest. This is an easy, flexible, and simple algorithm, which gives the best result most of the time, even without setting the hyperparameters. This reduces over-training, which leads to improving the accuracy of the decision tree [22]. It works for categorical and continuous data. However, the speed of Random Forest is low, because it requires more time on learning a set of decision trees for constructing strong classificatory [21].

4. Machine learning malware detection and its application

Confusion matrix is a matrix that is constructed for every classification model prediction and it shows the number of test cases correctly and incorrectly classified. It looks like this, which is shown in Table 3 (considering 1 (positive) and 0 (negative) are the target classes):

Table 3
Metrics of classification

	Actual 0	Actual 1
Predicted 0	True Negatives (TN)	False Negatives (FN)
Predicted 1	False Positives (FP)	True Positives (TP)

TN is the number of negative cases correctly classified; *TP* is the number of positive cases correctly classified; *FN* is the number of positive cases incorrectly classified as negative; *FP* is the number of negative cases incorrectly classified as positive.

Accuracy is the simplest indicator, which can determine the number of correct classification test examples divided by the total number of test examples.

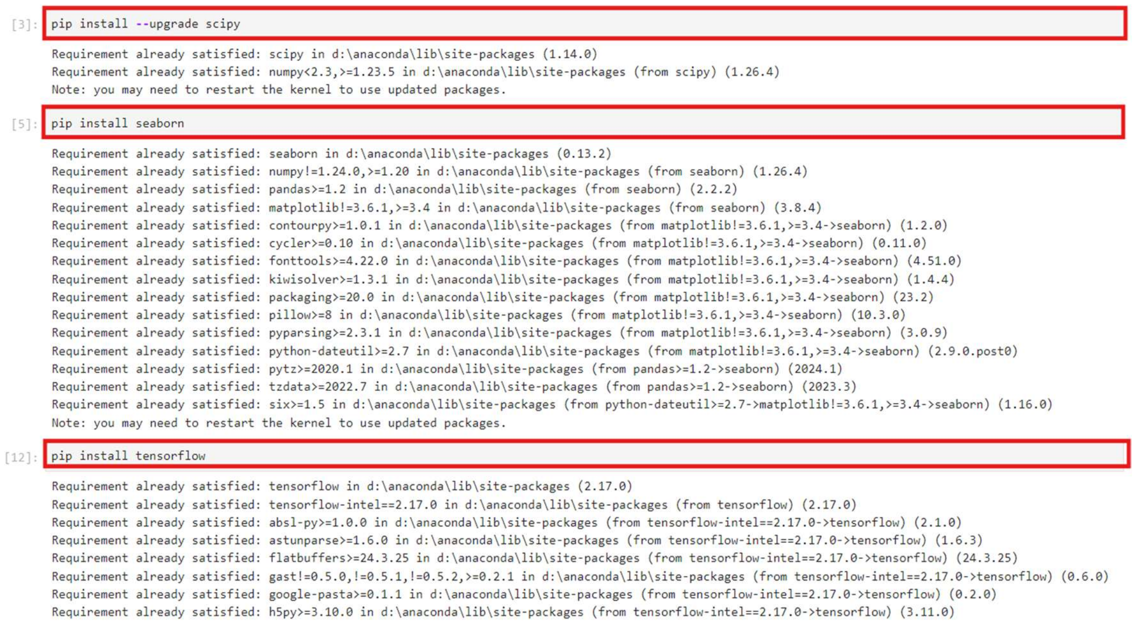
$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

It can implement many general issues, but it is not useful when we are talking about unbalanced data sets. For example, if we detect fraud in bank datasets, the ratio of fraud cases to non-fraud cases can be 1:99. In this case, if we use the accuracy, the model can prove to be 99% accurate, predicting all test cases as non-fraud. This way, accuracy is a false indicator of the model's performance, and in this case, the metric is required that can focus on fraud data points.

First of all, for doing research, necessary libraries, such as scipy, seaborn, and Tensorflow to analyze the data, build graphs, and study neural networks to find and classify images, which is shown in Fig.5.

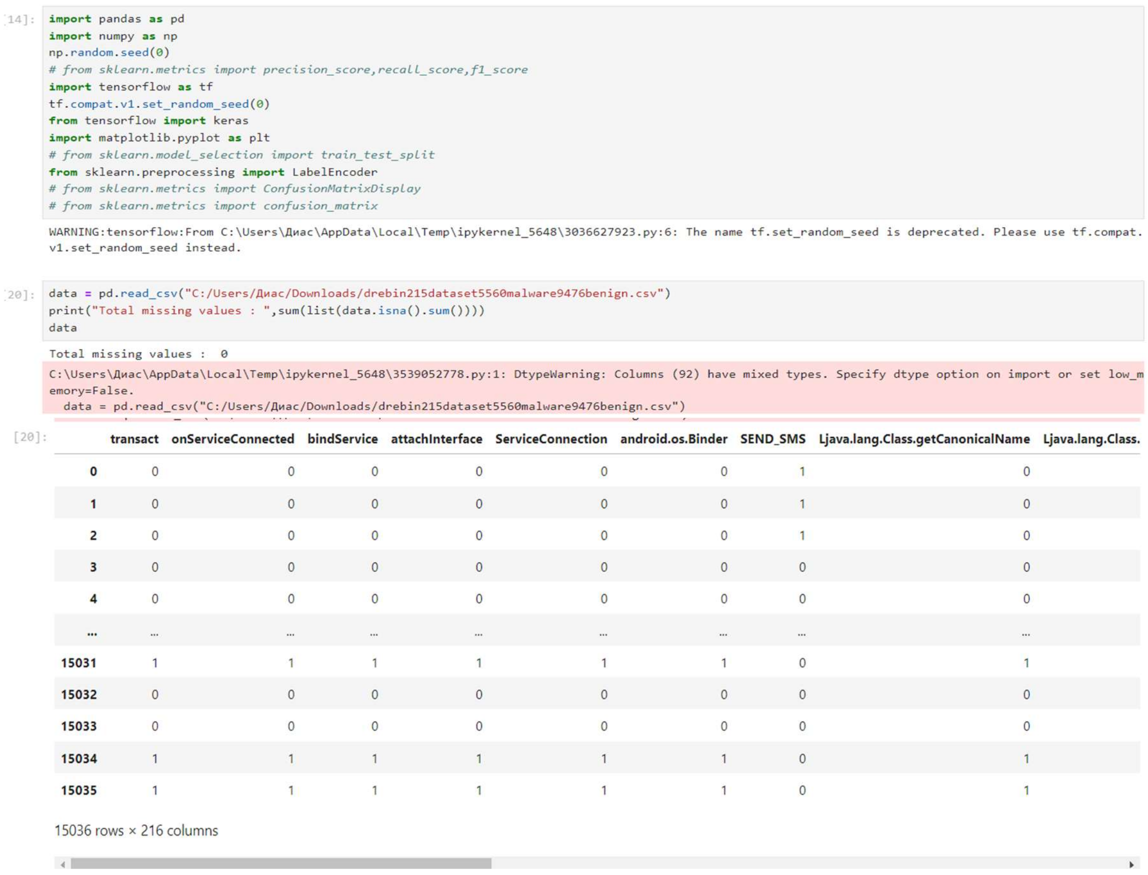
After that, this code which is shown below imports pandas, numpy, tensorflow, keras, and matplotlib libraries for working with data, creating and training machine learning models, and visualization. Fixed random number generator seeds are set to ensure reproducibility of results. Some tools for data preprocessing and model evaluation are commented out and are not currently used.

