Berliner Hochschule für Technik

Project Title: Detecting and Investigating Network Intrusions with NSL-KDD Dataset.

Master Colloquium B

Group-7

Supervisor: Prof. Dr. Ing. Marcus Purat

Submitted by: Md Ashraful Hakim Khan Siddiquee – 947152

Abstract

This study investigates the development and evaluation of a machine learning-based Intrusion Detection System (IDS) using the NSL-KDD dataset, a widely recognized benchmark in cybersecurity research. The primary goal is to enhance network intrusion detection by applying machine learning algorithms, specifically Logistic Regression, Random Forest, and Support Vector Machines (SVM). Various data preprocessing techniques, including feature selection, normalization, and handling class imbalance through class weighting, were applied to prepare the dataset. The models' performances were evaluated using key metrics such as accuracy, precision, recall, and F1-score. The Random Forest model achieved an accuracy of 81.79%, excelling particularly in detecting Denial of Service (DoS) and Probe attacks, while Logistic Regression achieved an accuracy of 82.91%. However, all models faced challenges in identifying Remote to Local (R2L) and User to Root (U2R) attacks, highlighting the difficulty of detecting rare attack types in imbalanced datasets. The results demonstrate the potential of machine learning to improve IDS effectiveness, though further research is necessary to enhance detection of more sophisticated attack vectors like R2L and U2R. This study contributes valuable insights into optimizing IDS performance through advanced data preprocessing and model selection techniques.

Table Of Contents

Chapter 1: Introduction

1.1 Context and Background

In today's digital age, network security has become a top priority for organizations worldwide. As businesses and institutions increasingly depend on digital platforms and the internet to manage their operations, the risk of cyber-attacks has grown significantly. These attacks can lead to severe financial losses, data breaches, and damage to an organization's reputation. As a result, implementing robust security measures is crucial to safeguard valuable information and maintain the integrity of network systems.

Intrusion Detection Systems (IDS) are key tools in cybersecurity, monitoring network traffic for suspicious behaviour and alerting administrators to potential threats. IDS are generally classified into two types: **Network-based Intrusion Detection Systems (NIDS)**, which monitor network traffic, and **Host-based Intrusion Detection Systems (HIDS)**, which monitor activity on individual devices or hosts. This study focuses on NIDS because they specialize in analysing network traffic and identifying unusual patterns that may signal malicious activity.

1.2 Motivation

The increasing sophistication and frequency of cyber-attacks demand more advanced security measures. Traditional IDS face several challenges when dealing with large-scale networks, managing vast amounts of data, and detecting new or unknown attack methods. These systems often struggle with scalability, and their ability to recognize novel threats while minimizing false alarms is limited.

To address these challenges, this study leverages the **NSL-KDD dataset**, a widely used benchmark in cybersecurity research, to assess the effectiveness of machine learning algorithms in intrusion detection. By incorporating machine learning, this study aims to develop a more effective IDS that can detect both known and unknown intrusions while reducing false positives. The ultimate goal is to build a system that enhances network security by providing timely and accurate alerts.

1.3 Significance of the Study

This research contributes to the growing field of Intrusion Detection Systems by exploring how machine learning can improve detection capabilities. By applying various machine learning algorithms to the NSL-KDD dataset, this study thoroughly evaluates their strengths and limitations. The objective is to develop an IDS that is better equipped to handle the evolving nature of cyber threats, thus addressing the shortcomings of traditional systems.

The findings from this research have the potential to benefit organizations looking to strengthen their cybersecurity defences, as well as researchers focused on advancing network security technologies. By improving the detection capabilities of IDS, this study aims to lower the success rate of cyber-attacks and help create a more secure and resilient digital environment.

Chapter 2: Literature Review

2.1 Introduction to Intrusion Detection Systems (IDS)

Intrusion Detection Systems (IDS) are essential tools for maintaining the security of computer networks by identifying unauthorized or malicious activities. An IDS can monitor network traffic or system logs for signs of intrusions, such as hacking attempts, malware infections, or other cyberattacks. IDS are generally classified into two types:

- **Signature-based IDS**: Detects attacks by comparing incoming data to known attack patterns or signatures.
- **Anomaly-based IDS**: Detects deviations from established normal behaviour patterns, flagging unusual activity as potential threats.

While signature-based systems are effective at identifying known threats, they often fail to detect novel or evolving attacks that do not match predefined signatures. Anomaly-based systems can detect unknown attacks but may suffer from high false-positive rates, flagging benign activity as malicious. In response to these limitations, machine learning techniques have emerged as a promising approach to enhance the accuracy and adaptability of IDS.

2.2 Evolution of Intrusion Detection Systems

The concept of IDS was first introduced in the early 1980s, focusing on analysing audit logs to detect unauthorized access. Since then, advancements in networking technologies and the rapid increase in cyber threats have driven the evolution of IDS into more sophisticated systems. Early IDS relied heavily on rule-based methods, but these systems proved inadequate in identifying complex attacks, especially as attackers developed more advanced techniques.

In recent years, researchers have increasingly focused on incorporating machine learning into IDS to overcome the limitations of rule-based approaches. Machine learning models can automatically learn patterns from data and generalize their understanding to detect both known and previously unseen attacks. This shift has opened new opportunities for developing more intelligent and adaptable IDS.

2.3 Machine Learning in IDS

The application of machine learning to IDS involves using algorithms to analyze network traffic or system logs and classify them as either normal or malicious. Various machine learning algorithms have been explored for intrusion detection, each with its strengths and limitations:

- **Supervised Learning**: Supervised learning techniques, such as Logistic Regression, Decision Trees, and Support Vector Machines (SVM), require labelled training data to learn patterns of malicious activity. These methods can achieve high accuracy when the training dataset is balanced and representative of real-world network traffic.
- **Unsupervised Learning**: Unsupervised learning algorithms, such as Clustering and Anomaly Detection models, do not require labelled data. They identify outliers or anomalous patterns that may indicate attacks, making them useful for detecting unknown threats. However, these models can be less accurate and more prone to false positives.
- **Semi-Supervised Learning**: Semi-supervised methods combine both labelled and unlabelled data to improve detection accuracy, especially when labelled data is scarce. These approaches have shown promise in detecting previously unknown attack types.
- **Deep Learning**: Deep learning, particularly Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), has gained popularity in IDS research for its ability to automatically extract features from raw data. These models have demonstrated superior performance in detecting complex attacks, though they require significant computational resources and large datasets for training.

2.4 Key Challenges in Machine Learning-based IDS

Although machine learning offers numerous advantages for IDS, several challenges must be addressed to develop effective systems:

2.4.1 Class Imbalance

One of the most significant challenges in developing machine learning-based IDS is the **class imbalance** present in most intrusion detection datasets. Attack classes such as **Denial of Service (DoS)** are often overrepresented, while rarer but critical attack types like **Remote to Local (R2L)** and **User to Root (U2R)** are underrepresented. This imbalance can lead to models that are biased toward the more frequent classes, making them less effective at detecting rare but potentially more dangerous attacks.

Researchers have addressed class imbalance using various techniques, such as **oversampling** (e.g., Synthetic Minority Over-sampling Technique, or SMOTE) and **undersampling**, as well as adjusting class weights during model training.

2.4.2 Feature Selection

Selecting the right features from a dataset is crucial to the performance of an IDS. Redundant or irrelevant features can introduce noise, decreasing model accuracy. Traditional IDS systems often rely on manually selected features, but with machine learning, **automated feature selection** techniques, such as **Select K-Best** and **Recursive Feature Elimination (RFE)**, have been applied to identify the most relevant features for intrusion detection.

2.4.3 Real-time Detection

Another challenge is achieving **real-time detection** in practical settings. Many machine learning models require significant computational resources, making it difficult to deploy them in environments where immediate detection is critical. Developing models that balance accuracy with speed is an ongoing area of research in IDS.

2.5 Previous Work

Numerous studies have explored the use of machine learning for IDS. Some notable works include:

- **Kumar and Spafford (1995)** introduced one of the earliest anomaly-based IDS, which analysed system audit logs for deviations from normal behaviour patterns. This laid the foundation for subsequent research into IDS using statistical and machine learning methods.
- **Lee et al. (1999)** proposed the use of data mining techniques for intrusion detection, identifying patterns in network traffic that could be used to detect attacks. Their work was a precursor to the use of machine learning models for IDS.
- **Sung and Mukkamala (2003)** applied Support Vector Machines (SVM) to detect intrusions, demonstrating the potential of machine learning in network security.

However, their work highlighted the need for more sophisticated approaches to address class imbalance and improve detection accuracy.

- **Mishra et al. (2018)** explored the use of deep learning for IDS, specifically employing Convolutional Neural Networks (CNNs) to automatically learn features from raw network data. Their study showed promising results but also underscored the computational challenges associated with deep learning.

2.6 NSL-KDD Dataset in IDS Research

The **NSL-KDD dataset** has become a widely accepted benchmark for evaluating IDS performance. It is a refined version of the original KDD 99 dataset, designed to address issues of redundancy and class imbalance. The NSL-KDD dataset includes records of network traffic labelled as either normal or an attack and has been used extensively in the development of machine learning models for IDS.

While the dataset provides a good starting point for testing models, it is not without limitations. The dataset does not fully capture the diversity of modern network attacks, and as a static dataset, it does not reflect the constantly evolving nature of cyber threats in real-world networks. Nonetheless, it remains a valuable tool for benchmarking IDS systems.

2.7 Attack Classes and Network Protocols

The NSL-KDD dataset also provides valuable insights into the protocols and attack vectors exploited by various cyberattacks. Key examples include:

- **DoS Attacks:** Often exploit weaknesses in the TCP/IP protocol suite, with SYN flooding being a common example. In such an attack, the target system is overwhelmed with an excessive number of SYN requests.
- **Probe Attacks:** Typically utilize protocols such as ICMP and DNS to gather information about a network's structure and services, potentially identifying vulnerabilities.
- **R2L Attacks:** These attacks exploit application-layer protocols such as HTTP and FTP to gain unauthorized access to systems.
- **U2R Attacks:** Attackers target system vulnerabilities to gain higher privileges, often by exploiting software or configuration flaws within the system.

Chapter 3: Problem Statement and Objectives

3.1 Problem Statement

As our world becomes more connected through the internet, the risk of cyberattacks continues to grow. Intrusion Detection Systems (IDS) are essential tools for monitoring and protecting networks from unauthorized access or malicious activity. However, traditional IDS methods often struggle to adapt to new, evolving attack techniques, leaving networks vulnerable to sophisticated threats.

One widely used dataset for evaluating IDS performance is the **NSL-KDD dataset**, which includes both normal network traffic and different types of cyberattacks, such as **Denial of Service (DoS)**, **Probe**, **Remote to Local (R2L)**, and **User to Root (U2R)** attacks. The challenge lies in the fact that some of these attacks, particularly R2L and U2R, are rare, making it difficult for machine learning models to accurately detect them. This imbalance in the dataset can lead to models that are biased toward the more common types of traffic, such as normal or DoS attacks, while failing to recognize the rarer but critical threats.

The main problem addressed in this project is to develop an effective machine learning-based IDS that can reliably detect both common and rare forms of cyberattacks. By exploring and refining various machine learning algorithms, this project aims to improve the detection rates of network intrusions, particularly the underrepresented attack types.

3.2 Objectives

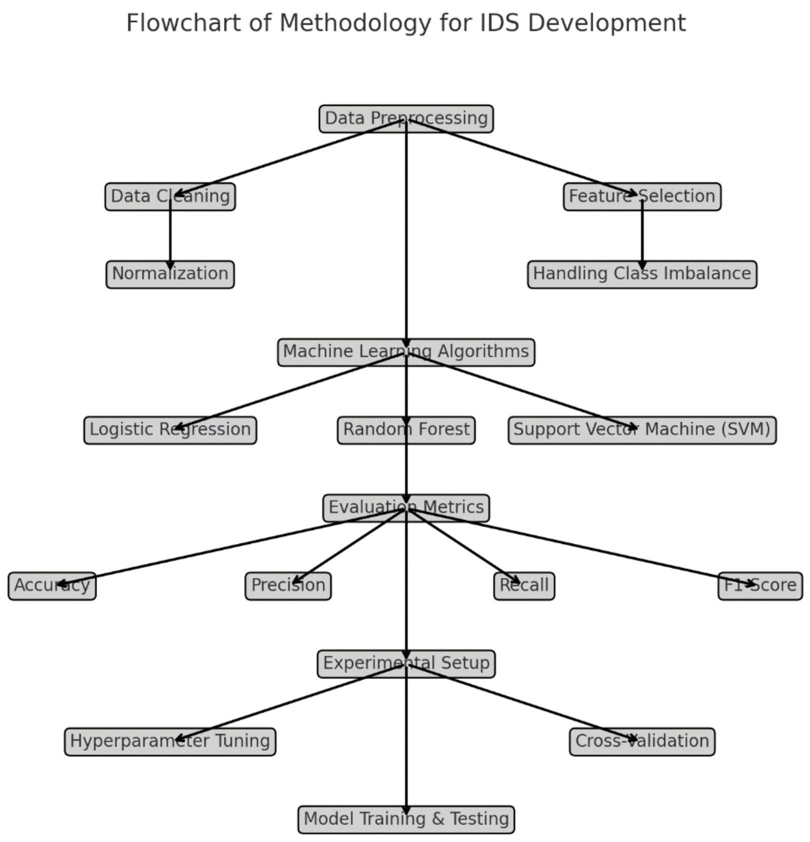
The core objective of this project is to design and evaluate a machine learning-based Network Intrusion Detection System (NIDS) using the NSL-KDD dataset. To achieve this goal, the following specific objectives are set:

1. **Preprocessing and Feature Selection**:
    - Clean and prepare the NSL-KDD dataset for model training by addressing missing values, standardizing numerical features, and encoding categorical variables where needed.
    - Use feature selection techniques to identify the most influential features that contribute to distinguishing between normal and attack traffic.

