CMR ENGINEERING COLLEGE
(UGC AUTONOMOUS)
(ACCREDITED BY NAAC AND NBA , APPROVED BY AICTE NEW DELHI, AFFILIATED TO JNTU HYDERABAD)
KANDLAKOYA, MEDCHAL ROAD, HYDERABAD-501 401.

Department of Cyber Security

A

Mini Project
On

CipherLink SecureShare: Encrypted File-Sharing Application with Uncompromising Security and User-Friendly Interface

B.SAI RAGHAVENDRA	**: (228R1A62B4)**
KURRA JAGRUTHI	**: (228R1A6297)**
CHINTHALA VENKAT SAI	**: (228R1A6283)**

Under the Guidance of,
MR VENKAT
Assistant Professor,
Dept. of CS.

A.Y 2024-2025

ABSTRACT

In today's increasingly interconnected digital environment, securing networked systems is crucial for protecting sensitive information, ensuring user privacy, and maintaining uninterrupted operations. The rapid rise in cyber threats such as malware, phishing, ransomware, and advanced persistent threats poses significant challenges to organizations worldwide. Traditional Intrusion Detection Systems (IDS), which often rely on rule-based or signature-based detection methods like those used in Random Forest models, have limitations. These systems tend to generate high false positive rates and struggle to detect new or evolving attack patterns, making them less effective in dynamic and complex threat landscapes.To overcome these challenges, this paper proposes an enhanced machine learning-based IDS that combines the strengths of k-Nearest Neighbors (kNN), Random Forest, and Decision Tree algorithms. By leveraging the unique capabilities of each model, our system significantly improves threat detection accuracy while minimizing false positives. This hybrid approach not only enhances the adaptability of the IDS to evolving threats but also enables real-time analysis and classification of network traffic. Through advanced analytics and continuous learning from data, the proposed system offers a more intelligent, flexible, and proactive defense mechanism against modern cyber attacks.

INTRODUCTION

In today’s digitally driven world, network security is of utmost importance for protecting sensitive information, maintaining user privacy, and ensuring the smooth functioning of organizational operations. With the rapid increase in cyberattacks such as malware, phishing, ransomware, and zero-day exploits, traditional security mechanisms are proving to be insufficient. These conventional Intrusion Detection Systems (IDS), which often rely on rule-based or signature-based methods, struggle to detect novel threats and typically generate a high number of false positives. As threats continue to evolve in both frequency and complexity, there is a pressing need for intelligent and adaptive systems that can identify and respond to suspicious activities more effectively and accurately.

Our project addresses this challenge by developing an advanced IDS that leverages the capabilities of machine learning algorithms—specifically k-Nearest Neighbors (kNN), Random Forest, and Decision Tree. This hybrid approach combines the strengths of each algorithm to improve threat detection accuracy while reducing false alarms. The kNN algorithm helps detect anomalies based on behavioral similarities, Random Forest contributes robustness and high accuracy, and Decision Tree enhances model interpretability and speed. Together, they form a comprehensive and dynamic detection system. We conducted extensive experiments using benchmark datasets to validate the effectiveness of our model. Results show that our IDS not only detects a wide variety of threats but also adapts to evolving attack patterns with minimal human intervention. By continuously learning from new data, the system maintains high performance in real-time environments. This project makes a meaningful contribution to modern cybersecurity by providing a scalable, intelligent, and proactive solution that addresses the limitations of traditional IDS technologies.

LITERATURE SURVEY

S.No	Title	Authors	Techniques Used	Key Findings
1	Network Intrusion Detection System using Deep Learning	Reshni S, Navateja V, Naveen V, Kumar R, Praveen K	Deep Learning, Autoencoder	Achieved high accuracy using a two-stage DL model on NSL-KDD and Bot-IoT data.
2	Network Intrusion Detection System: A Systematic Study of ML and DL Approaches	Zeeshan Ahmad, Adnan Shahid Khan, Cheah Wai Shiang, Johari Abdullah, Farhan Ahmad	Machine Learning, Deep Learning	Reviewed IDS techniques; noted issues with false alarms and new attack detection
3	Cyber Intrusion Detection System Using Deep Learning Approach	Reshni S, Vadipalli N, Vinukonda N, Kumar R, Praveen K	Deep Learning, Autoencoder, Grey Wolf Optimization	Proposed a two-stage IDS with feature selection, achieving improved accuracy on Bot-IoT and NSL-KDD datasets.
4	Network Intrusion Detection System using Machine Learning and Deep Learning Approach	Chauhan, N.	Machine Learning, Deep Learning	Discussed ML and DL methods for IDS; emphasized the importance of dataset selection and evaluation metrics.

EXISTING SYSTEMS

Most traditional Intrusion Detection Systems (IDS) rely heavily on rule-based or signature-based detection techniques. These systems monitor network traffic and compare it against a predefined database of known attack signatures or behavioral rules. A commonly used algorithm in such systems is the Random Forest, valued for its classification performance and ease of implementation. While Random Forest can achieve decent accuracy in detecting known threats, its effectiveness diminishes when facing novel or obfuscated attacks, as it depends largely on labeled historical data.

Limitations of the existing systems

• High False Positive Rate: Signature-based systems often flag normal but uncommon behavior as malicious, leading to frequent false alerts and making them impractical in dynamic network environments.

• Limited Adaptability: These systems struggle to recognize new or evolving threats since they cannot generalize beyond their predefined rules or training data.

• Static Detection Logic: Traditional IDSs lack real-time learning capabilities, which limits their ability to adapt to changing network patterns or sophisticated attack strategies.

• Scalability Issues: As network traffic volume grows, maintaining accuracy and speed becomes challenging, especially for single-model systems without optimization for performance..

PROPOSED SYSTEMS

The proposed Intrusion Detection System (IDS) utilizes a hybrid machine learning approach by combining three powerful algorithms: k-Nearest Neighbors (kNN), Random Forest, and Decision Tree. Each algorithm contributes distinct strengths to improve overall system performance. The kNN algorithm classifies network traffic by analyzing the similarity between new data points and known patterns, making it effective for detecting anomalies. Random Forest enhances classification by constructing multiple decision trees and aggregating their predictions, which increases accuracy and robustness. The Decision Tree algorithm, known for its interpretability and speed, partitions the feature space to differentiate between normal and malicious traffic. This ensemble method addresses the limitations of traditional IDS by enhancing detection accuracy, improving adaptability to evolving cyber threats, and significantly reducing false positive rates. Additionally, the combination of these algorithms ensures a balance between computational efficiency and performance, making the system scalable for real-time intrusion detection in dynamic network environments.

Benefits of the Proposed System:

• Enhanced Accuracy: The ensemble model improves overall detection accuracy by leveraging the strengths of multiple classifiers, reducing the likelihood of misclassifications.

• Improved Adaptability: The use of diverse algorithms allows the system to better adjust to new and evolving threats, adapting to changes in network traffic behavior and patterns.

• Reduced False Positives: By aggregating predictions from multiple models, the system mitigates false alarms and enhances the reliability of alerts.