This code loads data from a CSV file using pandas and stores it in the variable data. It then prints the total number of missing values in this dataset, using the isna() method to determine the missing values and the sum() function to count them. Finally, the code prints the DataFrame data itself.



```
[3]: pip install --upgrade scipy

Requirement already satisfied: scipy in d:\anaconda\lib\site-packages (1.14.0)
Requirement already satisfied: numpy<2.3,>=1.23.5 in d:\anaconda\lib\site-packages (from scipy) (1.26.4)
Note: you may need to restart the kernel to use updated packages.

[5]: pip install seaborn

Requirement already satisfied: seaborn in d:\anaconda\lib\site-packages (0.13.2)
Requirement already satisfied: numpy!=1.24.0,>=1.20 in d:\anaconda\lib\site-packages (from seaborn) (1.26.4)
Requirement already satisfied: pandas>=1.2 in d:\anaconda\lib\site-packages (from seaborn) (2.2.2)
Requirement already satisfied: matplotlib!=3.6.1,>=3.4 in d:\anaconda\lib\site-packages (from seaborn) (3.8.4)
Requirement already satisfied: contourpy>=1.0.1 in d:\anaconda\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (1.2.0)
Requirement already satisfied: cycler>=0.10 in d:\anaconda\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in d:\anaconda\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (4.51.0)
Requirement already satisfied: kiwisolver>=1.3.1 in d:\anaconda\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (1.4.4)
Requirement already satisfied: packaging>=20.0 in d:\anaconda\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (23.2)
Requirement already satisfied: pillow>=8 in d:\anaconda\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (10.3.0)
Requirement already satisfied: pyparsing>=2.3.1 in d:\anaconda\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in d:\anaconda\lib\site-packages (from matplotlib!=3.6.1,>=3.4->seaborn) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in d:\anaconda\lib\site-packages (from pandas>=1.2->seaborn) (2024.1)
Requirement already satisfied: tzdata>=2022.7 in d:\anaconda\lib\site-packages (from pandas>=1.2->seaborn) (2023.3)
Requirement already satisfied: six>=1.5 in d:\anaconda\lib\site-packages (from python-dateutil>=2.7->matplotlib!=3.6.1,>=3.4->seaborn) (1.16.0)
Note: you may need to restart the kernel to use updated packages.

[12]: pip install tensorflow

Requirement already satisfied: tensorflow in d:\anaconda\lib\site-packages (2.17.0)
Requirement already satisfied: tensorflow-intel==2.17.0 in d:\anaconda\lib\site-packages (from tensorflow) (2.17.0)
Requirement already satisfied: absl-py>=1.0.0 in d:\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (2.1.0)
Requirement already satisfied: astunparse>=1.6.0 in d:\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (1.6.3)
Requirement already satisfied: flatbuffers>=24.3.25 in d:\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (24.3.25)
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 in d:\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (0.6.0)
Requirement already satisfied: google-pasta>=0.1.1 in d:\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (0.2.0)
Requirement already satisfied: h5py>=3.10.0 in d:\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (3.11.0)
```

Figure 5: Installation of the necessary library for the research



```
14]: import pandas as pd
     import numpy as np
     np.random.seed(0)
     # from sklearn.metrics import precision_score,recall_score,f1_score
     import tensorflow as tf
     tf.compat.v1.set_random_seed(0)
     from tensorflow import keras
     import matplotlib.pyplot as plt
     # from sklearn.model_selection import train_test_split
     from sklearn.preprocessing import LabelEncoder
     # from sklearn.metrics import ConfusionMatrixDisplay
     # from sklearn.metrics import confusion_matrix

     WARNING:tensorflow:From C:\Users\Диас\AppData\Local\Temp\ipykernel_5648\3036627923.py:6: The name tf.set_random_seed is deprecated. Please use tf.compat.
     v1.set_random_seed instead.

20]: data = pd.read_csv("C:/Users/Диас/Downloads/drebin215dataset5560malware9476benign.csv")
     print("Total missing values : ",sum(list(data.isna().sum())))
     data

     Total missing values :  0
     C:\Users\Диас\AppData\Local\Temp\ipykernel_5648\3539052778.py:1: DtypeWarning: Columns (92) have mixed types. Specify dtype option on import or set low_m
     emory=False.
       data = pd.read_csv("C:/Users/Диас/Downloads/drebin215dataset5560malware9476benign.csv")
```

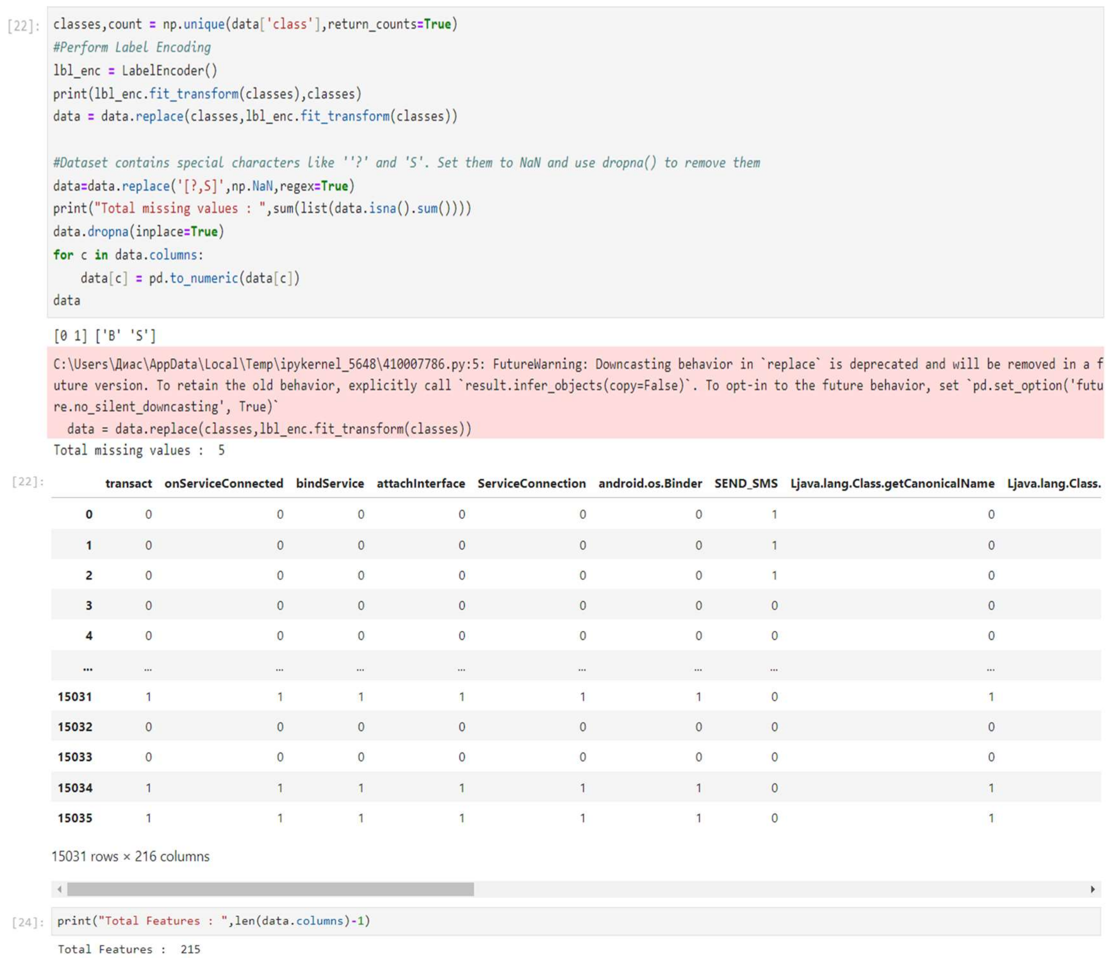
[20]:

	transact	onServiceConnected	bindService	attachInterface	ServiceConnection	android.os.Binder	SEND_SMS	Ljava.lang.Class.getCanonicalName	Ljava.lang.Class.
0	0	0	0	0	0	0	1	0	
1	0	0	0	0	0	0	1	0	
2	0	0	0	0	0	0	1	0	
3	0	0	0	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	
15031	1	1	1	1	1	1	0	1	
15032	0	0	0	0	0	0	0	0	
15033	0	0	0	0	0	0	0	0	
15034	1	1	1	1	1	1	0	1	
15035	1	1	1	1	1	1	0	1	

15036 rows × 216 columns

Figure 6: Visualization of code, which has missing values

This code parses the unique values and their counts in the “class” column, performs label encoding using “LabelEncoder”, replaces special characters with “NaN”, removes rows with missing values, converts all columns to numeric data types, and outputs the total number of features in the DataFrame after all transformations.



```
[22]: classes,count = np.unique(data['class'],return_counts=True)
      #Perform Label Encoding
      lbl_enc = LabelEncoder()
      print(lbl_enc.fit_transform(classes),classes)
      data = data.replace(classes,lbl_enc.fit_transform(classes))

      #Dataset contains special characters like ''?' and 'S'. Set them to NaN and use dropna() to remove them
      data=data.replace('[?,S]',np.NaN,regex=True)
      print("Total missing values : ",sum(list(data.isna().sum())))
      data.dropna(inplace=True)
      for c in data.columns:
          data[c] = pd.to_numeric(data[c])
      data
```

```
[0 1] ['B' 'S']
C:\Users\Диас\AppData\Local\Temp\ipykernel_5648\410007786.py:5: FutureWarning: Downcasting behavior in `replace` is deprecated and will be removed in a f
uture version. To retain the old behavior, explicitly call `result.infer_objects(copy=False)`. To opt-in to the future behavior, set `pd.set_option('futu
re.no_silent_downcasting', True)`
  data = data.replace(classes,lbl_enc.fit_transform(classes))
Total missing values :  5
```

[22]:

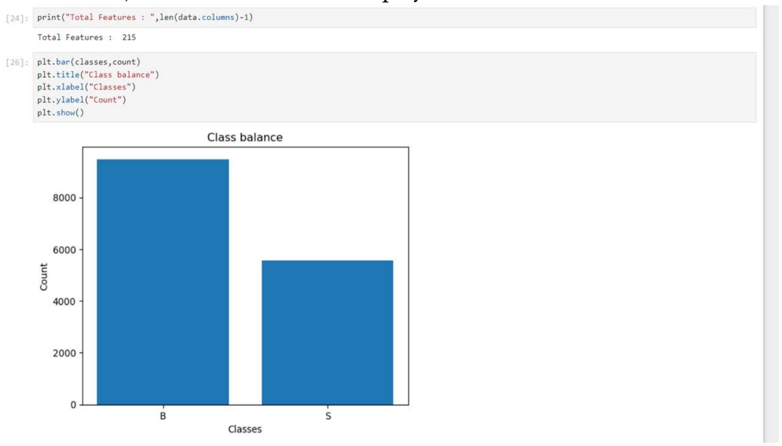
	transact	onServiceConnected	bindService	attachInterface	ServiceConnection	android.os.Binder	SEND_SMS	Ljava.lang.Class.getCanonicalName	Ljava.lang.Class.
0	0	0	0	0	0	0	1	0	
1	0	0	0	0	0	0	1	0	
2	0	0	0	0	0	0	1	0	
3	0	0	0	0	0	0	0	0	
4	0	0	0	0	0	0	0	0	
...	...	...	...	...	...	...	...	...	
15031	1	1	1	1	1	1	0	1	
15032	0	0	0	0	0	0	0	0	
15033	0	0	0	0	0	0	0	0	
15034	1	1	1	1	1	1	0	1	
15035	1	1	1	1	1	1	0	1	

15031 rows × 216 columns

```
[24]: print("Total Features : ",len(data.columns)-1)

      Total Features :  215
```

Figure 7: Data preprocessing: label encoding, cleaning, and data type conversion

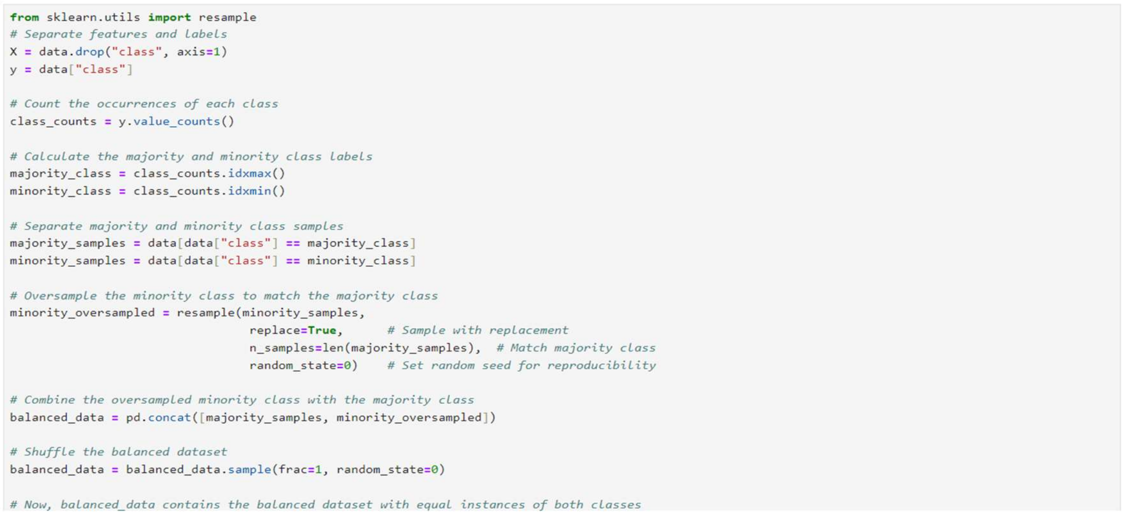
The code, which is shown in Fig.8 prints the total number of features in the DataFrame (excluding the class column), and then creates and displays a bar chart of the class distribution, showing the number of instanes for each class.



```
[24]: print("Total Features : ",len(data.columns)-1)

      Total Features :  215

[26]: plt.bar(classes,count)
      plt.title("Class balance")
      plt.xlabel("Classes")
      plt.ylabel("Count")
      plt.show()
```