2. **Handling Class Imbalance**:
    - Implement strategies to deal with the class imbalance present in the dataset, focusing on ensuring that rare attack types like R2L and U2R are effectively identified, alongside more frequent attack types such as DoS and Probe.
3. **Building and Evaluating Machine Learning Models**:
    - Develop and test several machine learning models for intrusion detection, including:
        - **Logistic Regression**: A linear model that helps to establish a baseline.
        - **Random Forest**: An ensemble method that combines multiple decision trees for improved performance.
        - **Support Vector Machine (SVM)**: A robust model designed to find the optimal boundary between classes, particularly useful for non-linear relationships.
    - Fine-tune each model by adjusting its hyperparameters to ensure the best possible performance.
4. **Performance Assessment**:
    - Measure each model's performance using key classification metrics such as **accuracy**, **precision**, **recall**, and **F1-score**.
    - Perform cross-validation to confirm that the models generalize well and are not overfitting to the training data.
5. **Comparing Models**:
    - Compare the results of the models to determine which performs best in detecting both the frequent and rare types of network intrusions, based on the evaluation metrics.
6. **Providing Practical Insights**:
    - Through this study, offer insights into how machine learning can improve intrusion detection systems and how addressing class imbalance can enhance the detection of rare but significant threats like R2L and U2R attacks.

Chapter 4: Methodology

This chapter outlines the steps taken to develop the machine learning-based Intrusion Detection System (IDS) using the NSL-KDD dataset. The process was designed to build models capable of detecting network intrusions while addressing challenges like class imbalance and model optimization.



4.1 Dataset and Problem Definition

The NSL-KDD dataset was selected for this project due to its extensive use as a benchmark for evaluating network intrusion detection systems. The dataset contains records of network traffic labelled as either normal or an attack. The attack types are divided into four main categories:

- **Denial of Service (DoS)**: Attacks that aim to overwhelm network resources, making services unavailable to legitimate users.

- **Probe**: Attempts to gather information about the network to find potential vulnerabilities.
- **Remote to Local (R2L)**: Unauthorized access from a remote system to the local system.
- **User to Root (U2R)**: Attacks where the attacker gains root privileges on the system.

The target variable, attack_class, was created to map these categories to numerical values: 0 for normal traffic, 1 for DoS, 2 for Probe, 3 for R2L, and 4 for U2R.

4.2 Data Preprocessing

Data preprocessing is a crucial step to prepare the dataset for machine learning models. The following steps were performed:

4.2.1 Feature Selection

The dataset originally contains 41 features. Feature selection was used to reduce the dimensionality while retaining the most relevant features for network intrusion detection. The **Select K-Best** method, using an analysis of variance (ANOVA) statistical test (f_classif), was employed to select the top 15 features based on their correlation with the target variable. The selected features include serror_rate, same_srv_rate, logged_in, and count, which are critical for differentiating between normal and malicious network traffic.

4.2.2 Handling Class Imbalance

One of the key challenges in this project was dealing with the class imbalance in the NSL-KDD dataset. The dataset contains significantly more records of normal traffic and DoS attacks compared to rare attack types like R2L and U2R. To address this imbalance, the class_weight='balanced' option was applied in the models. This technique adjusts the weights of classes based on their frequency, ensuring that the models do not become biased toward the more frequent classes.

4.2.3 Data Cleaning and Transformation

Before model building, the dataset was checked for missing values and duplicates, both of which were handled accordingly. For features with a large range of values, normalization was applied to ensure all features were on a similar scale. This step is particularly important for models like **Support Vector Machine (SVM)**, which are sensitive to feature scaling.

4.2.4 Data Splitting

The dataset was split into training and test sets. The training set was used to train the machine learning models, while the test set was reserved for evaluating their performance on unseen data. This approach helps to measure how well the models generalize to new data and avoid overfitting.

4.3 Machine Learning Models

Three machine learning algorithms were selected for this study: **Logistic Regression**, **Random Forest**, and **Support Vector Machine (SVM)**. These models were chosen for their proven effectiveness in classification tasks, particularly in handling structured data like the NSL-KDD dataset.

4.3.1 Logistic Regression

Logistic Regression is a widely used linear model that predicts the probability of a class based on input features. It was implemented using the LogisticRegression class from the scikit-learn library. The model was trained with the newton-cg solver and the class_weight='balanced' option to handle class imbalance. Hyperparameter tuning was performed to optimize the regularization strength C and the number of iterations max_iter. Cross-validation was used to validate the model's performance across different data splits.

4.3.2 Random Forest

Random Forest is an ensemble learning method that builds multiple decision trees and combines their predictions for a more robust classification. The RandomForestClassifier from scikit-learn was used to implement this model. To ensure optimal performance, hyperparameter tuning was conducted for the number of trees (n_estimators), the maximum depth of each tree (max_depth), and the minimum number of samples required for splits and leaf nodes. These adjustments helped the model capture complex patterns in the data without overfitting.

4.3.3 Support Vector Machine (SVM)

SVM is a powerful algorithm that seeks the optimal boundary between classes, particularly useful for non-linear decision boundaries. The SVC class from scikit-learn was used with a Radial Basis Function (RBF) kernel to account for non-linearity in the dataset. Like the other models, the class_weight='balanced' option was used to handle class imbalance. The key hyperparameters, such as the regularization parameter C and the kernel coefficient gamma, were fine-tuned using **Grid Search Cross-Validation** to improve performance.

4.4 Hyperparameter Tuning

Hyperparameter tuning was performed using **Grid Search Cross-Validation** for all models to find the best combination of parameters. The GridSearchCV function from scikit-learn was used to test a range of hyperparameter values and select the combination that maximized the accuracy on the training set. This method helps ensure that each model was optimally configured to achieve the best results.

- **Logistic Regression**: Tuned hyperparameters included regularization strength C, solver (newton-cg, liblinear, etc.), and max_iter.
- **Random Forest**: Key parameters included the number of trees (n_estimators), the maximum depth of trees (max_depth), and the minimum samples for splits and leaves.
- **SVM**: The regularization parameter C and the kernel coefficient gamma were tuned to ensure the best fit.

4.5 Model Evaluation

Each model's performance was evaluated using the following metrics:

- **Accuracy**: The proportion of correctly classified samples.
- **Precision**: The ability of the model to correctly identify positive instances.
- **Recall**: The model's ability to capture all relevant positive instances.
- **F1-Score**: The harmonic mean of precision and recall, which is particularly useful for evaluating models on imbalanced datasets.

Cross-validation was employed to further evaluate the models and ensure that they generalize well to unseen data, thus preventing overfitting.

Chapter 5: Implementation

This chapter describes the detailed steps followed to implement the machine learning-based Intrusion Detection System (IDS) using the NSL-KDD dataset. The implementation involves several key stages: dataset preparation, model development, hyperparameter tuning, and model evaluation.

5.1 Tools and Libraries

The implementation was carried out using **Python** and key machine learning libraries. The following tools and libraries were utilized:

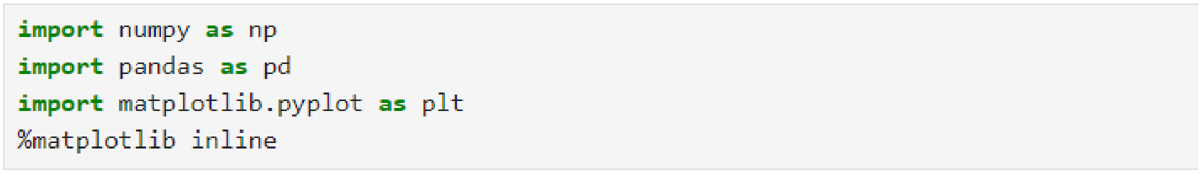
- **Python**: The programming language used for the entire project.
- **Pandas**: For data manipulation and preprocessing.
- **NumPy**: For numerical computations.
- **scikit-learn**: For building machine learning models, feature selection, hyperparameter tuning, and performance evaluation.
- **Matplotlib** and **Seaborn**: For data visualization and exploratory data analysis.
- **GridSearchCV**: For hyperparameter tuning to optimize model performance.

5.2 Data Preprocessing

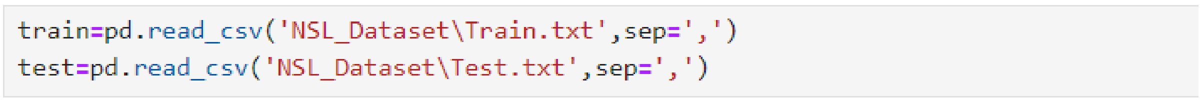
Before training the machine learning models, the NSL-KDD dataset was pre-processed. The following steps were undertaken to prepare the data:

5.2.1 Loading the Dataset

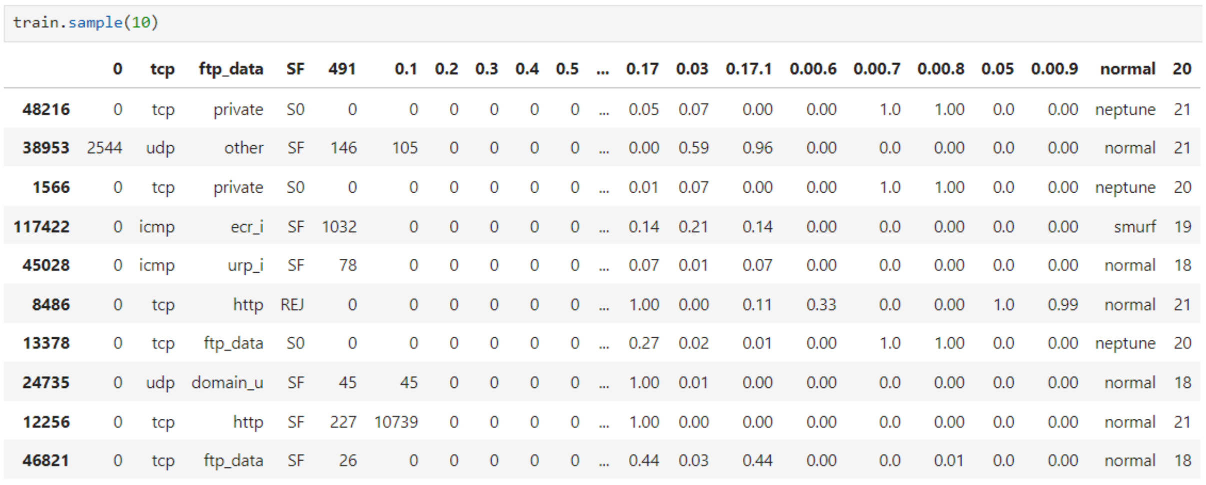
The NSL-KDD dataset was loaded into a pandas DataFrame from CSV files, separating the training and test datasets.



```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```



```python
train=pd.read_csv('NSL_Dataset\Train.txt',sep=',')
test=pd.read_csv('NSL_Dataset\Test.txt',sep=',')
```

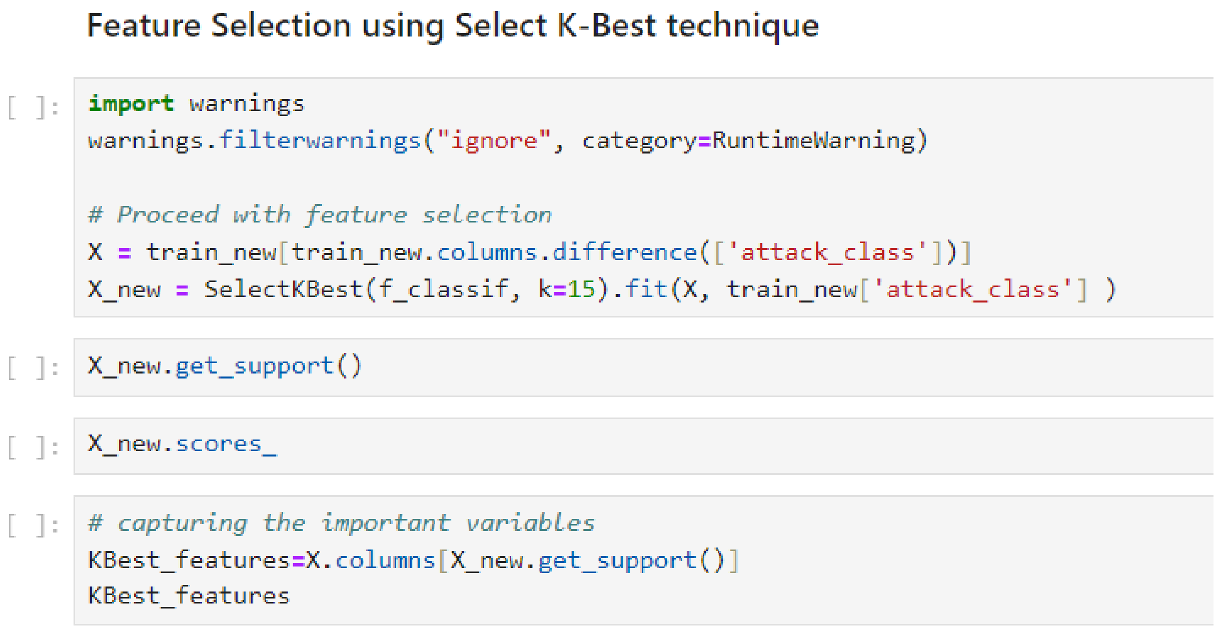


```
train.sample(10)
```

	0	tcp	ftp_data	SF	491	0.1	0.2	0.3	0.4	0.5	...	0.17	0.03	0.17.1	0.00.6	0.00.7	0.00.8	0.05	0.00.9	normal	20
48216	0	tcp	private	S0	0	0	0	0	0	0	...	0.05	0.07	0.00	0.00	1.0	1.00	0.0	0.00	neptune	21
38953	2544	udp	other	SF	146	105	0	0	0	0	...	0.00	0.59	0.96	0.00	0.0	0.00	0.0	0.00	normal	21
1566	0	tcp	private	S0	0	0	0	0	0	0	...	0.01	0.07	0.00	0.00	1.0	1.00	0.0	0.00	neptune	20
117422	0	icmp	ecr_i	SF	1032	0	0	0	0	0	...	0.14	0.21	0.14	0.00	0.0	0.00	0.0	0.00	smurf	19
45028	0	icmp	urp_i	SF	78	0	0	0	0	0	...	0.07	0.01	0.07	0.00	0.0	0.00	0.0	0.00	normal	18
8486	0	tcp	http	REJ	0	0	0	0	0	0	...	1.00	0.00	0.11	0.33	0.0	0.00	1.0	0.99	normal	21
13378	0	tcp	ftp_data	S0	0	0	0	0	0	0	...	0.27	0.02	0.01	0.00	1.0	1.00	0.0	0.00	neptune	20
24735	0	udp	domain_u	SF	45	45	0	0	0	0	...	1.00	0.01	0.00	0.00	0.0	0.00	0.0	0.00	normal	18
12256	0	tcp	http	SF	227	10739	0	0	0	0	...	1.00	0.00	0.00	0.00	0.0	0.00	0.0	0.00	normal	21
46821	0	tcp	ftp_data	SF	26	0	0	0	0	0	...	0.44	0.03	0.44	0.00	0.0	0.01	0.0	0.00	normal	18

5.2.2 Feature Selection

Feature selection was done using **Select K-Best** to reduce the dimensionality of the dataset while retaining the most important features.



Feature Selection using Select K-Best technique