SYSTEM SPECIFICATIONS

SOFTWARE REQUIREMENTS

Tools Used:

- Anaconda Navigator
- Jupyter Notebook

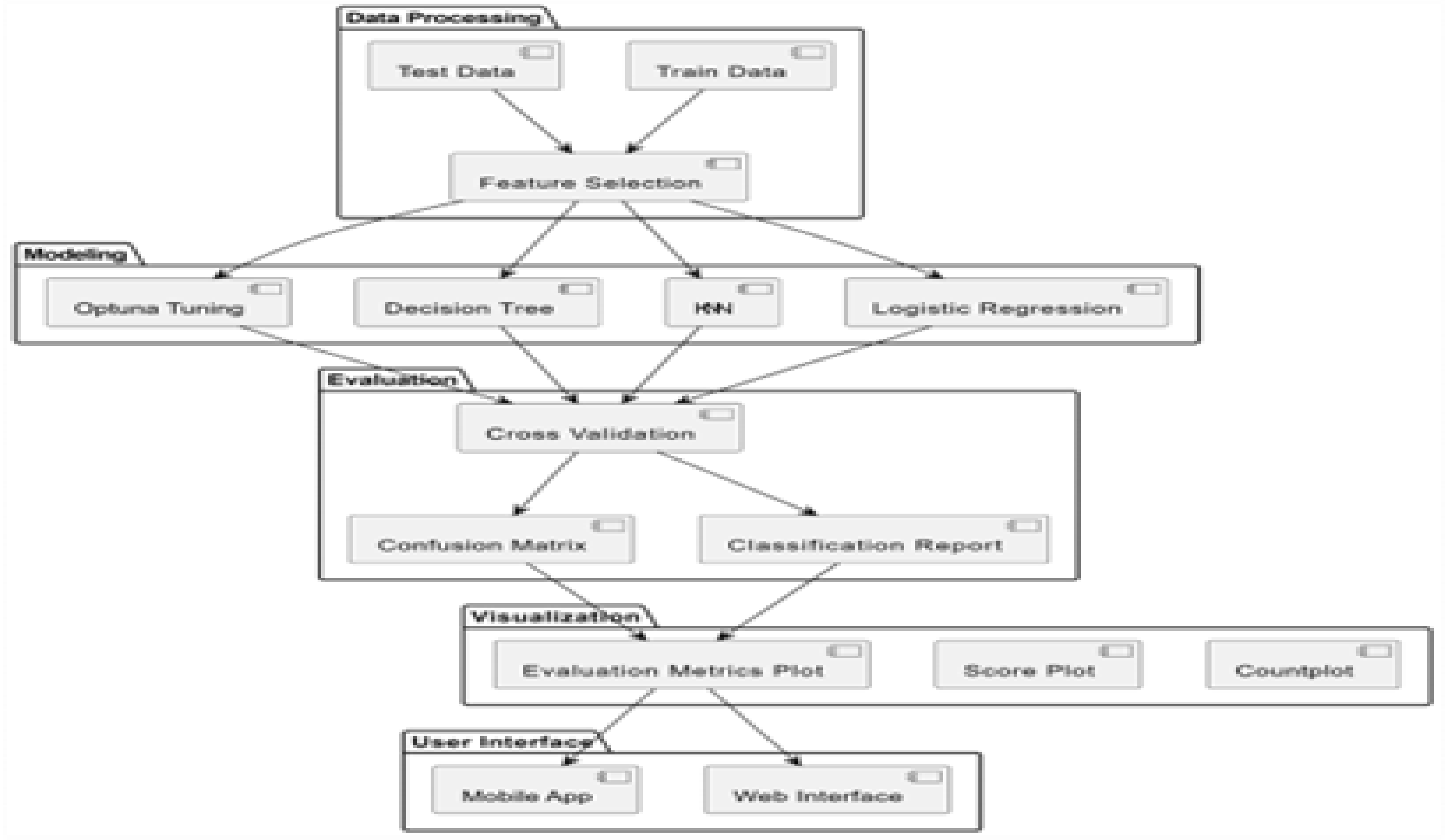
Packages Used:

- pandas
- scikit-learn
- matplotlib
- seaborn
- numpy
- lightgbm

HARDWARE REQUIREMENTS

- RAM: 8 GB
- Storage: 512 GB
- Processor: Intel Core i5
- Graphics: 2 GB GPU

SYSTEM ARCHITECTURE



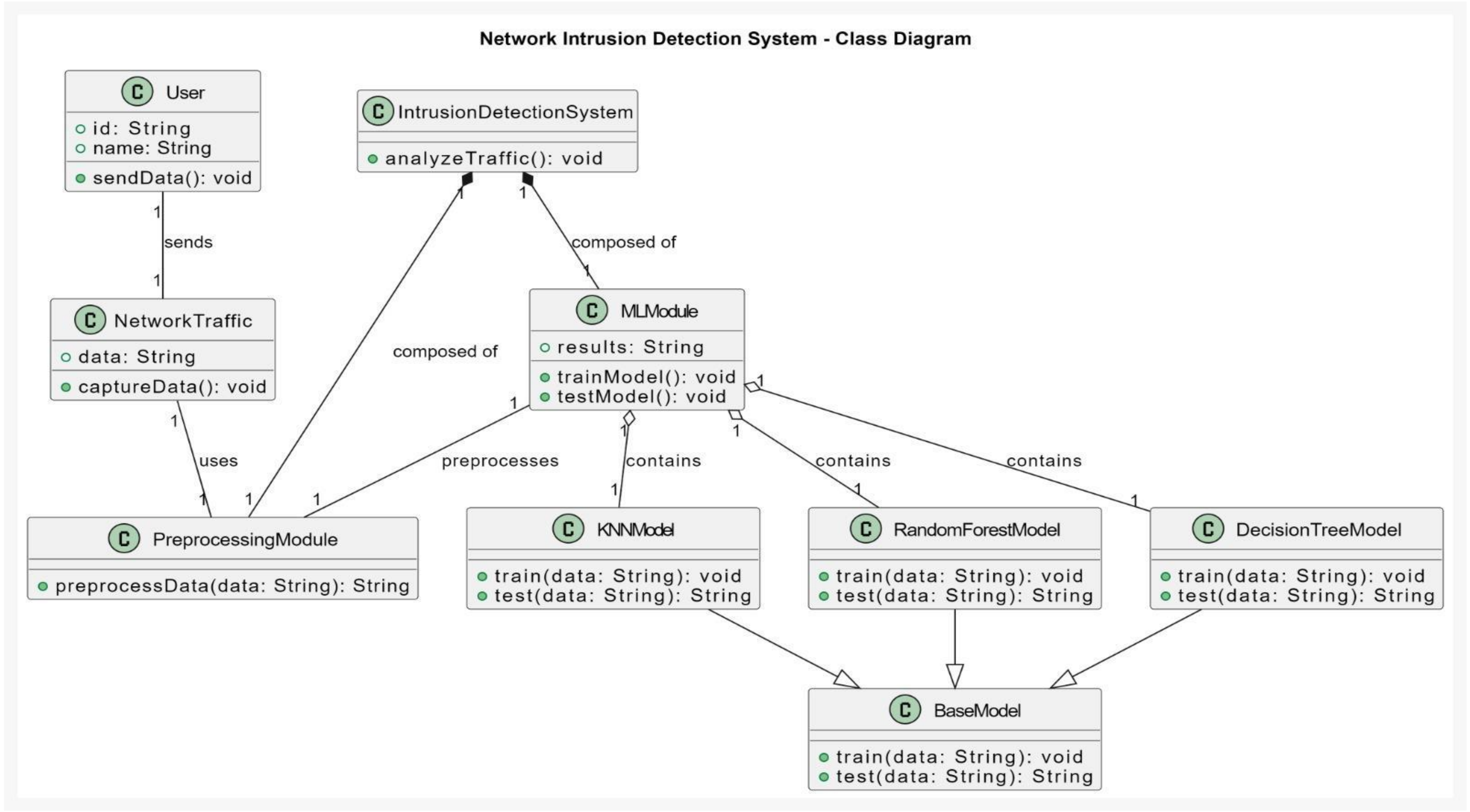
SYSTEM ARCHITECTURE

It presents a systematic workflow for a machine learning pipeline that begins with data processing and proceeds through modeling, evaluation, visualization, and user interface development. In the data processing stage, raw data is split into train and test datasets, followed by a crucial feature selection process to identify the most relevant attributes for model training. The selected features are then passed into the modeling phase, where multiple algorithms are employed—such as Decision Trees, K-Nearest Neighbors (KNN), and Logistic Regression—alongside Optuna Tuning, a hyperparameter optimization technique. These models are refined and validated through a robust cross-validation process under the evaluation stage.

The evaluation stage further expands into the generation of key performance insights using a confusion matrix and a classification report, offering metrics like accuracy, precision, recall, and F1-score. These insights feed into the visualization module, where results are translated into intuitive visual formats such as evaluation metric plots, score plots, and count plots to support interpretability. Finally, the pipeline culminates in the user interface development, which includes both mobile app and web interface integrations, enabling end-users to interact with the analytical outcomes. This entire structure ensures that data-driven decisions are well-informed, reproducible, and user-accessible, streamlining the workflow for practical machine learning applications.

UML DIAGRAMS

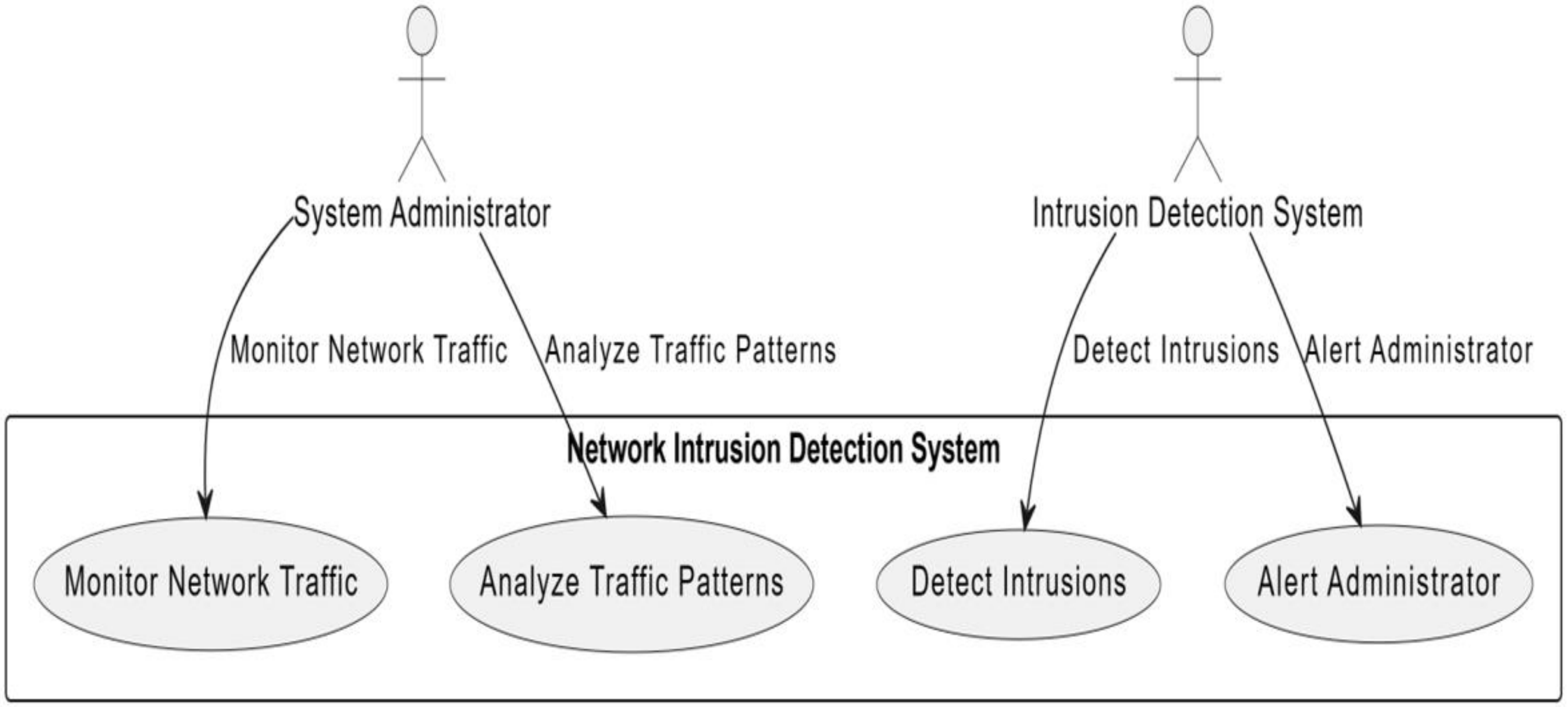
CLASS DIAGRAM



- The class diagram provided represents a Network Intrusion Detection System (NIDS), illustrating how different components interact to analyze and detect malicious network traffic using machine learning. At the top level, the system is driven by the IntrusionDetectionSystem class, which contains a method analyzeTraffic() and is composed of two key components: MLModule and PreprocessingModule. The system interacts with network data through the NetworkTraffic class, which is responsible for capturing the data via captureData() and holds a data: String attribute. This network traffic is generated by the User class, which sends data to the system using the sendData() method. The User is associated with a unique id and name, making it traceable in the system. The PreprocessingModule, in turn, is tightly integrated with the NetworkTraffic class, performing data cleaning or transformation through its preprocessData(data: String): String method. This preprocessed data is passed to the MLModule for model-based analysis.

- The MLModule is the core of the learning system, responsible for training and testing using various machine learning algorithms. It contains an attribute results: String and two methods: trainModel() and testModel(), which facilitate model evaluation. This module houses three distinct models—KNNModel, RandomForestModel, and DecisionTreeModel—each of which contains its own implementations of train(data: String): void and test(data: String): String. These models extend a common BaseModel, promoting polymorphism and consistency across different algorithms. This design allows easy integration of new models while maintaining structural uniformity. All these components together reflect a well-structured and modular system where raw data is captured, preprocessed, and analyzed using sophisticated ML techniques, enabling effective detection of network intrusions. The relationships between classes (association, composition, and inheritance) are clearly defined, ensuring maintainability and scalability of the detection framework.