Figure 8: Visualization of class distribution

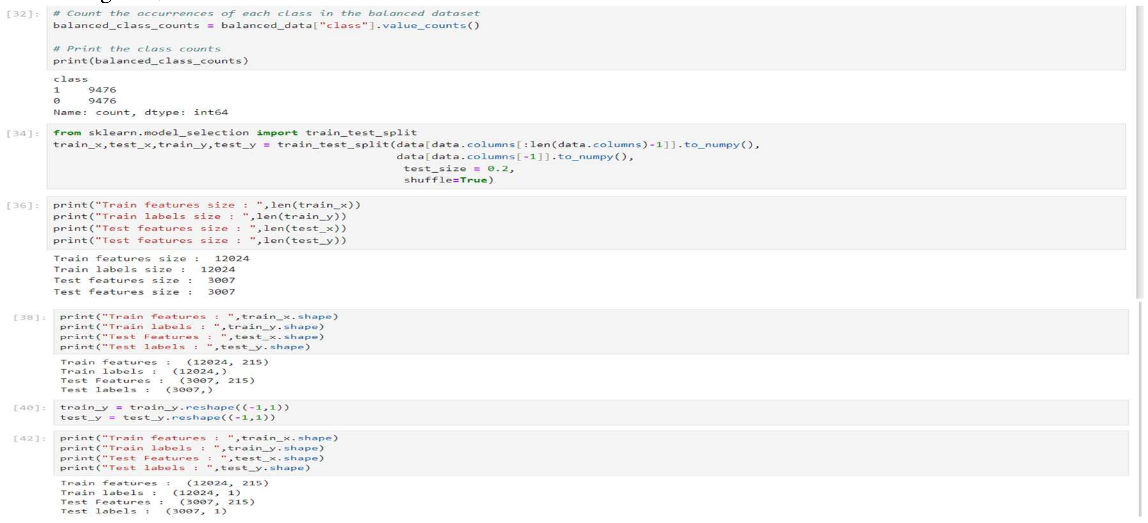
The code shown in below balances a dataset to solve the class imbalance problem. It splits the data into features and labels, identifies the largest and smallest classes, oversamples the minority class to equalize the number of instances with the majority class, and then merges and shuffles the balanced dataset.



```
from sklearn.utils import resample
# Separate features and labels
X = data.drop("class", axis=1)
y = data["class"]

# Count the occurrences of each class
class_counts = y.value_counts()

# Calculate the majority and minority class labels
majority_class = class_counts.idxmax()
minority_class = class_counts.idxmin()

# Separate majority and minority class samples
majority_samples = data[data["class"] == majority_class]
minority_samples = data[data["class"] == minority_class]

# Oversample the minority class to match the majority class
minority_oversampled = resample(minority_samples,
                                replace=True,      # Sample with replacement
                                n_samples=len(majority_samples),  # Match majority class
                                random_state=0)    # Set random seed for reproducibility

# Combine the oversampled minority class with the majority class
balanced_data = pd.concat([majority_samples, minority_oversampled])

# Shuffle the balanced dataset
balanced_data = balanced_data.sample(frac=1, random_state=0)

# Now, balanced_data contains the balanced dataset with equal instances of both classes
```

Figure 9: Class Balancing: Increasing the Sample of the Minority Class

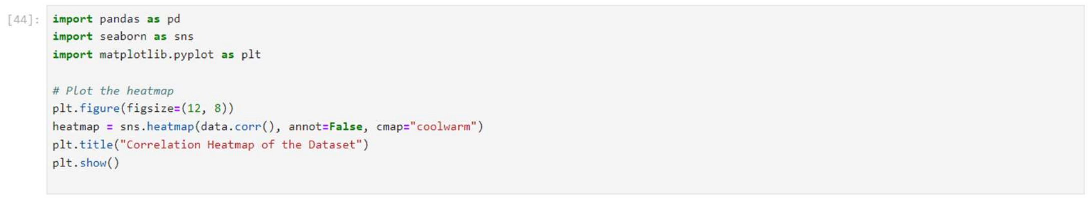
The code below balances the classes in a dataset by counting the number of instances of each class and outputting that data. It then splits the data into training and testing sets, where 80% is used to train the model and 20% is used for testing, and outputs the sizes of both sets.



```
[32]: # Count the occurrences of each class in the balanced dataset
      balanced_class_counts = balanced_data["class"].value_counts()

      # Print the class counts
      print(balanced_class_counts)

      class
      1    9476
      0    9476
      Name: count, dtype: int64

[34]: from sklearn.model_selection import train_test_split
      train_x,test_x,train_y,test_y = train_test_split(data[data.columns[:len(data.columns)-1]].to_numpy(),
                                                       data[data.columns[-1]].to_numpy(),
                                                        test_size = 0.2,
                                                        shuffle=True)

[36]: print("Train features size : ",len(train_x))
      print("Train labels size : ",len(train_y))
      print("Test features size : ",len(test_x))
      print("Test features size : ",len(test_y))

      Train features size :  12024
      Train labels size :  12024
      Test features size :  3007
      Test features size :  3007

[38]: print("Train features : ",train_x.shape)
      print("Train labels : ",train_y.shape)
      print("Test Features : ",test_x.shape)
      print("Test labels : ",test_y.shape)

      Train features :  (12024, 215)
      Train labels :  (12024,)
      Test Features :  (3007, 215)
      Test labels :  (3007,)

[40]: train_y = train_y.reshape((-1,1))
      test_y = test_y.reshape((-1,1))

[42]: print("Train features : ",train_x.shape)
      print("Train labels : ",train_y.shape)
      print("Test Features : ",test_x.shape)
      print("Test labels : ",test_y.shape)

      Train features :  (12024, 215)
      Train labels :  (12024, 1)
      Test Features :  (3007, 215)
      Test labels :  (3007, 1)
```

Figure 10: Counting classes and dividing data into training and test samples

Below is the code that builds a heat map of correlations between features in a dataset. It uses the seaborn and matplotlib libraries for visualization, displaying correlation coefficients as a color map. The heat map helps to identify dependencies and relationships between different features.



```
[44]: import pandas as pd
      import seaborn as sns
      import matplotlib.pyplot as plt

      # Plot the heatmap
      plt.figure(figsize=(12, 8))
      heatmap = sns.heatmap(data.corr(), annot=False, cmap="coolwarm")
      plt.title("Correlation Heatmap of the Dataset")
      plt.show()
```