```
[ ]: import warnings
     warnings.filterwarnings("ignore", category=RuntimeWarning)

     # Proceed with feature selection
     X = train_new[train_new.columns.difference(['attack_class'])]
     X_new = SelectKBest(f_classif, k=15).fit(X, train_new['attack_class'] )
```

```
[ ]: X_new.get_support()
```

```
[ ]: X_new.scores_
```

```
[ ]: # capturing the important variables
     KBest_features=X.columns[X_new.get_support()]
     KBest_features
```

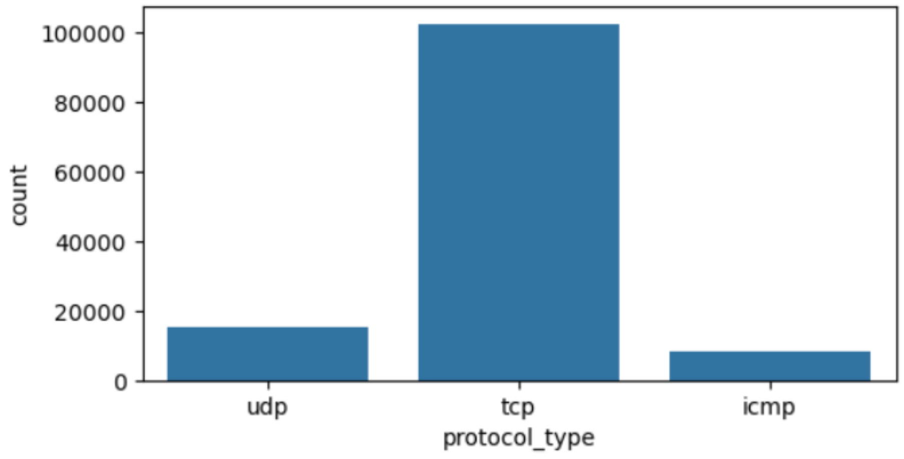
5.2.3 Basic Exploratory Data Analysis (EDA)

Basic Exploratory Analysis

```
2]: train['protocol_type'].value_counts()
```

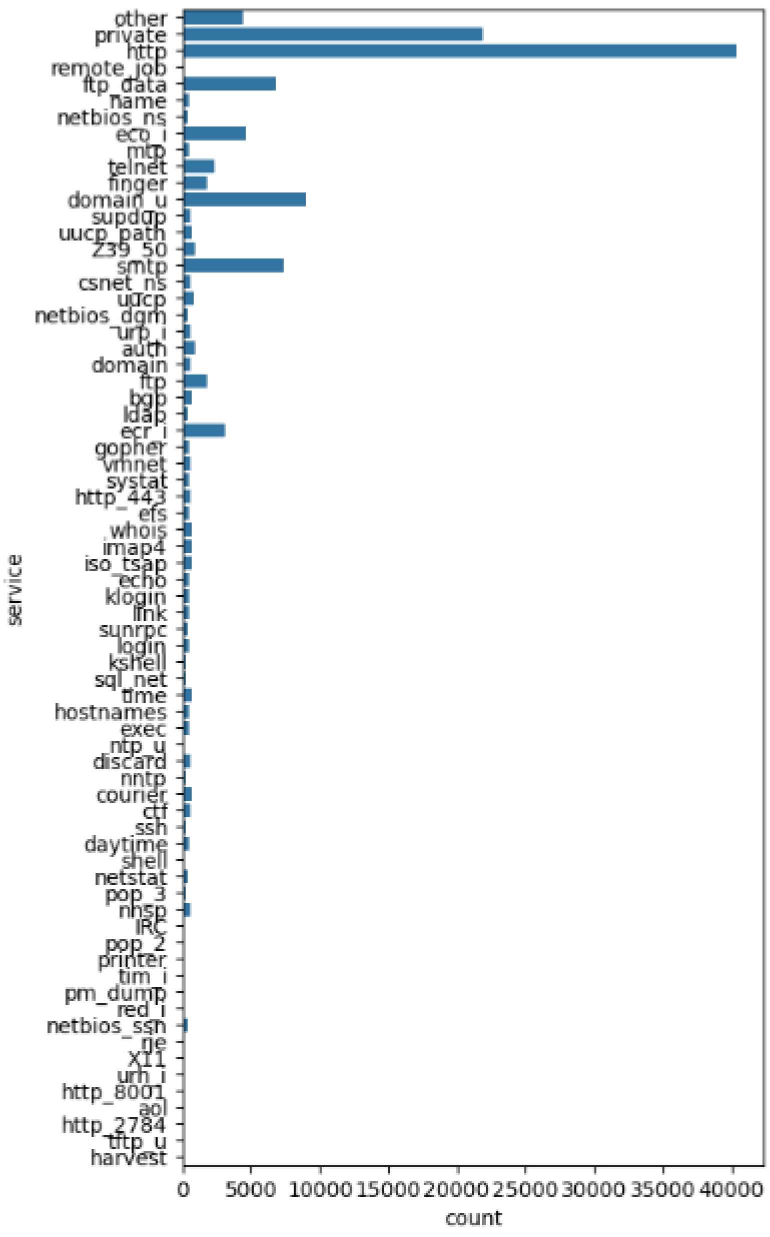
```
2]: protocol_type
    tcp      102688
    udp       14993
    icmp       8291
    Name: count, dtype: int64
```

```
3]: # Protocol type distribution
    plt.figure(figsize=(6,3))
    sns.countplot(x='protocol_type', data=train)
    plt.show()
```



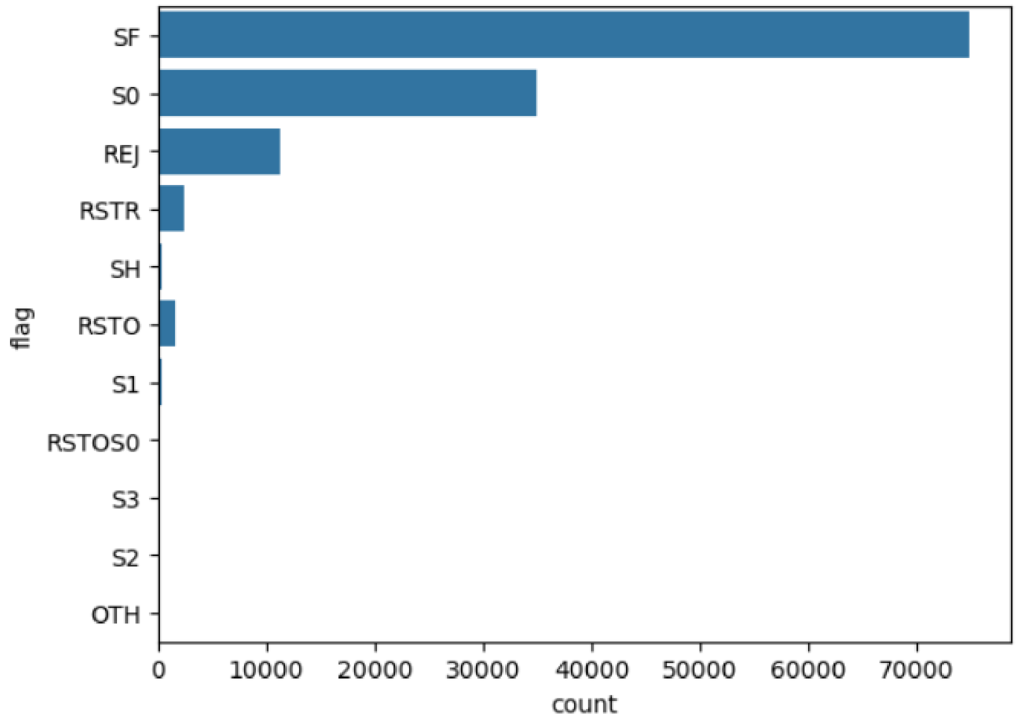
```
4]: # service distribution
    plt.figure(figsize=(5,10))
    sns.countplot(train['service'])
```

```
4]: <Axes: xlabel='count', ylabel='service'>
```



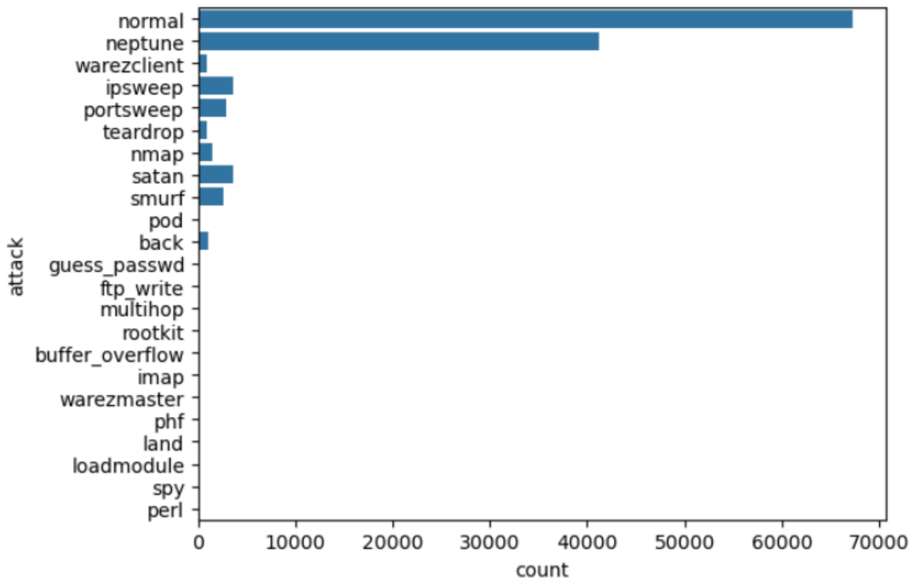
```
: # flag distribution
  sns.countplot(train['flag'])
```

```
: <Axes: xlabel='count', ylabel='flag'>
```

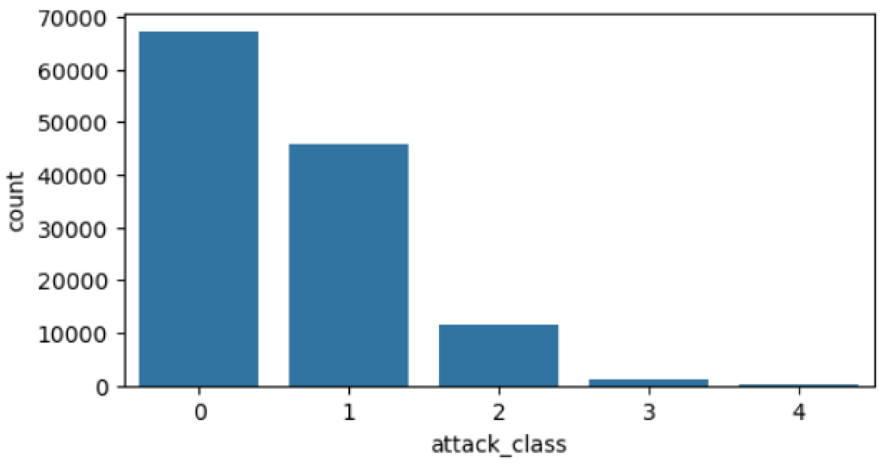


```
[106]: # attack distribution
       sns.countplot(train['attack'])
```

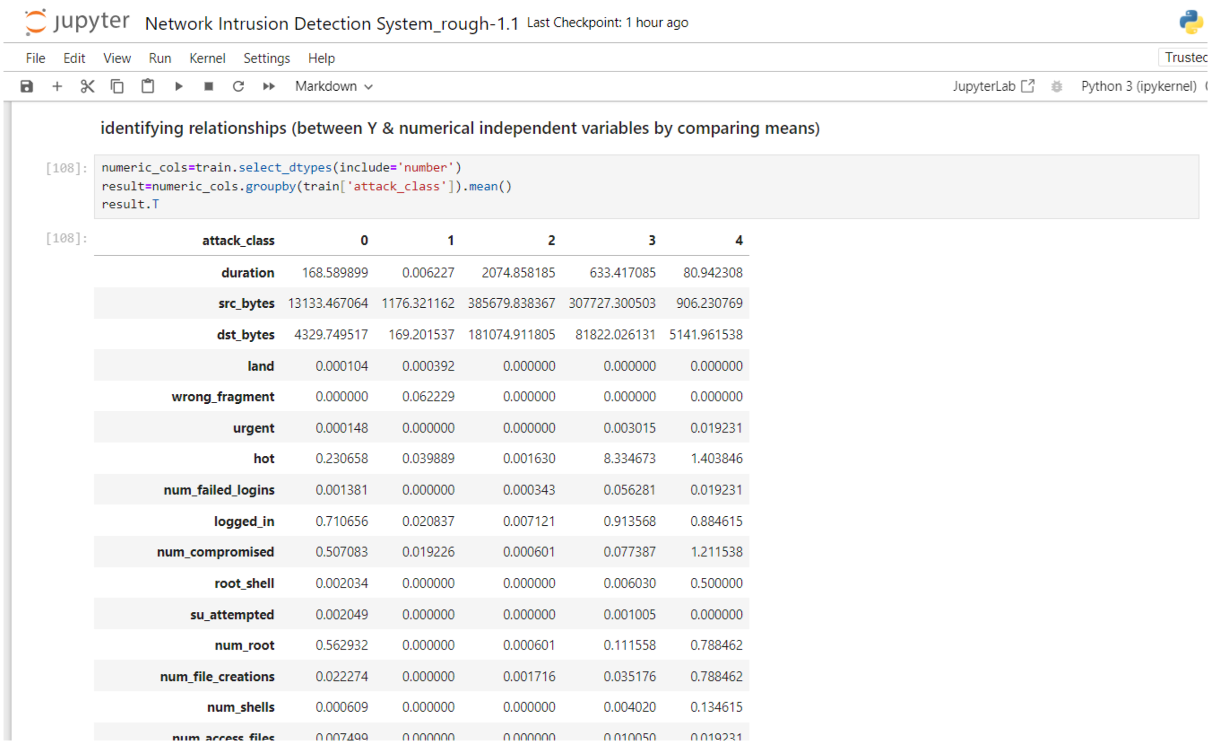
```
[106]: <Axes: xlabel='count', ylabel='attack'>
```



```
[107]: # attack class distribution
       plt.figure(figsize=(6,3))
       sns.countplot(x='attack_class', data=train)