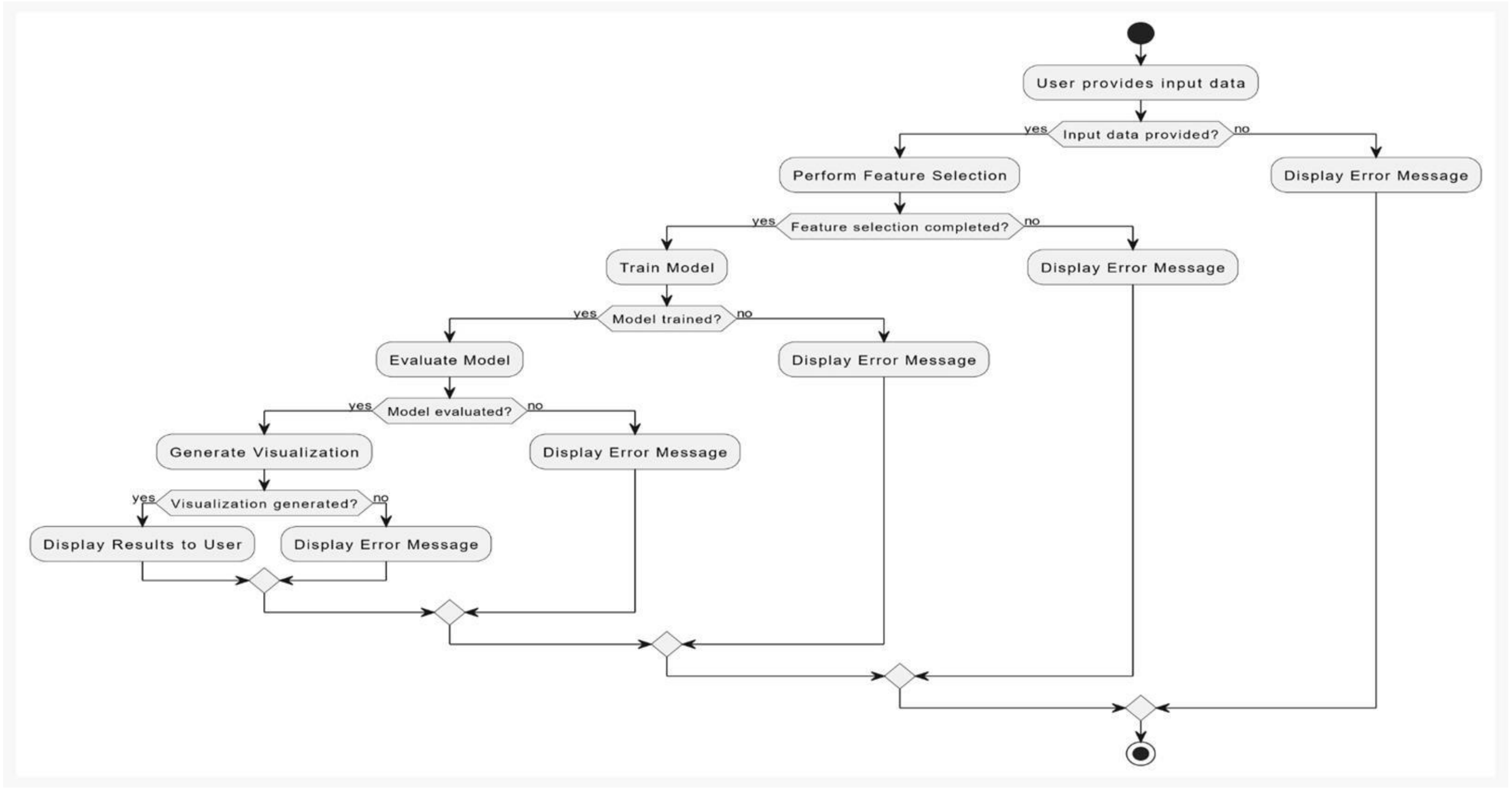
USE CASE DIAGRAM



- The provided use case diagram outlines the primary interactions between users and a Network Intrusion Detection System (NIDS). Two main actors are identified: the System Administrator and the Intrusion Detection System (IDS). The System Administrator is responsible for operational oversight and ensures the security of the network. This actor interacts with the NIDS through two main use cases: "Monitor Network Traffic" and "Analyze Traffic Patterns." In the first case, the administrator continuously observes incoming and outgoing data across the network to detect any abnormal behavior. The second case, "Analyze Traffic Patterns," involves examining these traffic logs and flow data to identify patterns that might suggest potential security breaches or performance issues. These actions play a critical role in establishing a baseline of normal behavior, which becomes essential when detecting anomalies. The use case diagram shows that the administrator's involvement is proactive, emphasizing the human role in both routine surveillance and deeper analysis, which ultimately supports the automated components of the system.

- The Intrusion Detection System, on the other hand, operates as an intelligent actor within the architecture. It performs automated functions that are crucial for real-time threat identification and response. Its first use case, "Detect Intrusions," signifies the core functionality of the system where it applies rules, algorithms, or machine learning models to identify malicious activities. These intrusions could include unauthorized access attempts, denial-of-service attacks, or the transmission of malicious payloads. Once an intrusion is detected, the IDS triggers the "Alert Administrator" use case. This use case is vital for ensuring that human operators are promptly notified about potential threats, allowing them to initiate appropriate incident response actions. The use case diagram illustrates a seamless collaboration between human and machine—while the system performs continuous monitoring and intrusion detection automatically, it still relies on human oversight to interpret alerts and take remedial measures. Overall, this diagram emphasizes the dual approach of automation and human analysis in securing a network environment, aligning with best practices in cybersecurity for enhanced threat management.