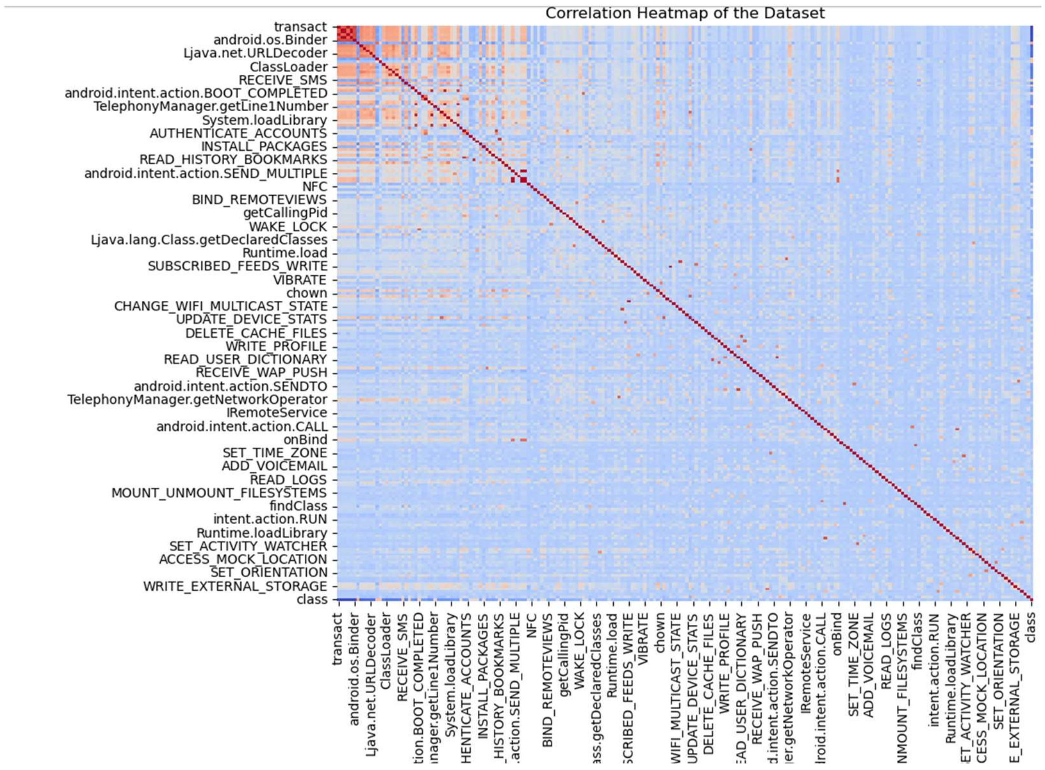
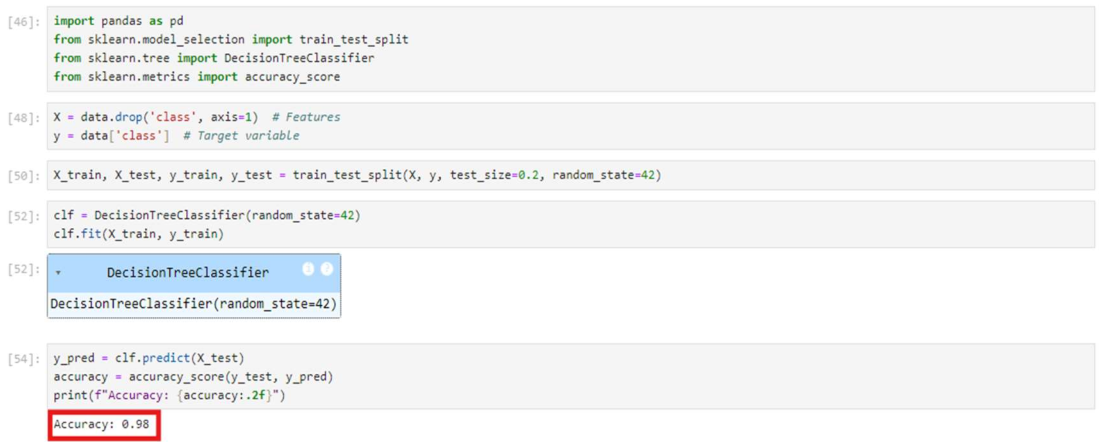


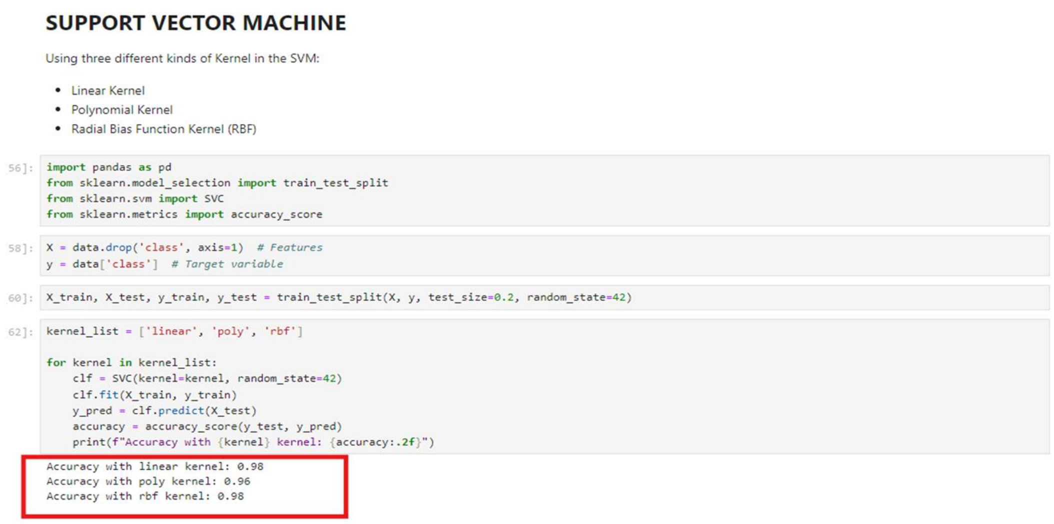
Figure 11: Correlation Heatmap of the Dataset

The code below trains a decision tree classifier on the training data, makes predictions on the test data, and computes the accuracy of the model.



```
[46]: import pandas as pd
      from sklearn.model_selection import train_test_split
      from sklearn.tree import DecisionTreeClassifier
      from sklearn.metrics import accuracy_score

[48]: X = data.drop('class', axis=1)  # Features
      y = data['class']  # Target variable

[50]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

[52]: clf = DecisionTreeClassifier(random_state=42)
      clf.fit(X_train, y_train)

[52]: DecisionTreeClassifier
      DecisionTreeClassifier(random_state=42)

[54]: y_pred = clf.predict(X_test)
      accuracy = accuracy_score(y_test, y_pred)
      print(f"Accuracy: {accuracy:.2f}")

      Accuracy: 0.98
```

Figure 12: Training and evaluation of a decision tree model



SUPPORT VECTOR MACHINE

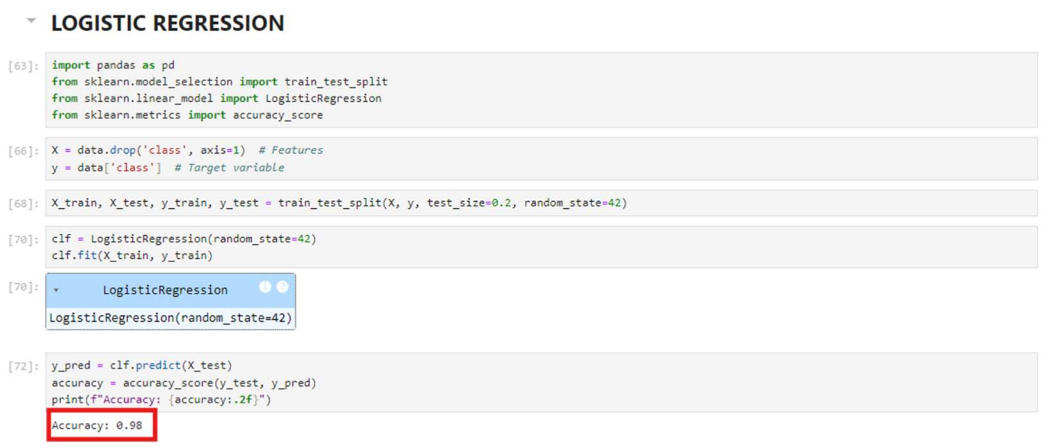
Using three different kinds of Kernel in the SVM:

- Linear Kernel
- Polynomial Kernel
- Radial Bias Function Kernel (RBF)

```
56]: import pandas as pd
     from sklearn.model_selection import train_test_split
     from sklearn.svm import SVC
     from sklearn.metrics import accuracy_score

58]: X = data.drop('class', axis=1)  # Features
     y = data['class']  # Target variable

60]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

62]: kernel_list = ['linear', 'poly', 'rbf']

     for kernel in kernel_list:
         clf = SVC(kernel=kernel, random_state=42)
         clf.fit(X_train, y_train)
         y_pred = clf.predict(X_test)
         accuracy = accuracy_score(y_test, y_pred)
         print(f"Accuracy with {kernel} kernel: {accuracy:.2f}")
```

```
Accuracy with linear kernel: 0.98
Accuracy with poly kernel: 0.96
Accuracy with rbf kernel: 0.98
```

Figure 13: Training and evaluation of a support vector machine model

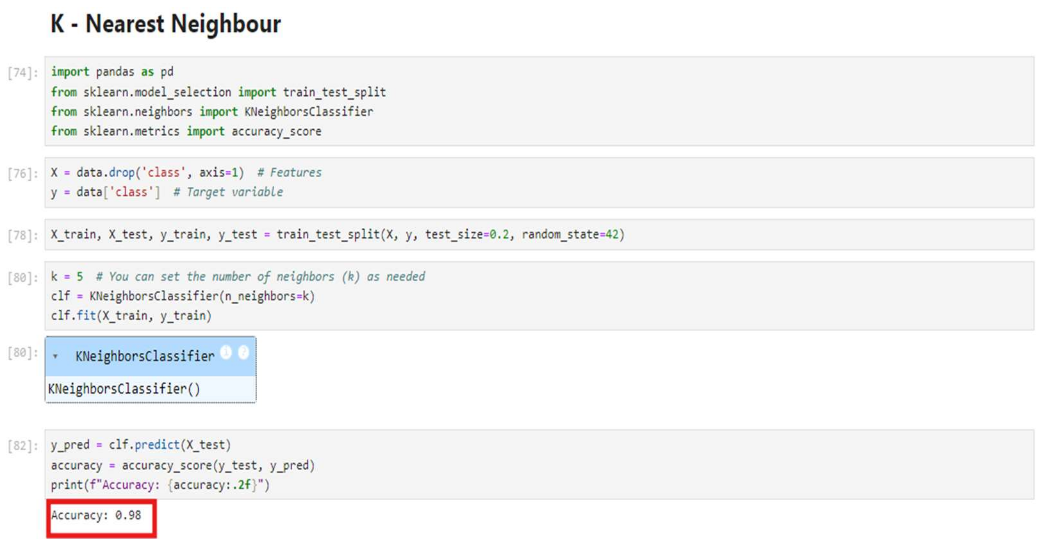


LOGISTIC REGRESSION

```
[63]: import pandas as pd
      from sklearn.model_selection import train_test_split
      from sklearn.linear_model import LogisticRegression
      from sklearn.metrics import accuracy_score

[66]: X = data.drop('class', axis=1)  # Features
      y = data['class']  # Target variable

[68]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

[70]: clf = LogisticRegression(random_state=42)
      clf.fit(X_train, y_train)

[70]: LogisticRegression
      LogisticRegression(random_state=42)

[72]: y_pred = clf.predict(X_test)
      accuracy = accuracy_score(y_test, y_pred)
      print(f"Accuracy: {accuracy:.2f}")
```

```
Accuracy: 0.98
```

Figure 14: Training and evaluation of a logistic regression model

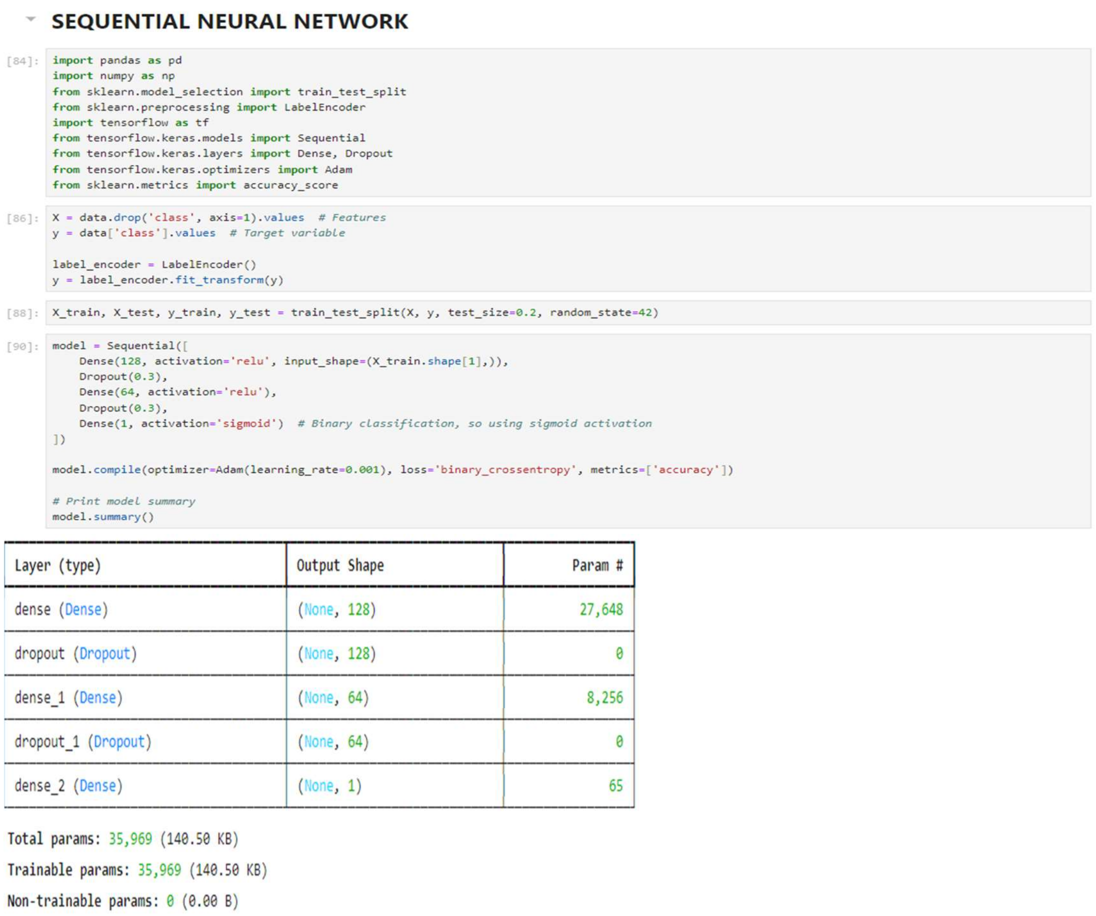


K - Nearest Neighbour

```
[74]: import pandas as pd
      from sklearn.model_selection import train_test_split
      from sklearn.neighbors import KNeighborsClassifier
      from sklearn.metrics import accuracy_score

[76]: X = data.drop('class', axis=1)  # Features
      y = data['class']  # Target variable

[78]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

[80]: k = 5  # You can set the number of neighbors (k) as needed
      clf = KNeighborsClassifier(n_neighbors=k)
      clf.fit(X_train, y_train)

[80]: KNeighborsClassifier
      KNeighborsClassifier()

[82]: y_pred = clf.predict(X_test)
      accuracy = accuracy_score(y_test, y_pred)
      print(f"Accuracy: {accuracy:.2f}")
```

```
Accuracy: 0.98
```

Figure 15: Training and evaluation of a K-Nearest Neighbor model.