[107]: <Axes: xlabel='attack_class', ylabel='count'>
```



5.2.4 Identifying Relationships (Between Y & Numerical Independent Variables)



Jupyter Network Intrusion Detection System_rough-1.1 Last Checkpoint: 1 hour ago
File Edit View Run Kernel Settings Help
Trusted
Markdown
JupyterLab
Python 3 (ipykernel)

identifying relationships (between Y & numerical independent variables by comparing means)

```
[108]: numeric_cols=train.select_dtypes(include='number')
       result=numeric_cols.groupby(train['attack_class']).mean()
       result.T
```

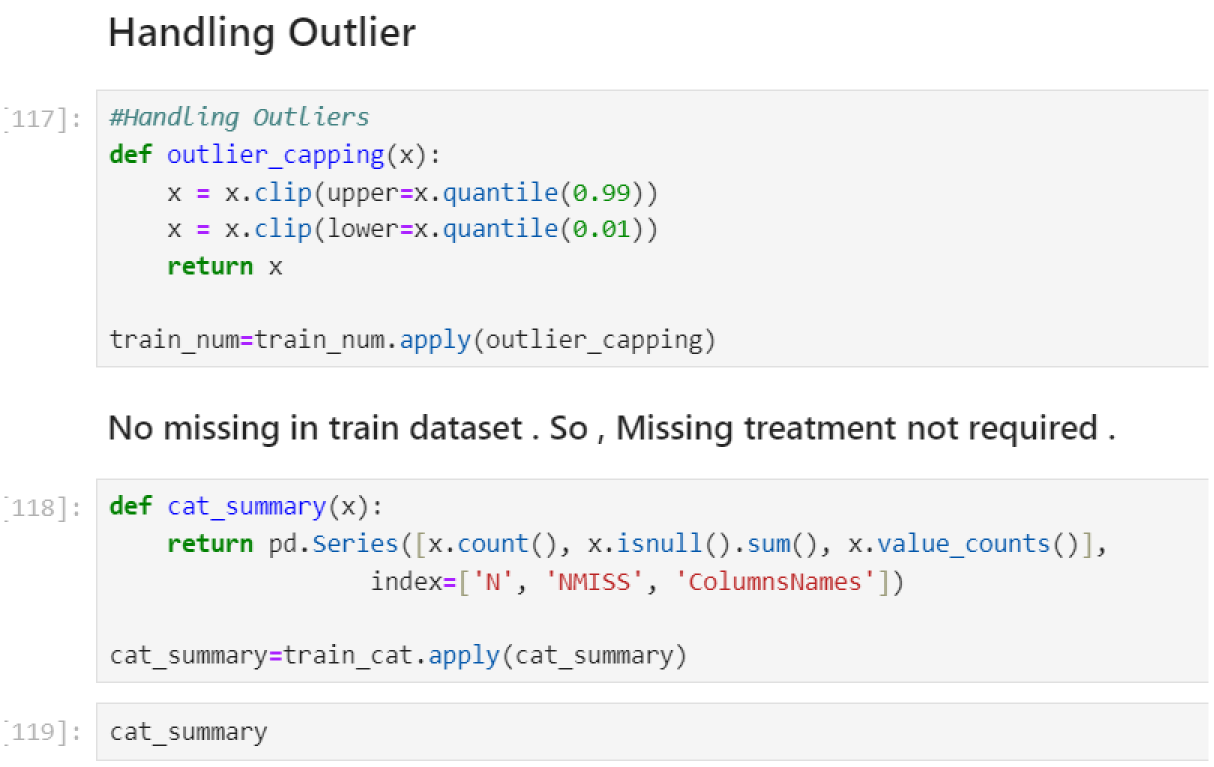
[108]:

attack_class	0	1	2	3	4
duration	168.589899	0.006227	2074.858185	633.417085	80.942308
src_bytes	13133.467064	1176.321162	385679.838367	307727.300503	906.230769
dst_bytes	4329.749517	169.201537	181074.911805	81822.026131	5141.961538
land	0.000104	0.000392	0.000000	0.000000	0.000000
wrong_fragment	0.000000	0.062229	0.000000	0.000000	0.000000
urgent	0.000148	0.000000	0.000000	0.003015	0.019231
hot	0.230658	0.039889	0.001630	8.334673	1.403846
num_failed_logins	0.001381	0.000000	0.000343	0.056281	0.019231
logged_in	0.710656	0.020837	0.007121	0.913568	0.884615
num_compromised	0.507083	0.019226	0.000601	0.077387	1.211538
root_shell	0.002034	0.000000	0.000000	0.006030	0.500000
su_attempted	0.002049	0.000000	0.000000	0.001005	0.000000
num_root	0.562932	0.000000	0.000601	0.111558	0.788462
num_file_creations	0.022274	0.000000	0.001716	0.035176	0.788462
num_shells	0.000609	0.000000	0.000000	0.004020	0.134615
num_access_files	0.007499	0.000000	0.000000	0.010050	0.019231

Observations:

- The length of time duration of connection for attack is higher than normal.
- Wrong fragments in the connection are only present in attack.
- Number of outbound commands in an ftp session are 0 in both normal and attack.

5.2.5 Handling Outliers



Handling Outlier

117]:

```
#Handling Outliers
def outlier_capping(x):
    x = x.clip(upper=x.quantile(0.99))
    x = x.clip(lower=x.quantile(0.01))
    return x

train_num=train_num.apply(outlier_capping)
```

No missing in train dataset . So , Missing treatment not required .

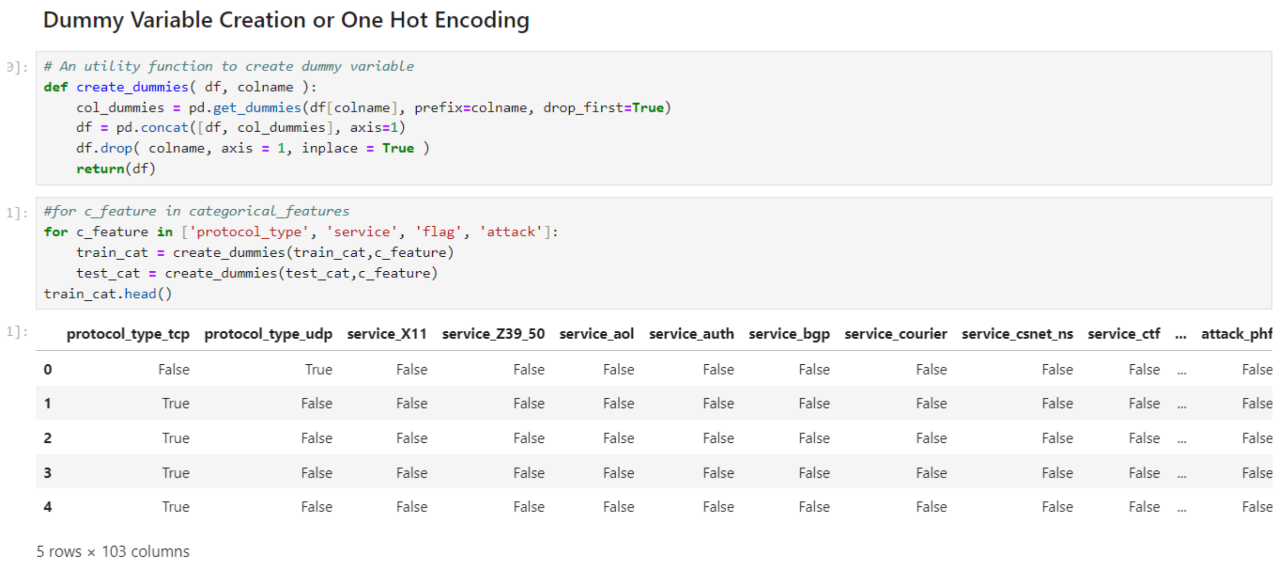
118]:

```
def cat_summary(x):
    return pd.Series([x.count(), x.isnull().sum(), x.value_counts()],
                     index=['N', 'NMISS', 'ColumnsNames'])

cat_summary=train_cat.apply(cat_summary)
```

119]:

```
cat_summary
```

5.2.6 Dummy Variable Creation / One Hot Encoding



Dummy Variable Creation or One Hot Encoding

0]:

```
# An utility function to create dummy variable
def create_dummies( df, colname ):
    col_dummies = pd.get_dummies(df[colname], prefix=colname, drop_first=True)
    df = pd.concat([df, col_dummies], axis=1)
    df.drop( colname, axis = 1, inplace = True )
    return(df)
```

1]:

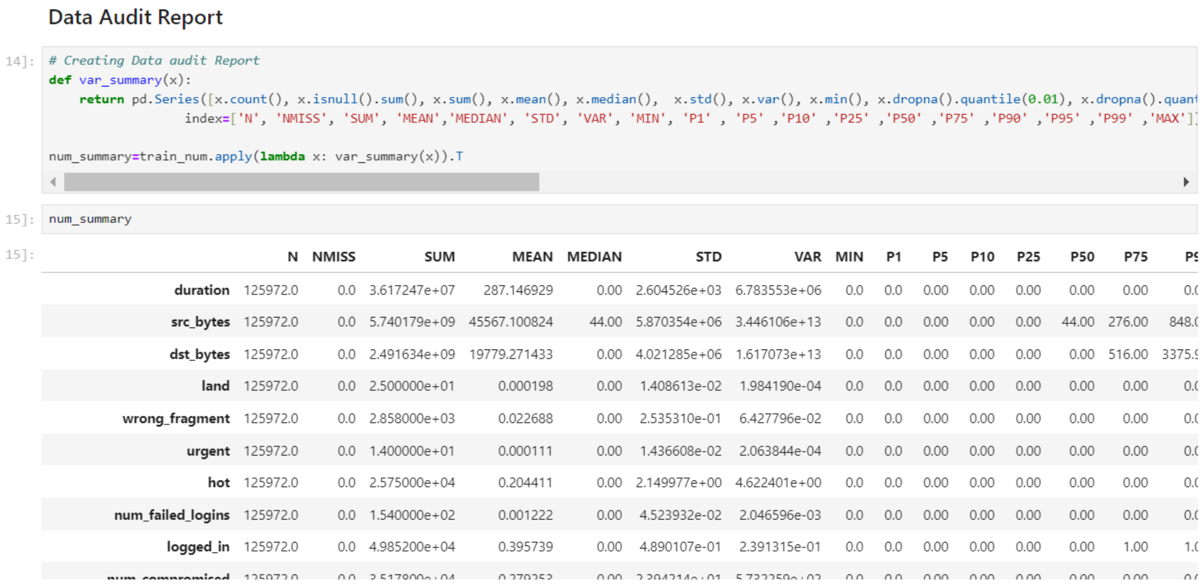
```
#for c_feature in categorical_features
for c_feature in ['protocol_type', 'service', 'flag', 'attack']:
    train_cat = create_dummies(train_cat,c_feature)
    test_cat = create_dummies(test_cat,c_feature)
train_cat.head()
```

1]:

	protocol_type_tcp	protocol_type_udp	service_X11	service_Z39_50	service_aol	service_auth	service_bgp	service_courier	service_csnet_ns	service_ctf	...	attack_phf
0	False	True	False	False	False	False	False	False	False	False	...	False
1	True	False	False	False	False	False	False	False	False	False	...	False
2	True	False	False	False	False	False	False	False	False	False	...	False
3	True	False	False	False	False	False	False	False	False	False	...	False
4	True	False	False	False	False	False	False	False	False	False	...	False

5 rows × 103 columns

5.2.7 Data Audit Report



Data Audit Report

```
14]: # Creating Data audit Report
     def var_summary(x):
         return pd.Series([x.count(), x.isnull().sum(), x.sum(), x.mean(), x.median(),  x.std(), x.var(), x.min(), x.dropna().quantile(0.01), x.dropna().quan
                       index=['N', 'NMISS', 'SUM', 'MEAN','MEDIAN', 'STD', 'VAR', 'MIN', 'P1' , 'P5' ,'P10' ,'P25' ,'P50' ,'P75' ,'P90' ,'P95' ,'P99' ,'MAX']

     num_summary=train_num.apply(lambda x: var_summary(x)).T
```

```
15]: num_summary
```

15]:

	N	NMISS	SUM	MEAN	MEDIAN	STD	VAR	MIN	P1	P5	P10	P25	P50	P75	P9
duration	125972.0	0.0	3.617247e+07	287.146929	0.00	2.604526e+03	6.783553e+06	0.0	0.0	0.00	0.00	0.00	0.00	0.00	0.0
src_bytes	125972.0	0.0	5.740179e+09	45567.100824	44.00	5.870354e+06	3.446106e+13	0.0	0.0	0.00	0.00	0.00	44.00	276.00	848.0
dst_bytes	125972.0	0.0	2.491634e+09	19779.271433	0.00	4.021285e+06	1.617073e+13	0.0	0.0	0.00	0.00	0.00	0.00	516.00	3375.9
land	125972.0	0.0	2.500000e+01	0.000198	0.00	1.408613e-02	1.984190e-04	0.0	0.0	0.00	0.00	0.00	0.00	0.00	0.0
wrong_fragment	125972.0	0.0	2.858000e+03	0.022688	0.00	2.535310e-01	6.427796e-02	0.0	0.0	0.00	0.00	0.00	0.00	0.00	0.0
urgent	125972.0	0.0	1.400000e+01	0.000111	0.00	1.436608e-02	2.063844e-04	0.0	0.0	0.00	0.00	0.00	0.00	0.00	0.0
hot	125972.0	0.0	2.575000e+04	0.204411	0.00	2.149977e+00	4.622401e+00	0.0	0.0	0.00	0.00	0.00	0.00	0.00	0.0
num_failed_logins	125972.0	0.0	1.540000e+02	0.001222	0.00	4.523932e-02	2.046596e-03	0.0	0.0	0.00	0.00	0.00	0.00	0.00	0.0
logged_in	125972.0	0.0	4.985200e+04	0.395739	0.00	4.890107e-01	2.391315e-01	0.0	0.0	0.00	0.00	0.00	0.00	1.00	1.0
num_compromised	125972.0	0.0	3.517800e+04	0.279253	0.00	2.394214e+01	5.732259e+02	0.0	0.0	0.00	0.00	0.00	0.00	0.00	0.0

5.2.8 Dropping Columns Based on Data Audit Report

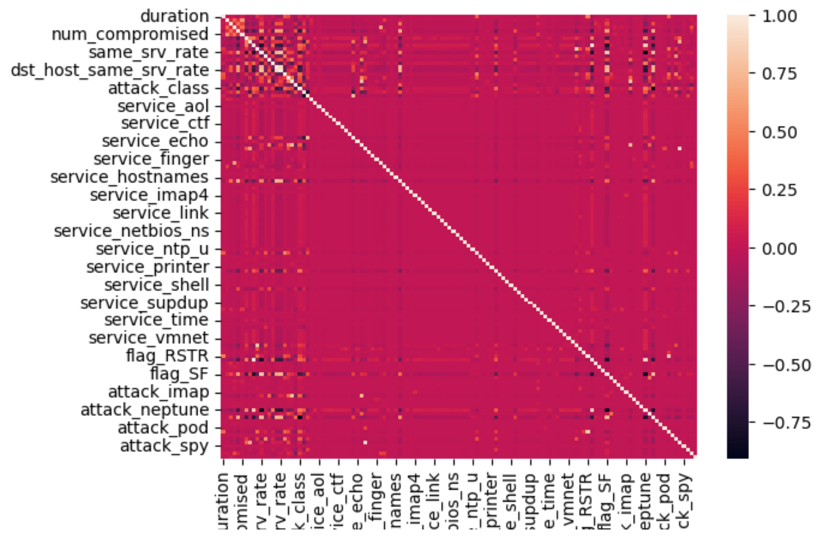
Dropping columns based on data audit report