ACTIVITY DIAGRAM

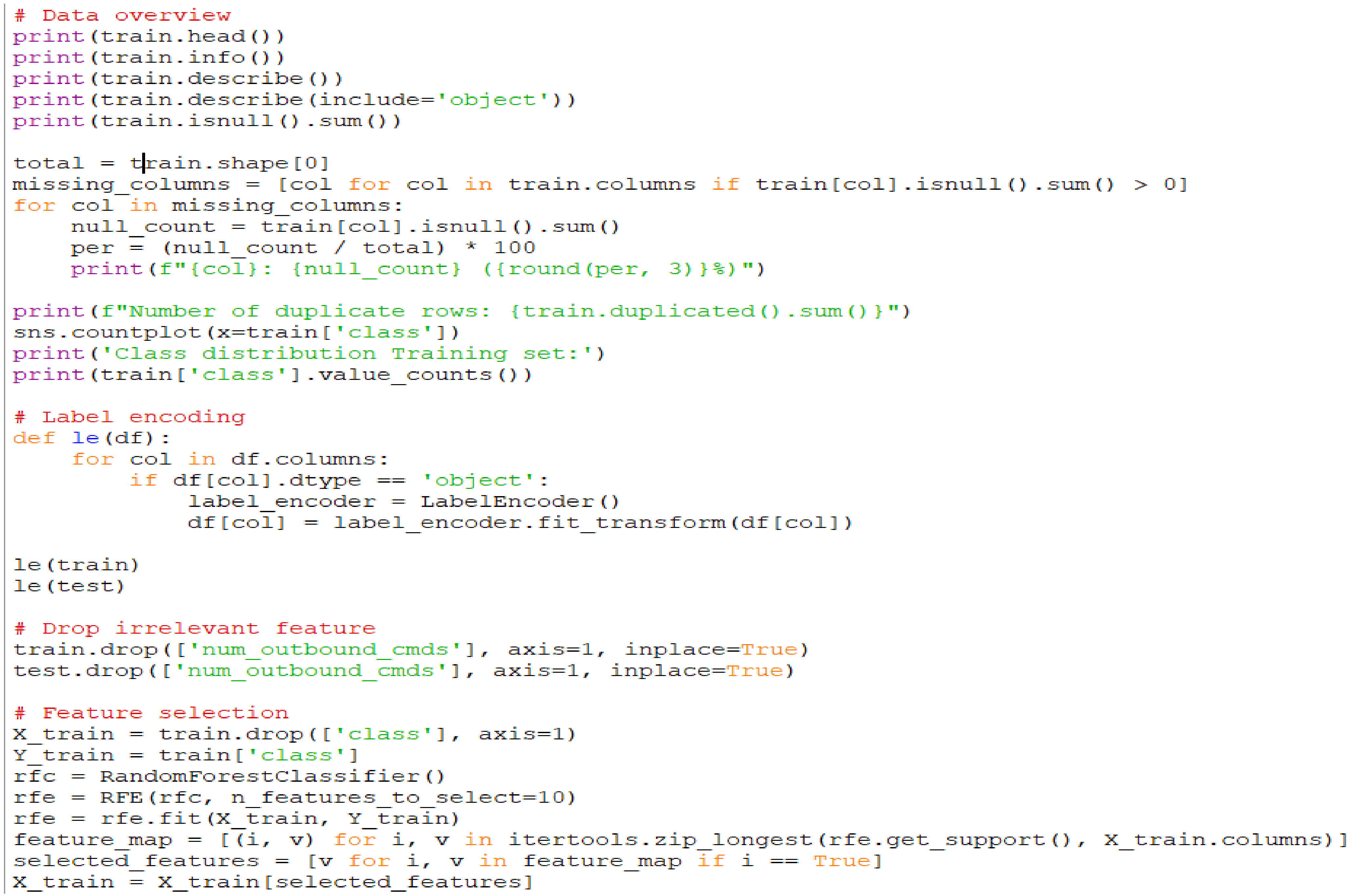


- The activity diagram represents the complete workflow of a machine learning pipeline from the initial step of input data collection to the final stage of displaying results to the user. The process begins when a user provides input data, which is then checked to ensure it is valid. If the input is missing or incorrect, an error message is displayed. If the input is valid, the system proceeds to perform feature selection, an essential step in which relevant data features are selected to train the model more efficiently. If feature selection is successful, the process continues to the model training stage. If not, an error message interrupts the pipeline. At the training stage, the model is built based on the provided and filtered data. If the training fails, another error message is shown. If successful, the pipeline continues to evaluate the trained model using testing data or evaluation metrics. The evaluation ensures the model performs well and is accurate. In case the model evaluation fails, an error message is displayed to notify the user of the failure.

- If the model is successfully evaluated, the next step is to generate a visualization of the results. Visualization is crucial for understanding the model's performance and communicating insights clearly. If visualization is not successfully generated, the system shows an error message. However, if it succeeds, the system displays the final results to the user in a comprehensible visual format. Throughout the workflow, decision diamonds are used to evaluate whether each task was completed successfully or not (marked by "yes" or "no" paths). Any "no" leads to an error message, ensuring that users are informed at every stage where a problem occurs. The flowchart uses standardized symbols: ovals for start/end points, rectangles for actions or processes (like training or generating visualization), and diamonds for decisions (e.g., "Model trained?"). This structure ensures clarity, logic, and ease of debugging. The overall layout emphasizes a linear but conditional flow, which is typical in machine learning pipelines where each step depends on the success of the previous one. It highlights the importance of error handling and proper verification at each phase, ensuring the pipeline is robust and user-friendly.

SOURCE CODE

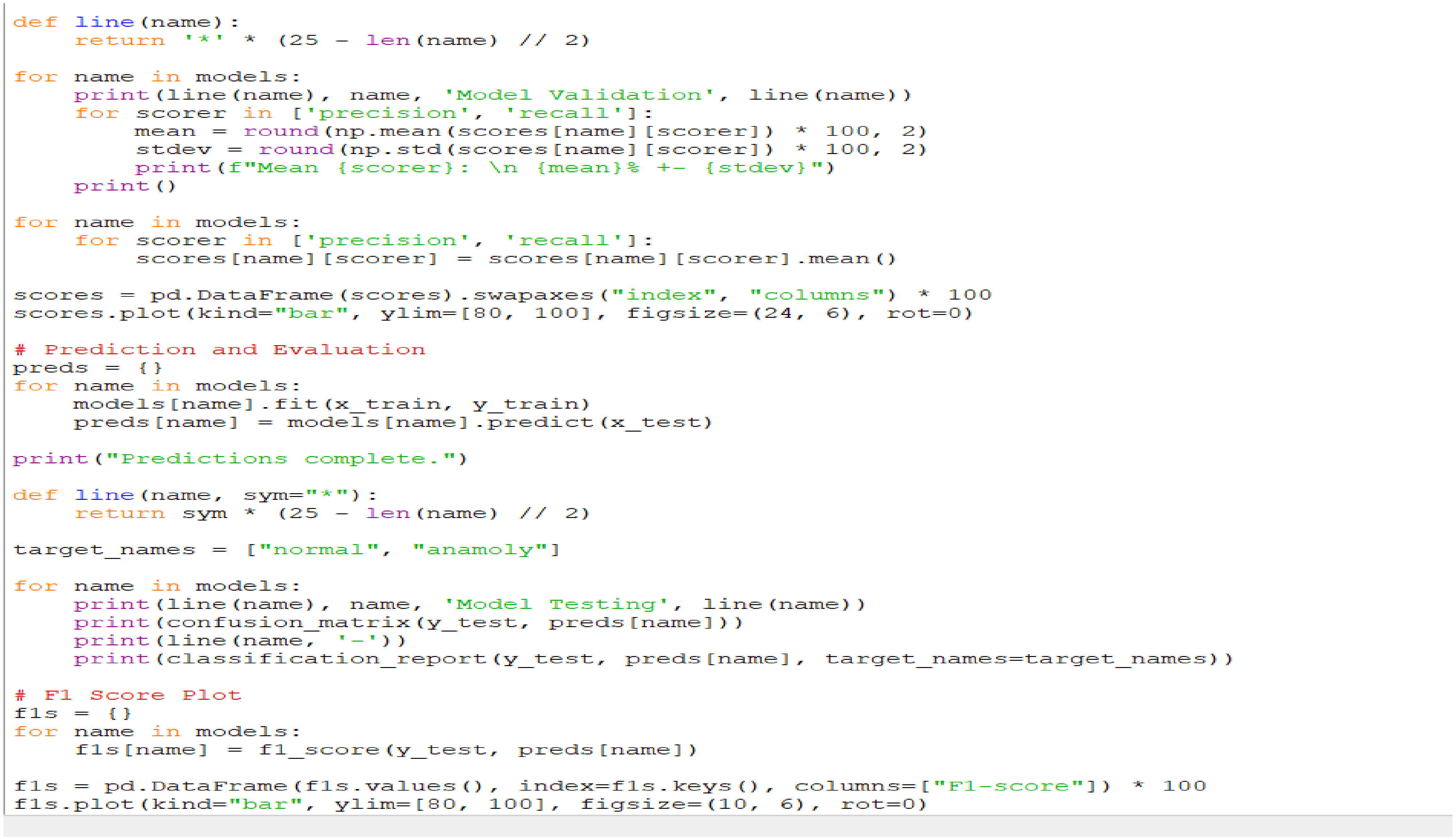
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from pandas.api.types import is_numeric_dtype
import warnings
from sklearn import tree
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier, VotingClassifier, GradientBoostingClassifier
from sklearn.svm import SVC, LinearSVC
from sklearn.naive_bayes import BernoulliNB
from lightgbm import LGBMClassifier
from sklearn.feature_selection import RFE
import itertools
from xgboost import XGBClassifier
from tabulate import tabulate
import optuna
import time
from sklearn.model_selection import cross_val_score
from sklearn.metrics import confusion_matrix, classification_report, f1_score