SEQUENTIAL NEURAL NETWORK

```
[84]: import pandas as pd
      import numpy as np
      from sklearn.model_selection import train_test_split
      from sklearn.preprocessing import LabelEncoder
      import tensorflow as tf
      from tensorflow.keras.models import Sequential
      from tensorflow.keras.layers import Dense, Dropout
      from tensorflow.keras.optimizers import Adam
      from sklearn.metrics import accuracy_score
```

```
[86]: X = data.drop('class', axis=1).values  # Features
      y = data['class'].values  # Target variable

      label_encoder = LabelEncoder()
      y = label_encoder.fit_transform(y)
```

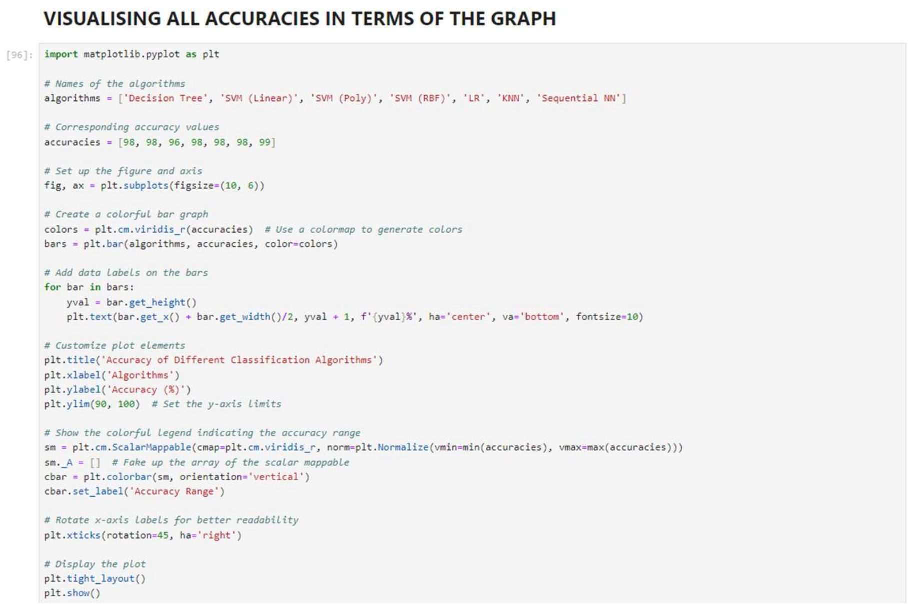
```
[88]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
[90]: model = Sequential([
          Dense(128, activation='relu', input_shape=(X_train.shape[1],)),
          Dropout(0.3),
          Dense(64, activation='relu'),
          Dropout(0.3),
          Dense(1, activation='sigmoid')  # Binary classification, so using sigmoid activation
      ])

      model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy', metrics=['accuracy'])

      # Print model summary
      model.summary()
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                  ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ dense (Dense)                 │ (None, 128)            │        27,648 │
├───────────────────────────────┼────────────────────────┼───────────────┤
│ dropout (Dropout)             │ (None, 128)            │             0 │
├───────────────────────────────┼────────────────────────┼───────────────┤
│ dense_1 (Dense)               │ (None, 64)             │         8,256 │
├───────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_1 (Dropout)           │ (None, 64)             │             0 │
├───────────────────────────────┼────────────────────────┼───────────────┤
│ dense_2 (Dense)               │ (None, 1)              │            65 │
└───────────────────────────────┴────────────────────────┴───────────────┘

 Total params: 35,969 (140.50 KB)

 Trainable params: 35,969 (140.50 KB)

 Non-trainable params: 0 (0.00 B)
```

Figure 16: Training and evaluation of a Sequential neural network model

The code, which is shown in Fig. 12 creates a graph showing the accuracy of different classification algorithms. It plots a bar chart using a color map to show the accuracy and adds labels to each bar. It also displays a color scale indicating the accuracy range.



VISUALISING ALL ACCURACIES IN TERMS OF THE GRAPH

```
[96]: import matplotlib.pyplot as plt

      # Names of the algorithms
      algorithms = ['Decision Tree', 'SVM (Linear)', 'SVM (Poly)', 'SVM (RBF)', 'LR', 'KNN', 'Sequential NN']

      # Corresponding accuracy values
      accuracies = [98, 98, 96, 98, 98, 98, 99]

      # Set up the figure and axis
      fig, ax = plt.subplots(figsize=(10, 6))

      # Create a colorful bar graph
      colors = plt.cm.viridis_r(accuracies)  # Use a colormap to generate colors
      bars = plt.bar(algorithms, accuracies, color=colors)

      # Add data labels on the bars
      for bar in bars:
          yval = bar.get_height()
          plt.text(bar.get_x() + bar.get_width()/2, yval + 1, f'{yval}%', ha='center', va='bottom', fontsize=10)

      # Customize plot elements
      plt.title('Accuracy of Different Classification Algorithms')
      plt.xlabel('Algorithms')
      plt.ylabel('Accuracy (%)')
      plt.ylim(90, 100)  # Set the y-axis limits

      # Show the colorful legend indicating the accuracy range
      sm = plt.cm.ScalarMappable(cmap=plt.cm.viridis_r, norm=plt.Normalize(vmin=min(accuracies), vmax=max(accuracies)))
      sm._A = []  # Fake up the array of the scalar mappable
      cbar = plt.colorbar(sm, orientation='vertical')
      cbar.set_label('Accuracy Range')

      # Rotate x-axis labels for better readability
      plt.xticks(rotation=45, ha='right')

      # Display the plot
      plt.tight_layout()
      plt.show()
```

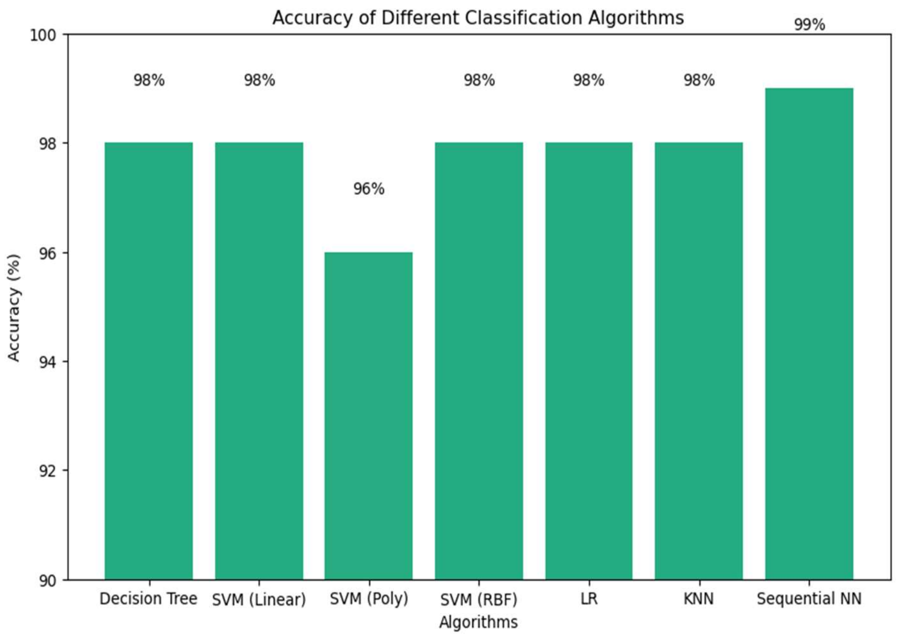


Figure 17: Comparison of the accuracy of different classification algorithms

This paper represented malware detection methods using machine learning along with basic concepts such as malware detection programs, and machine learning. Also, there were several classification algorithms and the accuracy of methods varied depending on the method used, the number of attributes, data sets, preprocessing methods, as well as tools implemented in the model. It also depends on the analysis method and the function used. As seen in Table 4, the best possible malware detection accuracy is a Sequential Neural Network. This algorithm achieves about 100% [23, 24].