```
- Based on low variance (near zero variance)
- High missings (>25% missings)
- High correlations between two numerical variables
```

```
26]: train_new.drop(columns=['land','wrong_fragment','urgent','num_failed_logins',"root_shell","su_attempted","num_root",
                              "num_file_creations","num_shells","num_access_files","num_outbound_cmds","is_host_login","is_guest_login",
                              'dst_host_rerror_rate','dst_host_serror_rate','dst_host_srv_rerror_rate','dst_host_srv_serror_rate',
                              'num_root','num_outbound_cmds','srv_rerror_rate','srv_serror_rate'], axis=1, inplace=True)
```

```
27]: sns.heatmap(train_new.corr())
```

```
27]: <Axes: >
```

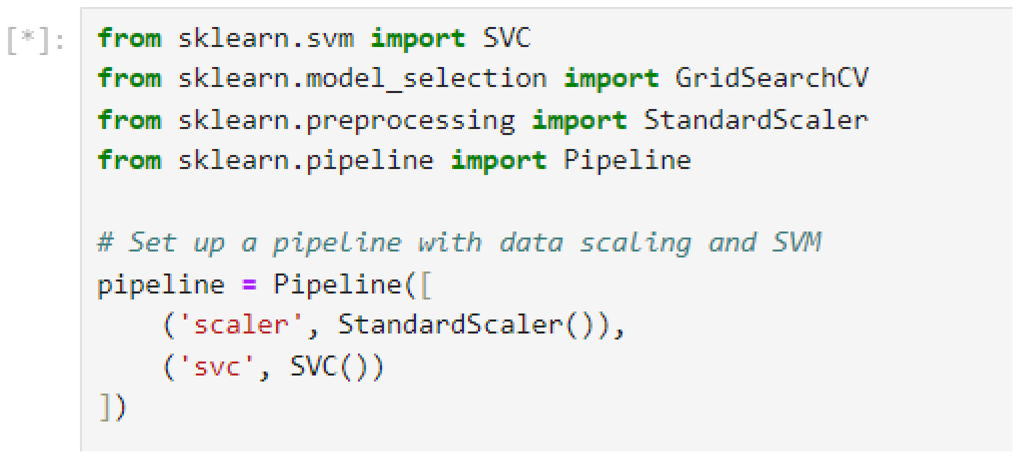


5.2.9 Handling Class Imbalance

To address the class imbalance issue in the dataset, the class_weight='balanced' option was used in all models. This ensures that the model assigns a higher weight to minority classes like **R2L** and **U2R**, making them less likely to be overlooked during the training process.

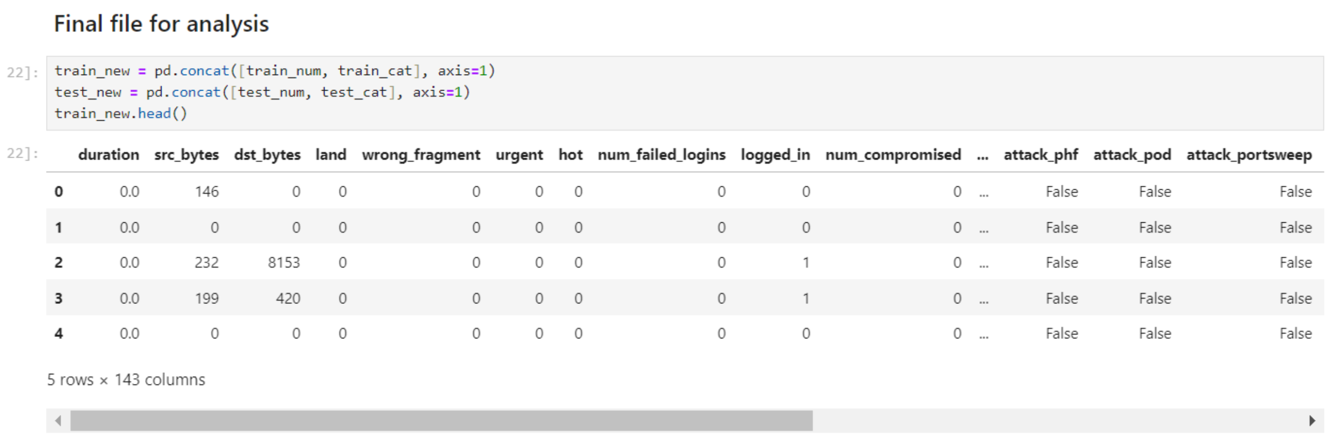
5.2.10 Data Scaling

Since **Support Vector Machine (SVM)** is sensitive to feature scaling, datasets were standardized using **StandardScaler**.



```
[*]: from sklearn.svm import SVC
     from sklearn.model_selection import GridSearchCV
     from sklearn.preprocessing import StandardScaler
     from sklearn.pipeline import Pipeline

     # Set up a pipeline with data scaling and SVM
     pipeline = Pipeline([
         ('scaler', StandardScaler()),
         ('svc', SVC())
     ])
```

5.2.11 Final file for analysis



Final file for analysis

```
22]: train_new = pd.concat([train_num, train_cat], axis=1)
     test_new = pd.concat([test_num, test_cat], axis=1)
     train_new.head()
```

22]:

	duration	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	num_failed_logins	logged_in	num_compromised	...	attack_phf	attack_pod	attack_portsweep
0	0.0	146	0	0	0	0	0	0	0	0	...	False	False	False
1	0.0	0	0	0	0	0	0	0	0	0	...	False	False	False
2	0.0	232	8153	0	0	0	0	0	1	0	...	False	False	False
3	0.0	199	420	0	0	0	0	0	1	0	...	False	False	False
4	0.0	0	0	0	0	0	0	0	0	0	...	False	False	False

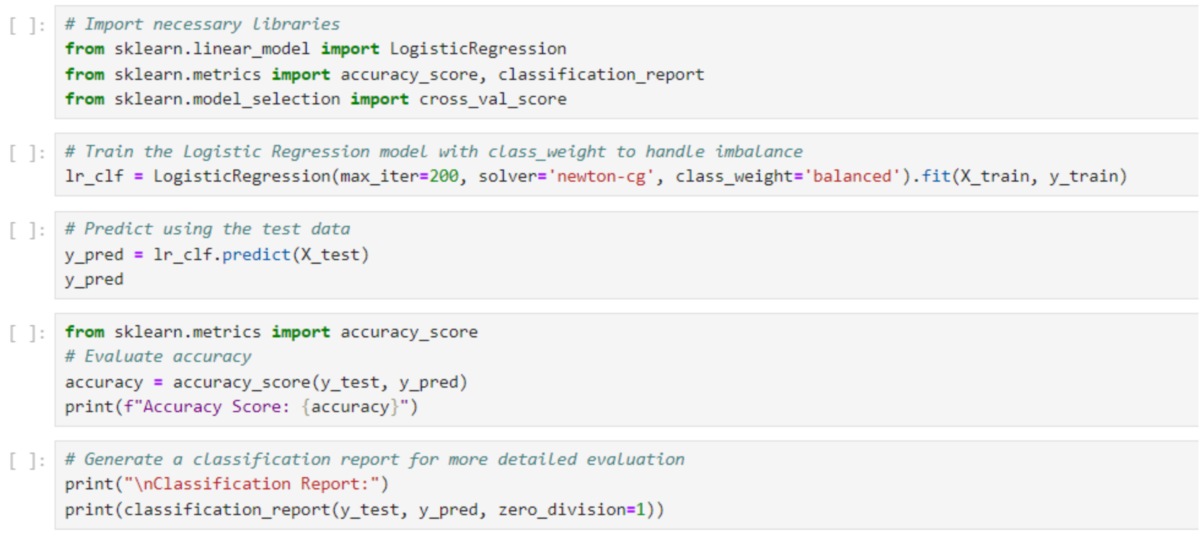
5 rows × 143 columns

5.3 Machine Learning Models

Three machine learning models were selected for this project: **Logistic Regression**, **Random Forest**, and **Support Vector Machine (SVM)**. Each model was built and trained on the pre-processed data.

5.3.1 Logistic Regression

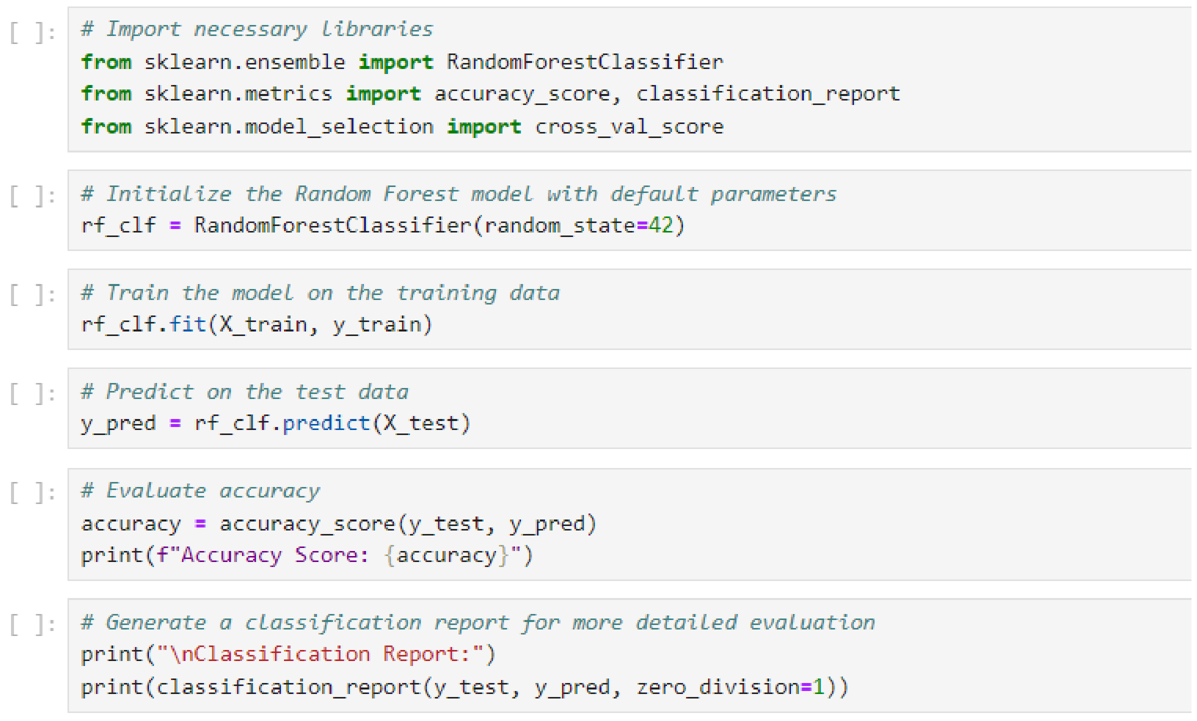
Logistic Regression was used as the baseline model for comparison. The model was initialized with the newton-cg solver, and the class_weight='balanced' parameter was applied to handle class imbalance.



```
[ ]: # Import necessary libraries
     from sklearn.linear_model import LogisticRegression
     from sklearn.metrics import accuracy_score, classification_report
     from sklearn.model_selection import cross_val_score

[ ]: # Train the Logistic Regression model with class_weight to handle imbalance
     lr_clf = LogisticRegression(max_iter=200, solver='newton-cg', class_weight='balanced').fit(X_train, y_train)

[ ]: # Predict using the test data
     y_pred = lr_clf.predict(X_test)
     y_pred

[ ]: from sklearn.metrics import accuracy_score
     # Evaluate accuracy
     accuracy = accuracy_score(y_test, y_pred)
     print(f"Accuracy Score: {accuracy}")

[ ]: # Generate a classification report for more detailed evaluation
     print("\nClassification Report:")
     print(classification_report(y_test, y_pred, zero_division=1))
```

5.3.2 Random Forest

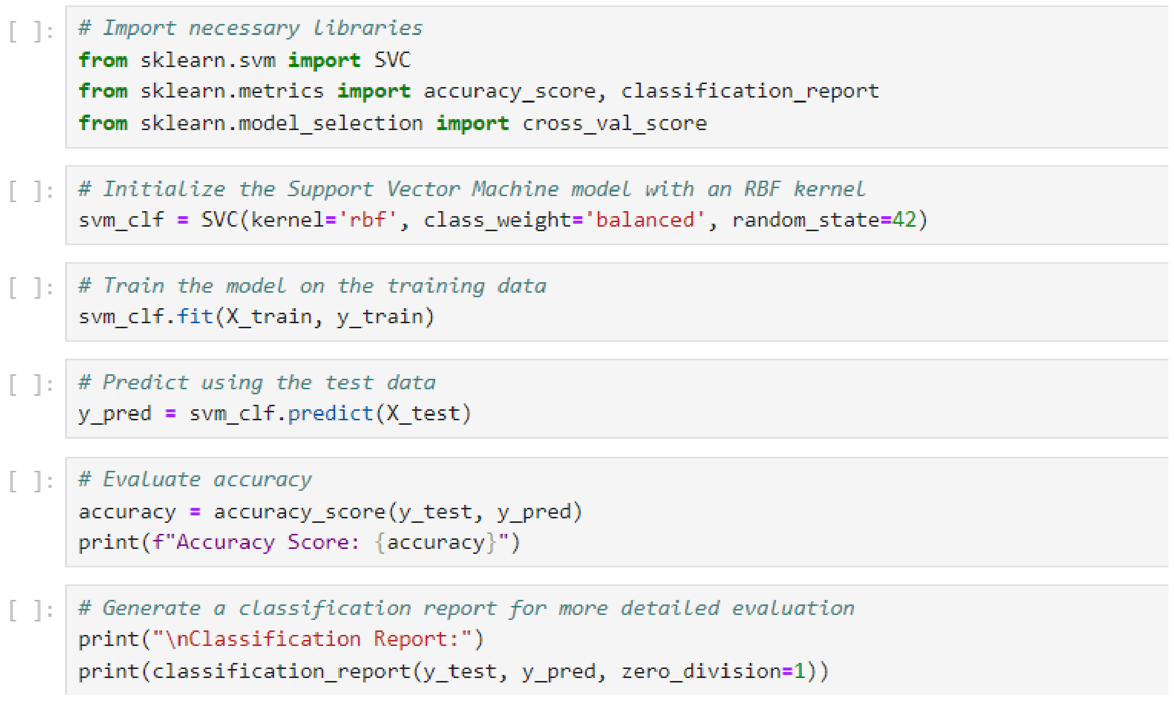
A **Random Forest** model was developed to leverage its ensemble nature, which combines multiple decision trees to improve prediction accuracy. The default model was first built, followed by hyperparameter tuning.