# Load data
train = pd.read_csv('Train_data.csv')
test = pd.read_csv('Test_data.csv')

# Data overview
print(train.head())
print(train.info())
print(train.describe())
print(train.describe(include='object'))
print(train.isnull().sum())
```



```python
# Data overview
print(train.head())
print(train.info())
print(train.describe())
print(train.describe(include='object'))
print(train.isnull().sum())

total = train.shape[0]
missing_columns = [col for col in train.columns if train[col].isnull().sum() > 0]
for col in missing_columns:
    null_count = train[col].isnull().sum()
    per = (null_count / total) * 100
    print(f"{col}: {null_count} ({round(per, 3)}%)")

print(f"Number of duplicate rows: {train.duplicated().sum()}")
sns.countplot(x=train['class'])
print('Class distribution Training set:')
print(train['class'].value_counts())

# Label encoding
def le(df):
    for col in df.columns:
        if df[col].dtype == 'object':
            label_encoder = LabelEncoder()
            df[col] = label_encoder.fit_transform(df[col])

le(train)
le(test)

# Drop irrelevant feature
train.drop(['num_outbound_cmds'], axis=1, inplace=True)
test.drop(['num_outbound_cmds'], axis=1, inplace=True)

# Feature selection
X_train = train.drop(['class'], axis=1)
Y_train = train['class']
rfc = RandomForestClassifier()
rfe = RFE(rfc, n_features_to_select=10)
rfe = rfe.fit(X_train, Y_train)
feature_map = [(i, v) for i, v in itertools.zip_longest(rfe.get_support(), X_train.columns)]
selected_features = [v for i, v in feature_map if i == True]
X_train = X_train[selected_features]
```

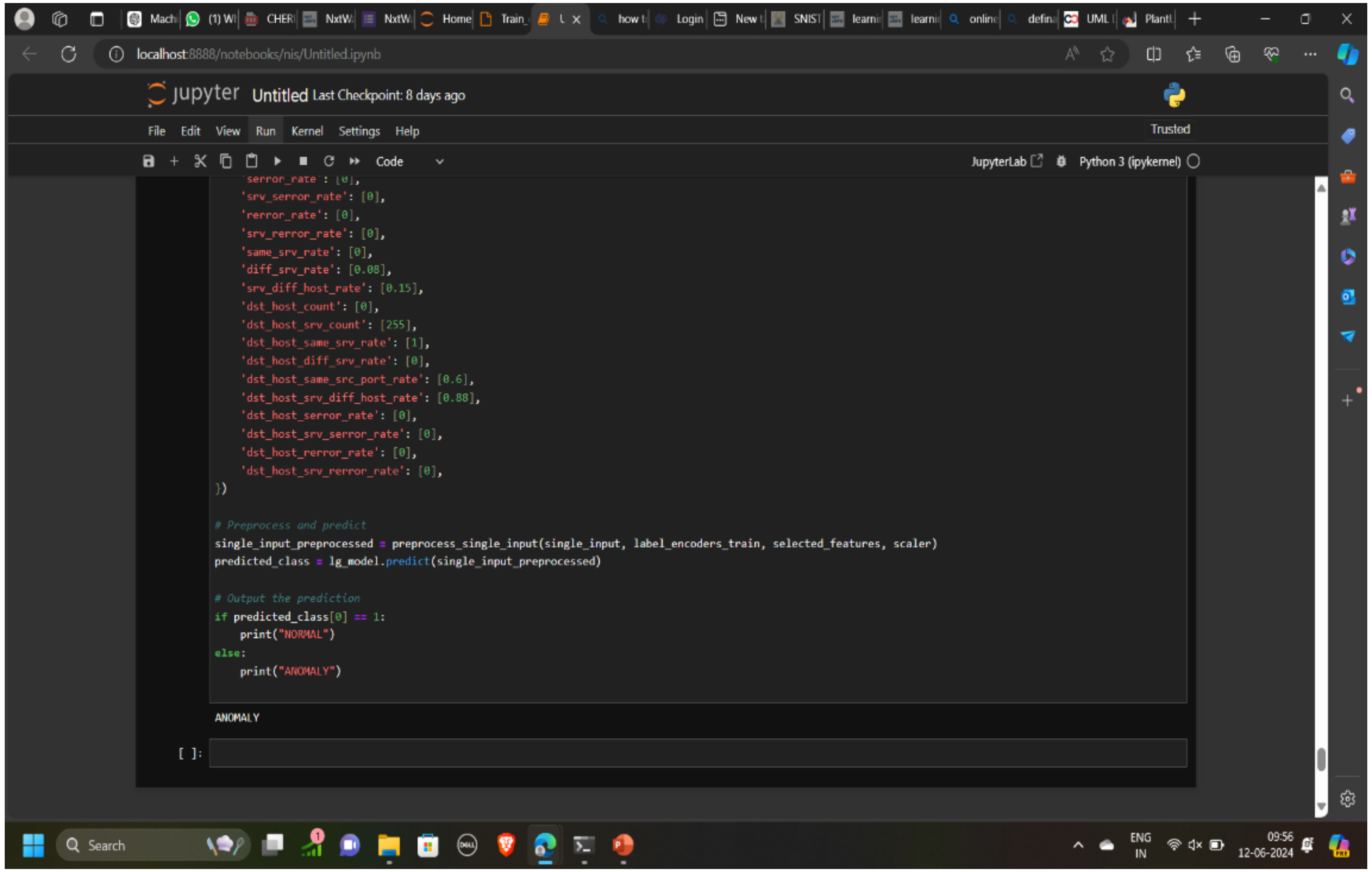
```python
# Scaling
scale = StandardScaler()
X_train = scale.fit_transform(X_train)
test = scale.fit_transform(test)