Table 4
Classification algorithms and their accuracy.

	Classification Algorithms	Accuracy of Classification Algorithms
1	Decision Tree	0.98
2	SVM (Linear)	0.98
3	SVM (Poly)	0.96
4	SVM (RBF) Algorithms	0.98
5	LR	0.98
6	KNN	0.98
7	Sequential NN	0.99

5. Conclusions

Sequential Neural Network refers to one of the neural network's architecture types, where data is processed sequentially, step by step. This is useful for tasks, including time series data, such as text or speech. Based on work, this algorithm reaches the best results. However, it should be recognized that this conclusion does not allow us to claim that the Sequential NN is the best algorithm for detecting malware. There are always other measures to take care of, such as the dataset, preprocessing methods, extracted features, and feature selection methods used.

Further research directions include several key aspects. First, it is worth exploring the possibilities of developing hybrid models that combine different machine learning algorithms to improve overall efficiency and robustness to changing data conditions. Second, it is worth focusing on improving the interpretability of complex models such as neural networks to make them more suitable for forensic practice. Third, it is necessary to explore adaptive learning methods that will allow models to quickly respond to the emergence of new types of malware. It is also worth conducting a comparative analysis of algorithms in different digital forensics scenarios to identify specific requirements and optimal solutions for each task. Finally, it is important to investigate the impact of different tools and frameworks for implementing machine learning on the analysis results, which can significantly improve the practical application of these technologies in forensic science.

References

[1] R. Samani, McAfee Labs Report Reveals Latest COVID-19 Threats and Malware Surges. McAfee Blog (2024). URL: https://www.mcafee.com/blogs/other-blogs/mcafee-labs/mcafee-labs-report-reveals-latest-covid-19-threats-and-malware-surges/

[2] AV-TEST | Antivirus & Security Software & Antimalware Reviews. URL: https://av-test.org/en/statistics/malware.

[3] O. Mykhaylova, et al., Person-of-Interest Detection on Mobile Forensics Data—AI-Driven Roadmap, in: Workshop on Cybersecurity

Providing in Information and Telecommunication Systems, CPITS, vol. 3654 (2024) 239–251.

[4] V. Buhas, et al., Using Machine Learning Techniques to Increase the Effectiveness of Cybersecurity, in: Cybersecurity Providing in Information and Telecommunication Systems, vol. 3188, no. 2 (2021) 273–281.

[5] V. Zhebka, et al., Methodology for Predicting Failures in a Smart Home based on Machine Learning Methods, in: Workshop on Cybersecurity Providing in Information and Telecommunication Systems, CPITS, vol. 3654 (2024) 322–332.

[6] J. Scott, Signature based Malware Detection is Dead, Institute for Critical Infrastructure Technology (2017). https://informationsecurity.report/Resources/Whitepapers/920fbb41-8dc9-4053-bd01-72f961db24d9_ICIT-Analysis-Signature-Based-Malware-Detection-is-Dead.pdf

[7] A. Shabtai, et al., Anomaly: A Behavioral Malware Detection Framework for Android Devices, Journal of Intelligent Information Systems, vol. 38, no. 1 (2012) 161–190.

[8] I. Firdausi, et al., Analysis of Machine Learning Techniques used in Behavior-based Malware Detection, in: Proceedings of the 2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies. Washington, DC, USA: IEEE Computer Society (2010) 201–203.

[9] Z. Bazrafshan, et al., A Survey on Heuristic Malware Detection Techniques, in: 5th Conference on Information and Knowledge Technology (2013) 113–120.

[10] M. Zakeri, F. Daneshgar, M. Abbaspour, A Static Heuristic Approach to Detecting Malware Targets, Security and Communication Networks, vol. 8, no. 17 (2015) 3015–3027.

[11] M. Ahmadi, et al., Malware Detection by Behavioural Sequential Patterns, Comput. Fraud Secur., vol. 8 (2013) 11–19. doi: 10.1016/S1361-3723(13)70072-1.

[12] M. Akour, I. Alsmadi, M. Alazab, The Malware Detection Challenge of Accuracy, in: 2nd Int. Conf. Open Source Softw. Comput. (2017) 1–6, doi: 10.1109/osscom.2016.7863676.

[13] M. Alazab, et al., A Hybrid Wrapper-Filter Approach for Malware Detection, Journal of Networks, vol. 9, no. 11 (2014) 2878–2891.

[14] M. Vasilescu, L. Gheorghe, N. Tapus, Practical Malware Analysis based on Sandboxing, Proc. RoEduNet IEEE International Conference (2014). doi: 10.1109/RoEduNet-RENAM.2014.6955304.

[15] M. Alazab, et al., A Hybrid Wrapper-Filter Approach for Malware Detection, J. Networks, vol. 9, no. 11 (1969) 2878–2891. doi: 10.4304/jnw.9.11.2878-2891.

[16] S. Kumar, et al., Malicious Data Classification using Structural Information and Behavioral Specifications in Executables, in: Recent Adv. Eng. Comput. Sci. RAECS (2014) pp. 6–8, doi: 10.1109/RAECS.2014.6799525.

[17] M.A.M. Ali, M.A. Maarof, Dynamic Innate Immune System Model for Malware Detection, in: Int. Conference IT Converg. Security ICITCS (2013) 3–6, doi: 10.1109/ICITCS.2013.6717828.

[18] I. You, K. Yim, Malware Obfuscation Techniques: A Brief Survey, in: Proceedings of International Conference Broadband, Wirel, Comput. Commun. Appl. (2010), 297–300, doi: 10.1109/BWCCA.2010.85.

[19] J. Singh, J. Singh, A Survey on Machine Learning-based Malware Detection in Executable Files, J. Syst. Archit., vol. 112 (2021) 101861. doi: 10.1016/j.sysarc.2020.101861.

[20] W. Kanyongo, A. E. Ezugwu, Feature Selection and Importance of Predictors of Non-Communicable Diseases Medication Adherence from Machine Learning Research Perspectives, Informatics in Medicine Unlocked, vol. 38 (2023) art. 101232.

[21] G. Iashvili, et al., Content-based Machine Learning Approach for Hardware Vulnerabilities Identification System, Lecture Notes on Data Engineering and Communications Technologies, vol. 83 (2021) 117–126.

[22] Z. Guo, et al., A Novel Deep Learning Model Integrating CNN and GRU to Predict Particulate Matter Concentrations, Process Safety and Environmental Protection, vol. 173 (2023) 604–613. doi: 10.1016/j.psep.2023.03.052.

[23] A. Imanbayev, et al., Research of Machine Learning Algorithms for the Development of Intrusion Detection Systems in 5G Mobile Networks and Beyond, Sensors, vol. 22, iss. 24 (2022) 9957.

[24] N. Baisholan, et al., Implementation of Machine Learning Techniques to Detect Fraudulent Credit Card Transactions on a Designed Dataset, Journal of Theoretical and Applied Information Technology, vol. 101(13) (2023) 5279–5287.