```
[ ]: # Import necessary libraries
     from sklearn.ensemble import RandomForestClassifier
     from sklearn.metrics import accuracy_score, classification_report
     from sklearn.model_selection import cross_val_score

[ ]: # Initialize the Random Forest model with default parameters
     rf_clf = RandomForestClassifier(random_state=42)

[ ]: # Train the model on the training data
     rf_clf.fit(X_train, y_train)

[ ]: # Predict on the test data
     y_pred = rf_clf.predict(X_test)

[ ]: # Evaluate accuracy
     accuracy = accuracy_score(y_test, y_pred)
     print(f"Accuracy Score: {accuracy}")

[ ]: # Generate a classification report for more detailed evaluation
     print("\nClassification Report:")
     print(classification_report(y_test, y_pred, zero_division=1))
```

5.3.3 Support Vector Machine (SVM)

SVM with an RBF kernel was implemented to capture non-linear patterns in the data. Like the other models, class_weight='balanced' was applied to handle the imbalanced nature of the dataset.

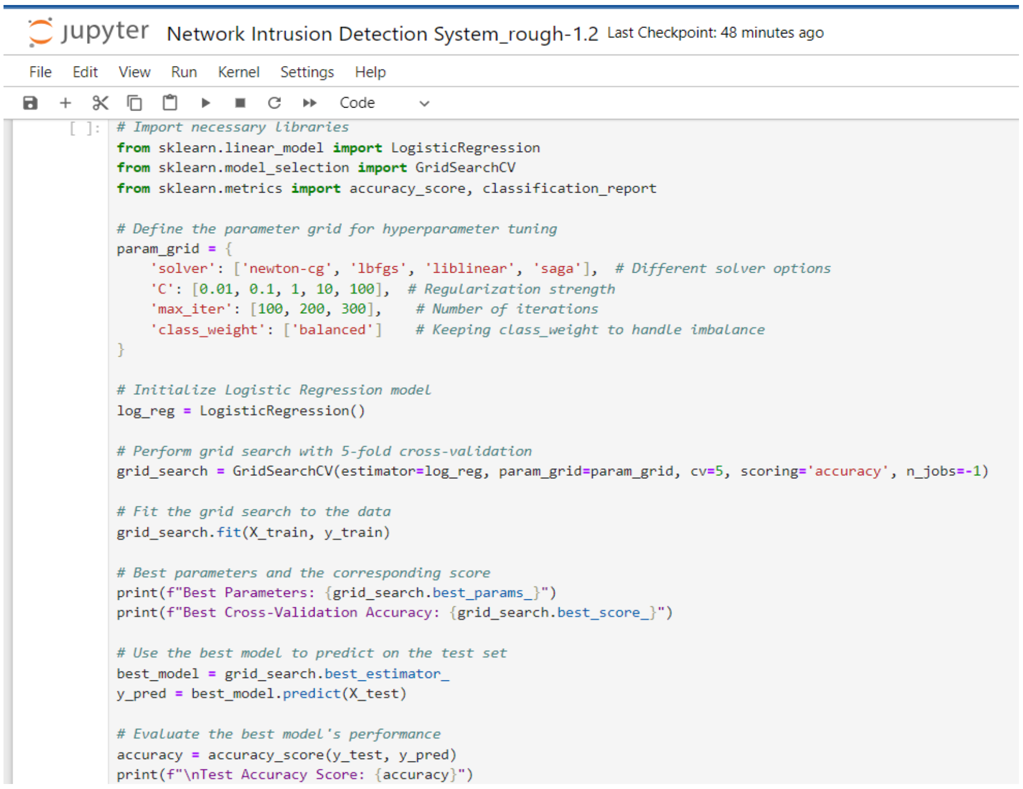


```
[ ]: # Import necessary libraries
     from sklearn.svm import SVC
     from sklearn.metrics import accuracy_score, classification_report
     from sklearn.model_selection import cross_val_score

[ ]: # Initialize the Support Vector Machine model with an RBF kernel
     svm_clf = SVC(kernel='rbf', class_weight='balanced', random_state=42)

[ ]: # Train the model on the training data
     svm_clf.fit(X_train, y_train)

[ ]: # Predict using the test data
     y_pred = svm_clf.predict(X_test)

[ ]: # Evaluate accuracy
     accuracy = accuracy_score(y_test, y_pred)
     print(f"Accuracy Score: {accuracy}")

[ ]: # Generate a classification report for more detailed evaluation
     print("\nClassification Report:")
     print(classification_report(y_test, y_pred, zero_division=1))
```

5.4 Hyperparameter Tuning

Grid Search Cross-Validation was applied to optimize the hyperparameters of each model. The GridSearchCV function from scikit-learn was used to search over a grid of hyperparameters, selecting the combination that resulted in the best performance.

5.4.1 Logistic Regression Tuning

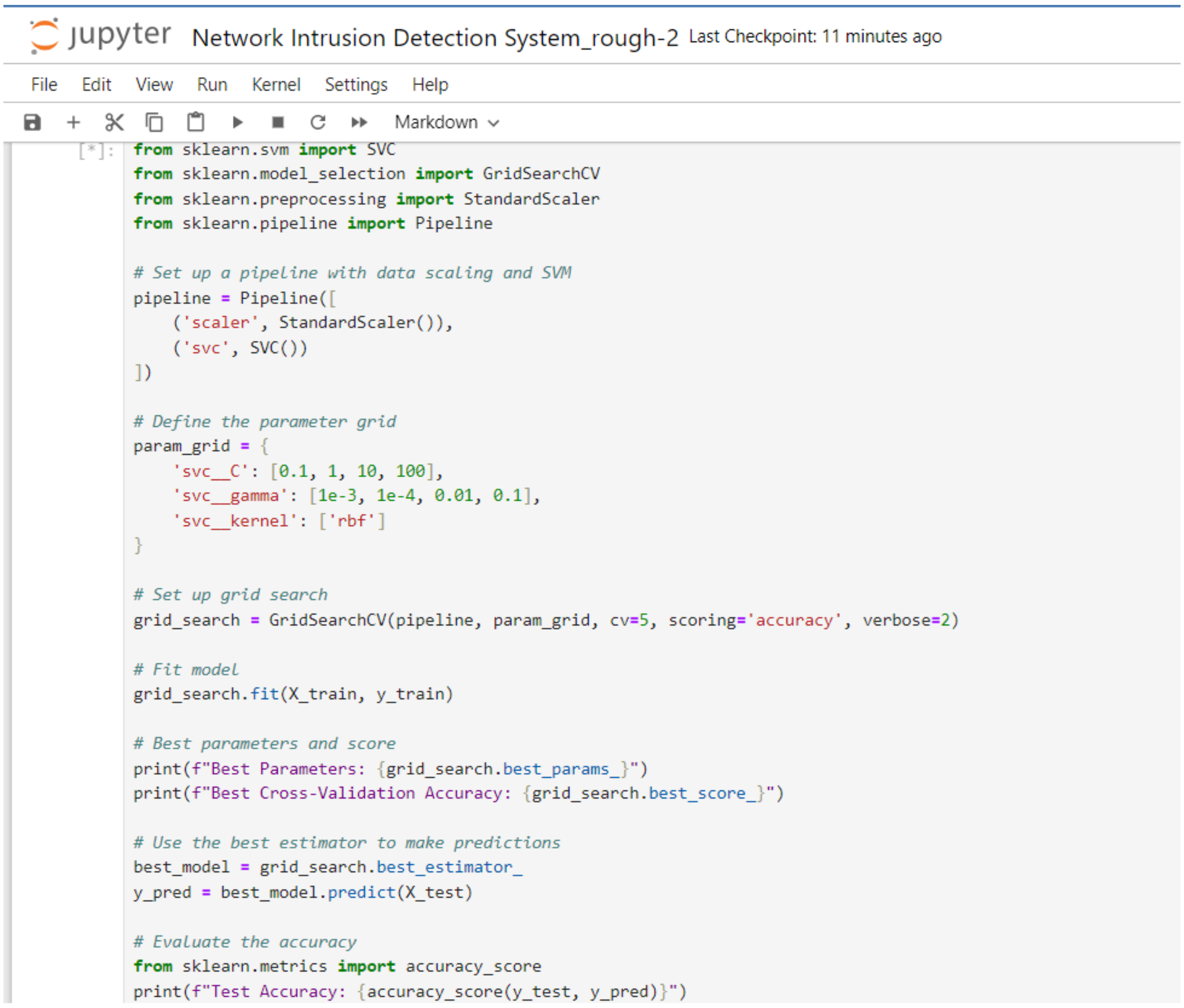

jupyter
Network Intrusion Detection System_rough-1.2 Last Checkpoint: 48 minutes ago
File Edit View Run Kernel Settings Help
Code
[ ]:

```python
# Import necessary libraries
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, classification_report

# Define the parameter grid for hyperparameter tuning
param_grid = {
    'solver': ['newton-cg', 'lbfgs', 'liblinear', 'saga'],  # Different solver options
    'C': [0.01, 0.1, 1, 10, 100],  # Regularization strength
    'max_iter': [100, 200, 300],    # Number of iterations
    'class_weight': ['balanced']    # Keeping class_weight to handle imbalance
}

# Initialize Logistic Regression model
log_reg = LogisticRegression()

# Perform grid search with 5-fold cross-validation
grid_search = GridSearchCV(estimator=log_reg, param_grid=param_grid, cv=5, scoring='accuracy', n_jobs=-1)

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# Best parameters and the corresponding score
print(f"Best Parameters: {grid_search.best_params_}")
print(f"Best Cross-Validation Accuracy: {grid_search.best_score_}")

# Use the best model to predict on the test set
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)

# Evaluate the best model's performance
accuracy = accuracy_score(y_test, y_pred)
print(f"\nTest Accuracy Score: {accuracy}")
```

5.4.2 Random Forest Tuning

```python
from sklearn.model_selection import GridSearchCV

# Define the parameter grid for grid search
param_grid = {
    'n_estimators': [100, 200, 300, 400],       # Fewer values for grid search
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': ['auto', 'sqrt'],
    'bootstrap': [True, False]
}

# Initialize Random Forest model
rf = RandomForestClassifier()

# Perform grid search with 5-fold cross-validation
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=5, n_jobs=-1, verbose=2)

# Fit the grid search model to the data
grid_search.fit(X_train, y_train)

# Best parameters and the corresponding score
print(f"Best Parameters: {grid_search.best_params_}")
print(f"Best Cross-Validation Accuracy: {grid_search.best_score_}")

# Use the best model to predict on the test set
best_rf_model = grid_search.best_estimator_
y_pred = best_rf_model.predict(X_test)

# Evaluate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"\nTest Accuracy Score: {accuracy}")
```

5.4.3 SVM Tuning


jupyter
Network Intrusion Detection System_rough-2 Last Checkpoint: 11 minutes ago
File Edit View Run Kernel Settings Help
Markdown
[*]:

```python
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

# Set up a pipeline with data scaling and SVM
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('svc', SVC())
])

# Define the parameter grid
param_grid = {
    'svc__C': [0.1, 1, 10, 100],
    'svc__gamma': [1e-3, 1e-4, 0.01, 0.1],
    'svc__kernel': ['rbf']
}

# Set up grid search
grid_search = GridSearchCV(pipeline, param_grid, cv=5, scoring='accuracy', verbose=2)

# Fit model
grid_search.fit(X_train, y_train)

# Best parameters and score
print(f"Best Parameters: {grid_search.best_params_}")
print(f"Best Cross-Validation Accuracy: {grid_search.best_score_}")

# Use the best estimator to make predictions
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)

# Evaluate the accuracy
from sklearn.metrics import accuracy_score
print(f"Test Accuracy: {accuracy_score(y_test, y_pred)}")
```

5.5 Model Evaluation

The performance of each model was evaluated using standard classification metrics such as **accuracy**, **precision**, **recall**, and **F1-score**.

```
from sklearn.metrics import accuracy_score
# Evaluate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy Score: {accuracy}")
```

```
Accuracy Score: 0.8291709178015348
```

```
# Generate a classification report for more detailed evaluation
print("\nClassification Report:")
print(classification_report(y_test, y_pred, zero_division=1))
```

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

           0       1.00      0.99      0.99      9711
           1       0.84      0.94      0.88      7459
           2       0.46      0.86      0.60      2421
           3       1.00      0.00      0.00      2885
           4       1.00      0.00      0.00        67

    accuracy                           0.83     22543
   macro avg       0.86      0.56      0.50     22543
weighted avg       0.89      0.83      0.79     22543
```

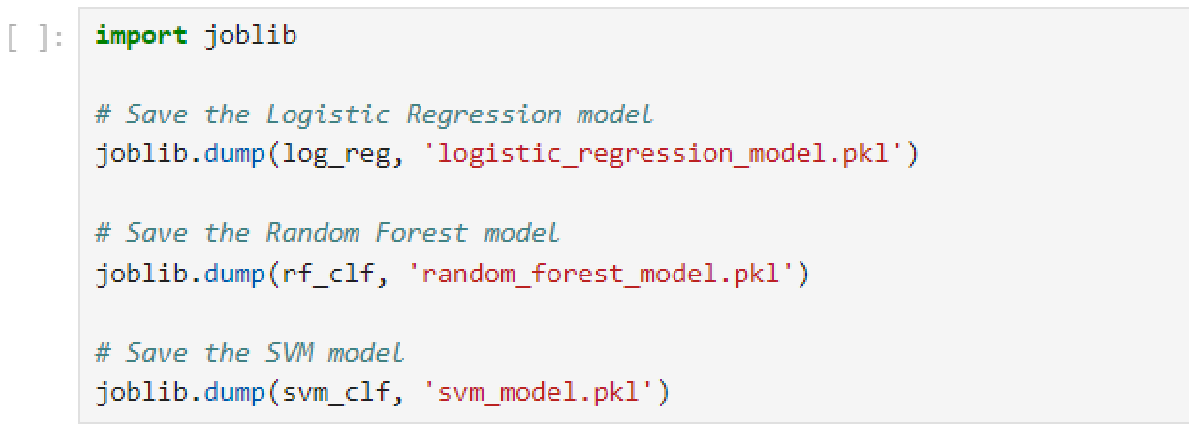
5.6 Saving and Loading the Model

Once the best-performing machine learning models were trained and evaluated, they were saved for future use. This prevents the need to retrain the models from scratch, which can be computationally expensive, particularly for models like **Random Forest** and **Support Vector Machine (SVM)**.