# Split data
x_train, x_test, y_train, y_test = train_test_split(X_train, Y_train, train_size=0.70, random_state=2)

# Logistic Regression
clfl = LogisticRegression(max_iter=1200000)
start_time = time.time()
clfl.fit(x_train, y_train.values.ravel())
print("Training time: ", time.time() - start_time)

start_time = time.time()
y_test_pred = clfl.predict(x_train)
print("Testing time: ", time.time() - start_time)

lg_model = LogisticRegression(random_state=42)
lg_model.fit(x_train, y_train)
lg_train, lg_test = lg_model.score(x_train, y_train), lg_model.score(x_test, y_test)
print(f"Training Score: {lg_train}")
print(f"Test Score: {lg_test}")

# KNN with Optuna
def objective(trial):
    n_neighbors = trial.suggest_int('KNN_n_neighbors', 2, 16, log=False)
    classifier_obj = KNeighborsClassifier(n_neighbors=n_neighbors)
    classifier_obj.fit(x_train, y_train)
    return classifier_obj.score(x_test, y_test)

optuna.logging.set_verbosity(optuna.logging.WARNING)
study_KNN = optuna.create_study(direction='maximize')
study_KNN.optimize(objective, n_trials=1)
print(study_KNN.best_trial)

KNN_model = KNeighborsClassifier(n_neighbors=study_KNN.best_trial.params['KNN_n_neighbors'])
KNN_model.fit(x_train, y_train)
KNN_train, KNN_test = KNN_model.score(x_train, y_train), KNN_model.score(x_test, y_test)
print(f"Train Score: {KNN_train}")
print(f"Test Score: {KNN_test}")
```

```python
# Decision Tree with Optuna
def objective(trial):
    dt_max_depth = trial.suggest_int('dt_max_depth', 2, 32, log=False)
    dt_max_features = trial.suggest_int('dt_max_features', 2, 10, log=False)
    classifier_obj = DecisionTreeClassifier(max_features=dt_max_features, max_depth=dt_max_depth)
    classifier_obj.fit(x_train, y_train)
    return classifier_obj.score(x_test, y_test)

study_dt = optuna.create_study(direction='maximize')
study_dt.optimize(objective, n_trials=30)
print(study_dt.best_trial)

dt = DecisionTreeClassifier(max_features=study_dt.best_trial.params['dt_max_features'], max_depth=study_dt.best_trial.params['dt_max_depth'])
dt.fit(x_train, y_train)
dt_train, dt_test = dt.score(x_train, y_train), dt.score(x_test, y_test)
print(f"Train Score: {dt_train}")
print(f"Test Score: {dt_test}")

# Model Comparison Table
data = [["KNN", KNN_train, KNN_test],
        ["Logistic Regression", lg_train, lg_test],
        ["Decision Tree", dt_train, dt_test]]
col_names = ["Model", "Train Score", "Test Score"]
print(tabulate(data, headers=col_names, tablefmt="fancy_grid"))

# Cross Validation
models = {
    'KNeighborsClassifier': KNN_model,
    'LogisticRegression': lg_model,
    'DecisionTreeClassifier': dt
}

scores = {}
for name in models:
    scores[name] = {}
    for scorer in ['precision', 'recall']:
        scores[name][scorer] = cross_val_score(models[name], x_train, y_train, cv=10, scoring=scorer)
```



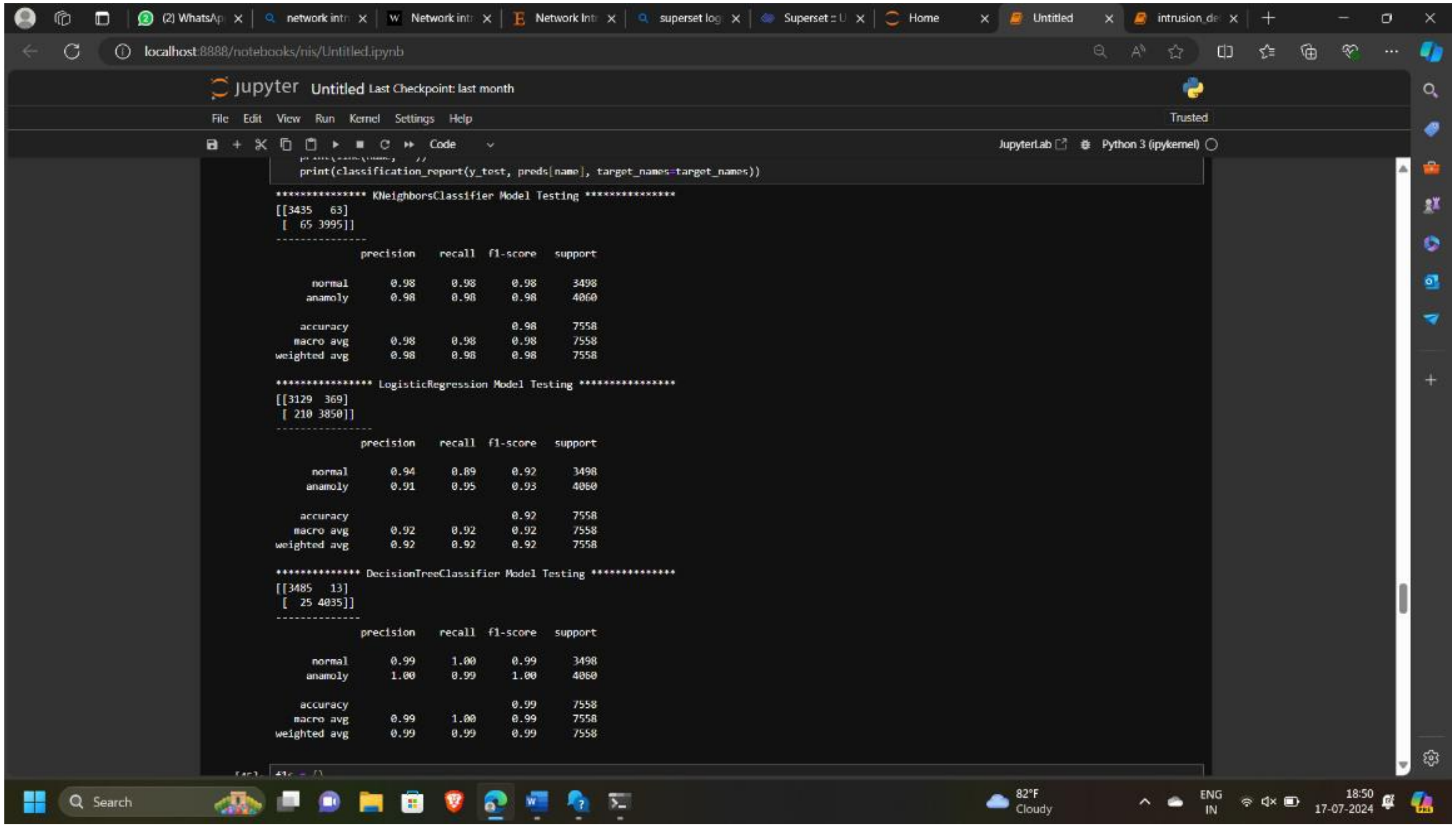
```python
def line(name):
    return '*' * (25 - len(name) // 2)