The models were saved using the **joblib** library, which is efficient for saving models trained on large datasets. The models can be reloaded later for making predictions or further fine-tuning.

5.6.1 Saving the Model

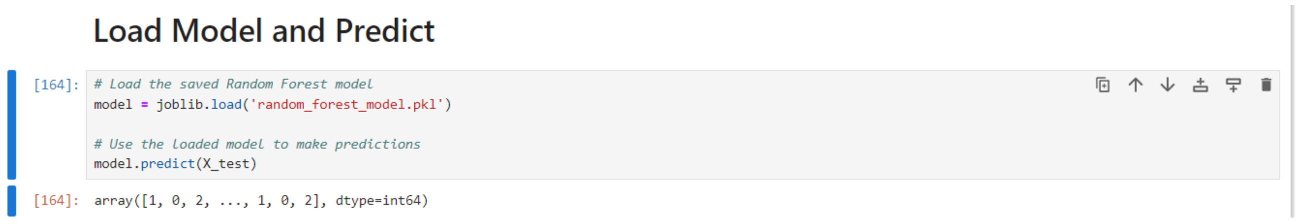
Each of the final trained models was saved to a file. For example, the Logistic Regression, Random Forest, and SVM models were saved as follows:



```
[ ]: import joblib

     # Save the Logistic Regression model
     joblib.dump(log_reg, 'logistic_regression_model.pkl')

     # Save the Random Forest model
     joblib.dump(rf_clf, 'random_forest_model.pkl')

     # Save the SVM model
     joblib.dump(svm_clf, 'svm_model.pkl')
```

5.6.2 Loading the Model

The saved models can be loaded whenever predictions are required. For example, to load the Random Forest model for future use:



Load Model and Predict

```
[164]: # Load the saved Random Forest model
       model = joblib.load('random_forest_model.pkl')

       # Use the loaded model to make predictions
       model.predict(X_test)

[164]: array([1, 0, 2, ..., 1, 0, 2], dtype=int64)
```

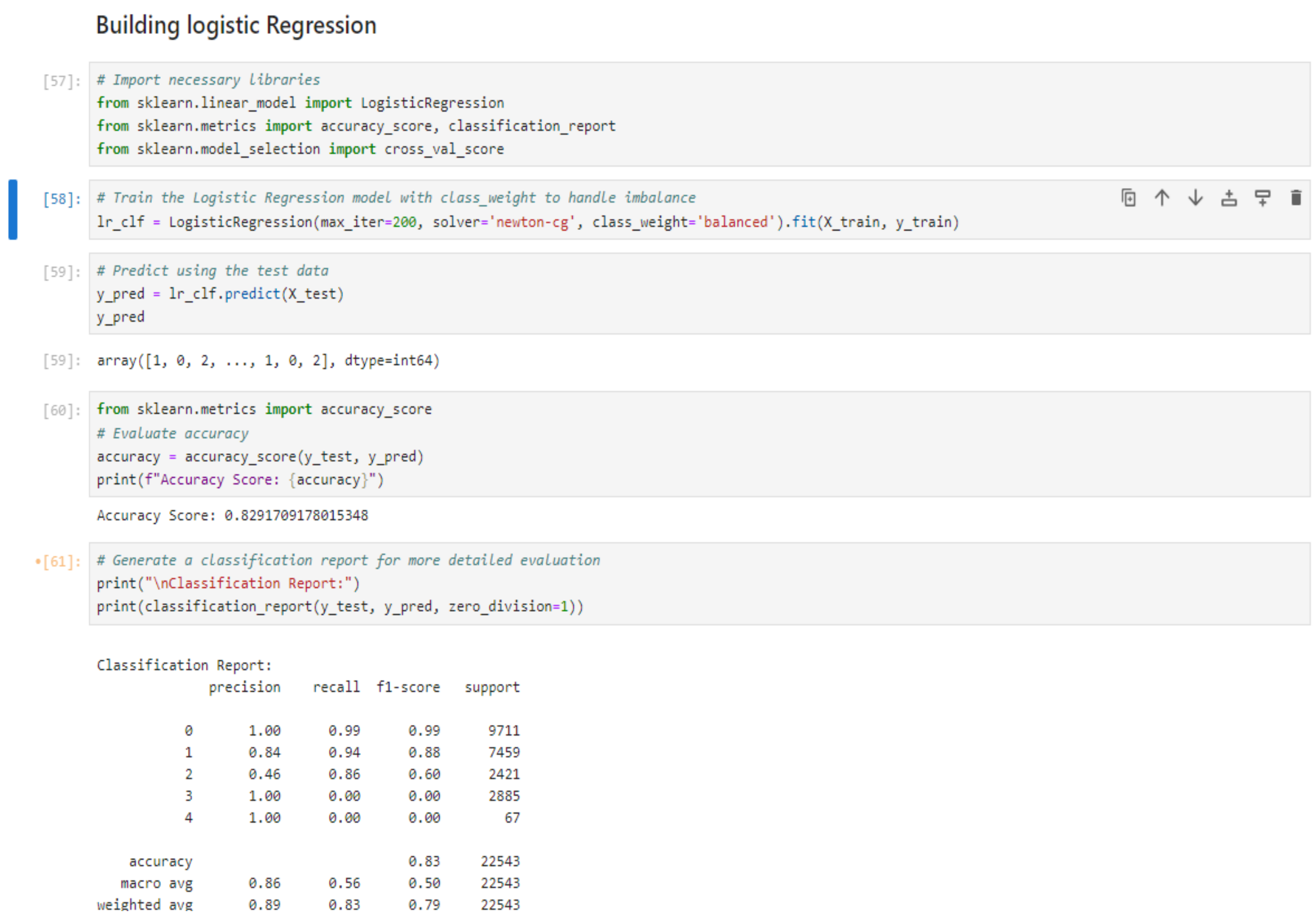
Chapter 6: Results and Analysis

6.1 Overview

I evaluated three machine learning algorithms—Logistic Regression, Random Forest, and Support Vector Machine (SVM)—on the Network Intrusion Detection dataset. Below is a summary of their performance based on accuracy, classification report metrics (precision, recall, and F1-score), and cross-validation results.

6.2 Performance Metrics

1. Logistic Regression:



Building logistic Regression

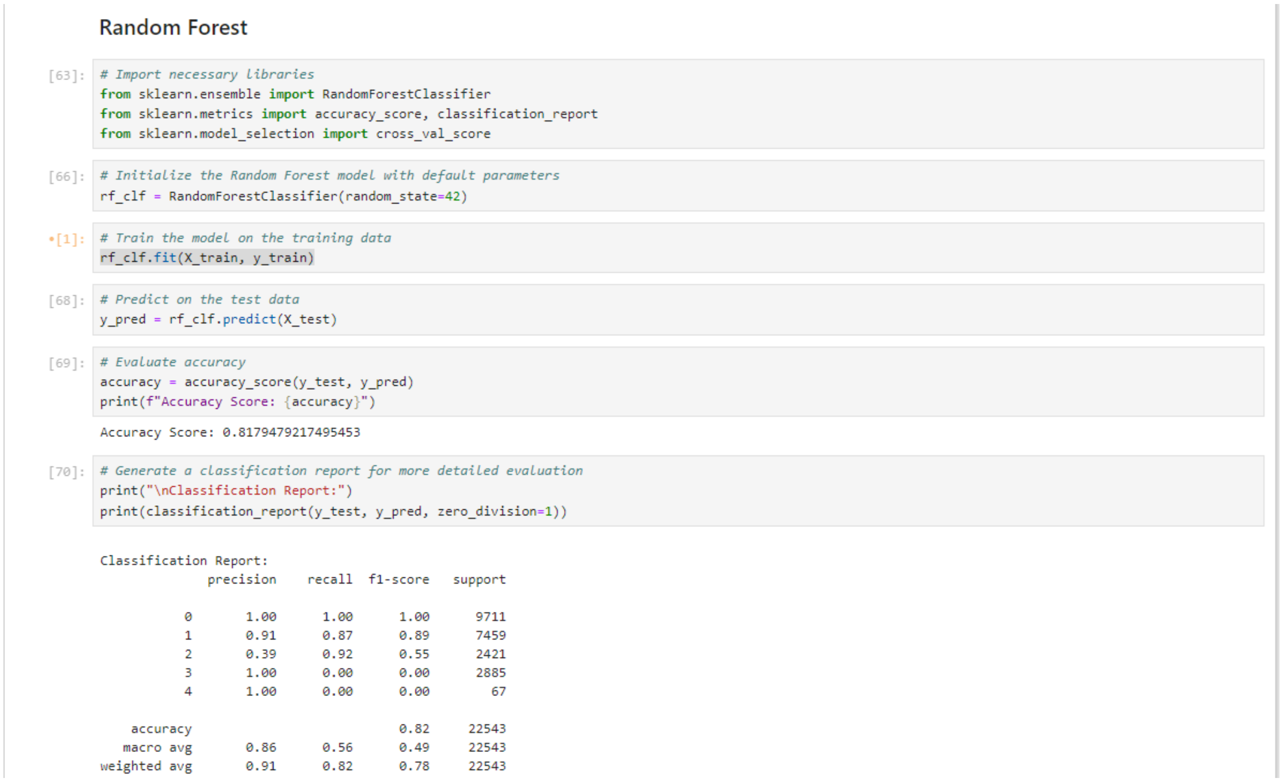
```
[57]: # Import necessary libraries
      from sklearn.linear_model import LogisticRegression
      from sklearn.metrics import accuracy_score, classification_report
      from sklearn.model_selection import cross_val_score

[58]: # Train the Logistic Regression model with class_weight to handle imbalance
      lr_clf = LogisticRegression(max_iter=200, solver='newton-cg', class_weight='balanced').fit(X_train, y_train)

[59]: # Predict using the test data
      y_pred = lr_clf.predict(X_test)
      y_pred

[59]: array([1, 0, 2, ..., 1, 0, 2], dtype=int64)

[60]: from sklearn.metrics import accuracy_score
      # Evaluate accuracy
      accuracy = accuracy_score(y_test, y_pred)
      print(f"Accuracy Score: {accuracy}")

      Accuracy Score: 0.8291709178015348

•[61]: # Generate a classification report for more detailed evaluation
      print("\nClassification Report:")
      print(classification_report(y_test, y_pred, zero_division=1))


      Classification Report:
                    precision    recall  f1-score   support

                 0       1.00      0.99      0.99      9711
                 1       0.84      0.94      0.88      7459
                 2       0.46      0.86      0.60      2421
                 3       1.00      0.00      0.00      2885
                 4       1.00      0.00      0.00        67

          accuracy                           0.83     22543
         macro avg       0.86      0.56      0.50     22543
      weighted avg       0.89      0.83      0.79     22543
```

- **Accuracy:** 82.92%

- **Precision, Recall, F1-score:**
  - Class 0: Precision: 1.00, Recall: 0.99, F1-Score: 0.99
  - Class 1: Precision: 0.84, Recall: 0.94, F1-Score: 0.88
  - Class 2: Precision: 0.46, Recall: 0.86, F1-Score: 0.60
  - Class 3 & 4: The model struggles, with zero recall and F1-scores.
- **Cross-validation Mean Score:** 99.67%

Analysis: Logistic Regression performs well on the majority classes (0 and 1), but significantly underperforms on the minority classes (3 and 4). This suggests the model struggles with the class imbalance, as shown by its inability to detect any instances of classes 3 and 4.

2. Random Forest:



Random Forest

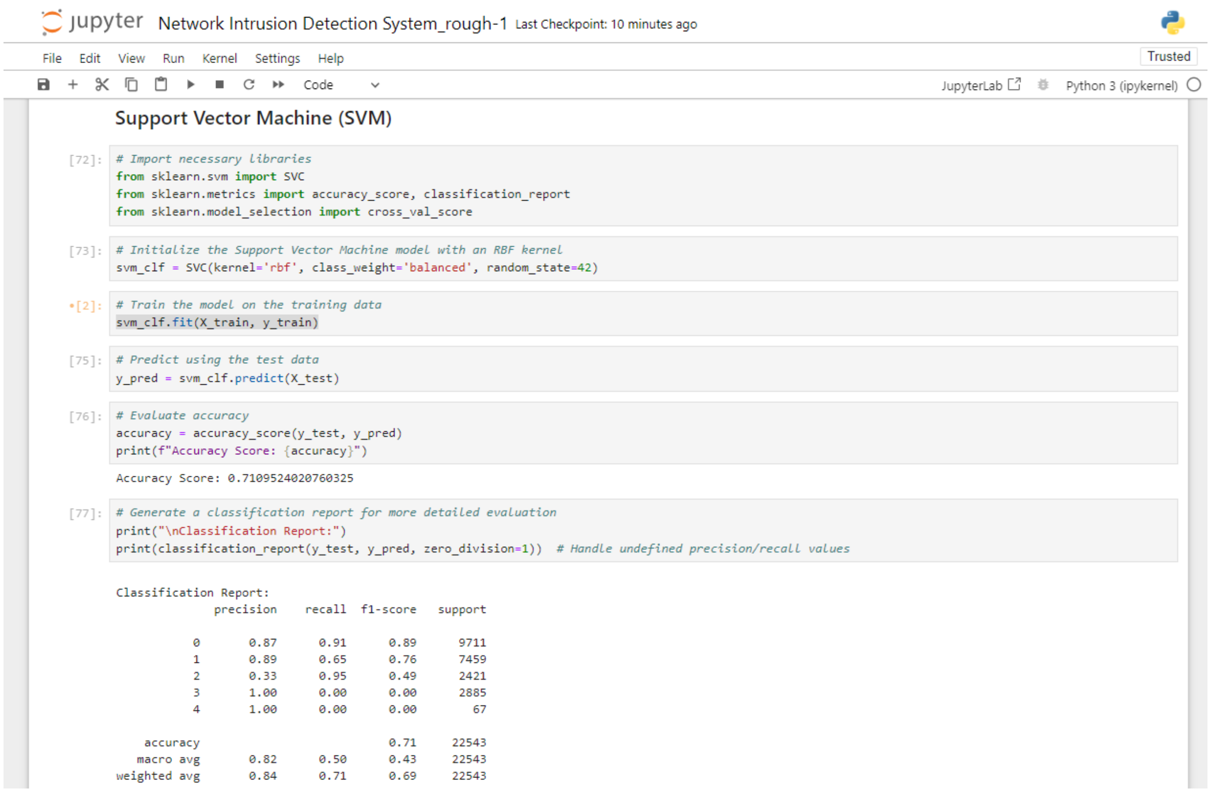
```
[63]: # Import necessary libraries
      from sklearn.ensemble import RandomForestClassifier
      from sklearn.metrics import accuracy_score, classification_report
      from sklearn.model_selection import cross_val_score

[66]: # Initialize the Random Forest model with default parameters
      rf_clf = RandomForestClassifier(random_state=42)

•[1]: # Train the model on the training data
      rf_clf.fit(X_train, y_train)

[68]: # Predict on the test data
      y_pred = rf_clf.predict(X_test)

[69]: # Evaluate accuracy
      accuracy = accuracy_score(y_test, y_pred)
      print(f"Accuracy Score: {accuracy}")

      Accuracy Score: 0.8179479217495453

[70]: # Generate a classification report for more detailed evaluation
      print("\nClassification Report:")
      print(classification_report(y_test, y_pred, zero_division=1))

      Classification Report:
                    precision    recall  f1-score   support

                 0       1.00      1.00      1.00      9711
                 1       0.91      0.87      0.89      7459
                 2       0.39      0.92      0.55      2421
                 3       1.00      0.00      0.00      2885
                 4       1.00      0.00      0.00        67

          accuracy                           0.82     22543
         macro avg       0.86      0.56      0.49     22543
      weighted avg       0.91      0.82      0.78     22543
```

- **Accuracy:** 81.79%
- **Precision, Recall, F1-score:**
  - Class 0: Precision: 1.00, Recall: 1.00, F1-Score: 1.00

    - Class 1: Precision: 0.91, Recall: 0.87, F1-Score: 0.89
    - Class 2: Precision: 0.39, Recall: 0.92, F1-Score: 0.55
    - Class 3 & 4: Zero recall and F1-scores.
- **Cross-validation Mean Score:** 99.95%

Analysis: Random Forest demonstrates strong performance for the majority classes (0 and 1) but continues to face difficulty with the minority classes, similar to Logistic Regression. It provides higher recall for class 2 but still fails to predict classes 3 and 4.

3. Support Vector Machine (SVM):


jupyter Network Intrusion Detection System_rough-1 Last Checkpoint: 10 minutes ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)

Support Vector Machine (SVM)

```
[72]: # Import necessary libraries
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report
from sklearn.model_selection import cross_val_score

[73]: # Initialize the Support Vector Machine model with an RBF kernel
svm_clf = SVC(kernel='rbf', class_weight='balanced', random_state=42)

•[2]: # Train the model on the training data
svm_clf.fit(X_train, y_train)

[75]: # Predict using the test data
y_pred = svm_clf.predict(X_test)

[76]: # Evaluate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy Score: {accuracy}")

Accuracy Score: 0.7109524020760325

[77]: # Generate a classification report for more detailed evaluation
print("\nClassification Report:")
print(classification_report(y_test, y_pred, zero_division=1))  # Handle undefined precision/recall values
```

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

           0       0.87      0.91      0.89      9711
           1       0.89      0.65      0.76      7459
           2       0.33      0.95      0.49      2421
           3       1.00      0.00      0.00      2885
           4       1.00      0.00      0.00        67

    accuracy                           0.71     22543
   macro avg       0.82      0.50      0.43     22543
weighted avg       0.84      0.71      0.69     22543
```

- **Accuracy:** 71.10%
- **Precision, Recall, F1-score:**
    - Class 0: Precision: 0.87, Recall: 0.91, F1-Score: 0.89
    - Class 1: Precision: 0.89, Recall: 0.65, F1-Score: 0.76
    - Class 2: Precision: 0.33, Recall: 0.95, F1-Score: 0.49

    - Class 3 & 4: Zero recall and F1-scores.

- **Cross-validation Mean Score:** 93.16%

Analysis: The SVM model performs the worst among the three, achieving the lowest overall accuracy. While it shows high recall for class 2, its performance for other classes, especially 1, 3, and 4, is weak. This suggests that SVM is less suited to this dataset, potentially due to the high **class imbalance.**

6.3 Comparison of Models

1. **Overall Accuracy:**
    - **Logistic Regression:** 82.92%
    - **Random Forest:** 81.79%
    - **SVM:** 71.10%

Logistic Regression achieved the highest test accuracy, closely followed by Random Forest. SVM underperformed compared to the other models.

2. **Handling Class Imbalance:**
    - All three models struggled with class imbalance, as evidenced by their failure to predict minority classes (3 and 4) correctly. Both Logistic Regression and Random Forest showed zero recall for these classes, while SVM had slightly better performance for class 2 but did poorly for others.

Despite using class weighting (class_weight='balanced'), the minority classes were still misclassified or ignored by all models, indicating that additional balancing techniques such as SMOTE or undersampling are necessary.

6.4 Conclusion and Recommendations

- **Best Model:** While Logistic Regression slightly outperformed Random Forest in test accuracy, both models demonstrated similar capabilities and limitations, especially in handling class imbalance. SVM, on the other hand, is less suitable for this dataset due to its significantly lower accuracy and inconsistent performance.

- **Recommendations:**
    - Address the class imbalance by applying SMOTE, undersampling, or other data balancing techniques to improve the performance on minority classes.
    - Consider alternative algorithms or ensemble methods, such as XGBoost or AdaBoost, which may better handle imbalanced datasets.
    - Perform further hyperparameter tuning to improve generalization and reduce overfitting, particularly in Logistic Regression and Random Forest.

Chapter 7: Discussion

This project, completed independently, faced constraints in terms of time and resources, which limited the scope to static data analysis rather than real-time detection and alerting functionalities. Expanding the project to include real-time detection and alerting would be a significant advancement in creating a fully functional Intrusion Detection System (IDS) for real-world environments. Working alone on this project presented unique challenges, but it also offered invaluable experience in understanding the intricacies of machine learning-based IDS development.

This chapter provides an in-depth discussion of the findings from the model evaluation, the challenges encountered during the project, and recommendations for future work. The results obtained from the machine learning models provide insight into both the strengths and limitations of current approaches to Intrusion Detection Systems (IDS) using the NSL-KDD dataset. This chapter also highlights opportunities for further improvement and research in the field of network intrusion detection.

7.1 Key Findings

The evaluation of three machine learning models—Logistic Regression, Random Forest, and Support Vector Machine (SVM)—revealed several important findings:

1. **High Accuracy for Common Attacks**: Both Logistic Regression and Random Forest achieved relatively high accuracy (82.91% and 81.79%, respectively) in detecting common attack types such as **Denial of Service (DoS)** and **Probe**. These attack types were well-represented in the dataset, and the models were able to effectively learn patterns for their detection.
2. **Challenges with Rare Attacks**: A major challenge observed across all models was the difficulty in detecting rare attack types like **Remote to Local (R2L)** and **User to Root (U2R)**. These attack types are critical to network security, as they can allow unauthorized users to gain access to a system and potentially escalate privileges to a root level. However, the imbalance in the dataset caused the models to focus primarily on more frequent attack types, resulting in poor recall for R2L and U2R attacks.

3. **Effectiveness of Class Weighting**: The use of class weighting in the models (via the class_weight='balanced' parameter) helped mitigate the impact of class imbalance, but it was not sufficient to fully address the issue. Although the models performed reasonably well for frequent attack types, the rare classes continued to be under-predicted, indicating that more advanced techniques are needed to handle imbalanced datasets in intrusion detection effectively.

4. **Model Comparisons**:

   - **Logistic Regression**: Despite being a linear model, Logistic Regression performed strongly, achieving the highest accuracy overall. This model's simplicity and interpretability make it a strong candidate for detecting frequent attack types.
   - **Random Forest**: Random Forest performed comparably to Logistic Regression but had the added benefit of handling non-linear relationships in the data. It was also more flexible but slightly less accurate overall.
   - **SVM**: Although SVM is powerful for non-linear problems, it struggled with the imbalanced dataset, leading to the lowest accuracy (71.10%). This suggests that SVM may require further adjustments, such as incorporating sampling techniques or advanced kernels, to perform well in this context.

7.2 Challenges and Limitations

Several challenges and limitations were encountered during the project, which affected the performance of the models:

1. **Class Imbalance**: The primary challenge was the class imbalance in the NSL-KDD dataset. The imbalance heavily favoured normal traffic and **DoS** attacks, with much fewer examples of **R2L** and **U2R** attacks. Despite using class weighting, the models still struggled to detect these rare attacks, highlighting the limitations of traditional machine learning models in dealing with imbalanced datasets.

2. **Limited Generalization**: While the models achieved relatively high accuracy on the test set, their ability to generalize to more sophisticated or previously unseen attack vectors remains uncertain. The NSL-KDD dataset, while widely used, does not

encompass all possible types of network attacks, meaning that the models might underperform in real-world settings where new and evolving attacks are more common.

3. **Computational Complexity**: Training models like Random Forest and SVM on large datasets can be computationally expensive. In particular, hyperparameter tuning using Grid Search Cross-Validation required significant computational resources and time. This limitation could affect scalability in real-world deployments where quick adaptation and tuning are necessary.

7.3 Recommendations for Future Work

To overcome the challenges identified in this study and further enhance the effectiveness of Intrusion Detection Systems, several areas of improvement can be explored:

1. **Advanced Imbalance Handling**: Beyond class weighting, techniques such as **Synthetic Minority Over-sampling Technique (SMOTE)** or **undersampling** can be applied to balance the dataset more effectively. SMOTE generates synthetic examples of minority classes, while undersampling reduces the number of examples from majority classes, allowing the model to focus more on rare attack types.

2. **Deep Learning Approaches**: Deep learning models, particularly **Convolutional Neural Networks (CNNs)** and **Recurrent Neural Networks (RNNs)**, have shown promise in detecting complex attack patterns that traditional machine learning models may miss. These models can automatically learn feature representations from raw network traffic, potentially improving the detection of rare and sophisticated attacks like **R2L** and **U2R**.

3. **Anomaly Detection Techniques**: In addition to supervised machine learning models, unsupervised or semi-supervised approaches can be explored for anomaly detection. Models like **Autoencoders** or **Isolation Forest** are designed to detect deviations from normal network behaviour, which could be useful in identifying zero-day attacks or attacks not represented in the training data.

4. **Real-Time IDS Deployment**: Future work could focus on deploying these models in a real-time intrusion detection environment. This would involve integrating the IDS into live network traffic monitoring systems, where the models continuously learn

from new data and adapt to emerging threats. Additionally, further optimization of model inference time would be needed to ensure that predictions are made in real-time.

5. **Ensemble Methods**: While this study did not focus on ensemble methods like **bagging**, **boosting**, or **voting**, these techniques could be explored to combine the strengths of multiple models. For example, combining Logistic Regression, Random Forest, and SVM in a voting classifier could lead to improved detection accuracy by leveraging the advantages of each model.

7.4 Practical Implications

The findings of this study demonstrate that machine learning-based Intrusion Detection Systems can effectively detect common attack types such as **DoS** and **Probe**. However, the study also highlights the limitations of current approaches in handling rare and sophisticated attacks like **R2L** and **U2R**. The practical implication is that while machine learning can enhance IDS performance, additional techniques are necessary to make these systems robust against a wider range of attack types.

This study provides a foundation for the development of more sophisticated IDS, particularly in how class imbalance is handled and how models can be optimized for real-world deployment. As networks continue to evolve and become more complex, intrusion detection will remain a critical area of research and development.

References:

1. Kumar, S., & Spafford, E. H. (1995). A software architecture to support misuse intrusion detection. *Proceedings of the 18th National Information Systems Security Conference*, 194–204.
2. Lee, W., Stolfo, S. J., & Mok, K. W. (1999). A data mining framework for building intrusion detection models. *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 120–132.
3. Sung, A. H., & Mukkamala, S. (2003). Identifying important features for intrusion detection using support vector machines and neural networks. *Proceedings of the 2003 Symposium on Applications and the Internet (SAINT'03)*, 209–216.
4. Tavallaee, M., Bagheri, E., Lu, W., & Ghorbani, A. A. (2009). A detailed analysis of the KDD CUP 99 dataset. *Proceedings of the 2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA)*, 1–6.
5. Mishra, R., Gupta, M., & Sharma, S. (2018). An efficient intrusion detection system using deep learning technique. *International Journal of Scientific Research in Science and Technology*, 5(4), 203-211.
6. Tan, Z., Jamdagni, A., He, X., Nanda, P., & Liu, R. P. (2014). A system for denial-of-service attack detection based on multivariate correlation analysis. *IEEE Transactions on Parallel and Distributed Systems*, 25(2), 447–456.
7. Zhang, J., Zulkernine, M., & Haque, A. (2008). Random-forest-based network intrusion detection systems. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(5), 649–659.
8. NSL-KDD dataset. (n.d.). Canadian Institute for Cybersecurity, University of New Brunswick. Retrieved from *https://www.unb.ca/cic/datasets/nsl.html*
9. Scikit-learn: Machine learning in Python. (n.d.). Retrieved from *https://scikit-learn.org/stable/*
10. IBM: The importance of anomaly detection in network security. *IBM Security Intelligence*. Retrieved from *https://www.ibm.com/security/anomaly-detection-network-security*
11. Kaspersky: The evolving landscape of network intrusion detection. *Kaspersky Lab Resources*. Retrieved *from* *https://www.kaspersky.com/blog/network-intrusion-detection-evolution/*
12. Norton: How to protect your network from probe attacks. *Norton Security Centre*. Retrieved from *https://us.norton.com/protect-network-from-probe-attacks*