for name in models:
    print(line(name), name, 'Model Validation', line(name))
    for scorer in ['precision', 'recall']:
        mean = round(np.mean(scores[name][scorer]) * 100, 2)
        stdev = round(np.std(scores[name][scorer]) * 100, 2)
        print(f"Mean {scorer}: \n {mean}% +- {stdev}")
    print()

for name in models:
    for scorer in ['precision', 'recall']:
        scores[name][scorer] = scores[name][scorer].mean()

scores = pd.DataFrame(scores).swapaxes("index", "columns") * 100
scores.plot(kind="bar", ylim=[80, 100], figsize=(24, 6), rot=0)

# Prediction and Evaluation
preds = {}
for name in models:
    models[name].fit(x_train, y_train)
    preds[name] = models[name].predict(x_test)

print("Predictions complete.")

def line(name, sym="*"):
    return sym * (25 - len(name) // 2)

target_names = ["normal", "anamoly"]

for name in models:
    print(line(name), name, 'Model Testing', line(name))
    print(confusion_matrix(y_test, preds[name]))
    print(line(name, '-'))
    print(classification_report(y_test, preds[name], target_names=target_names))

# F1 Score Plot
f1s = {}
for name in models:
    f1s[name] = f1_score(y_test, preds[name])

f1s = pd.DataFrame(f1s.values(), index=f1s.keys(), columns=["F1-score"]) * 100
f1s.plot(kind="bar", ylim=[80, 100], figsize=(10, 6), rot=0)
```

OUTPUT SCREENSHOTS



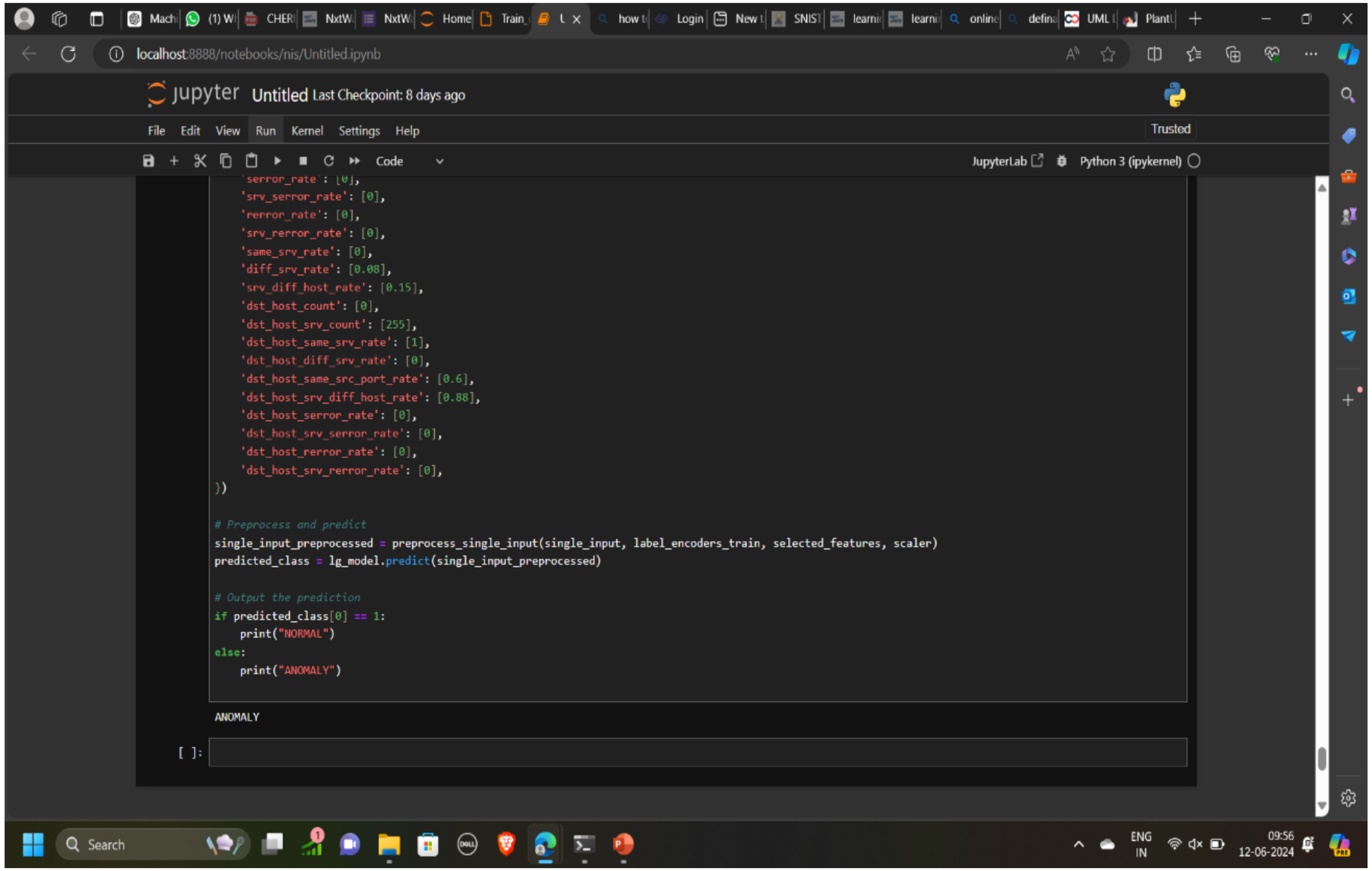
```
    'serror_rate': [0],
    'srv_serror_rate': [0],
    'rerror_rate': [0],
    'srv_rerror_rate': [0],
    'same_srv_rate': [0],
    'diff_srv_rate': [0.08],
    'srv_diff_host_rate': [0.15],
    'dst_host_count': [0],
    'dst_host_srv_count': [255],
    'dst_host_same_srv_rate': [1],
    'dst_host_diff_srv_rate': [0],
    'dst_host_same_src_port_rate': [0.6],
    'dst_host_srv_diff_host_rate': [0.88],
    'dst_host_serror_rate': [0],
    'dst_host_srv_serror_rate': [0],
    'dst_host_rerror_rate': [0],
    'dst_host_srv_rerror_rate': [0],
})

# Preprocess and predict
single_input_preprocessed = preprocess_single_input(single_input, label_encoders_train, selected_features, scaler)
predicted_class = lg_model.predict(single_input_preprocessed)

# Output the prediction
if predicted_class[0] == 1:
    print("NORMAL")
else:
    print("ANOMALY")
```

```
ANOMALY
```



```
    print(classification_report(y_test, preds[name], target_names=target_names))
```

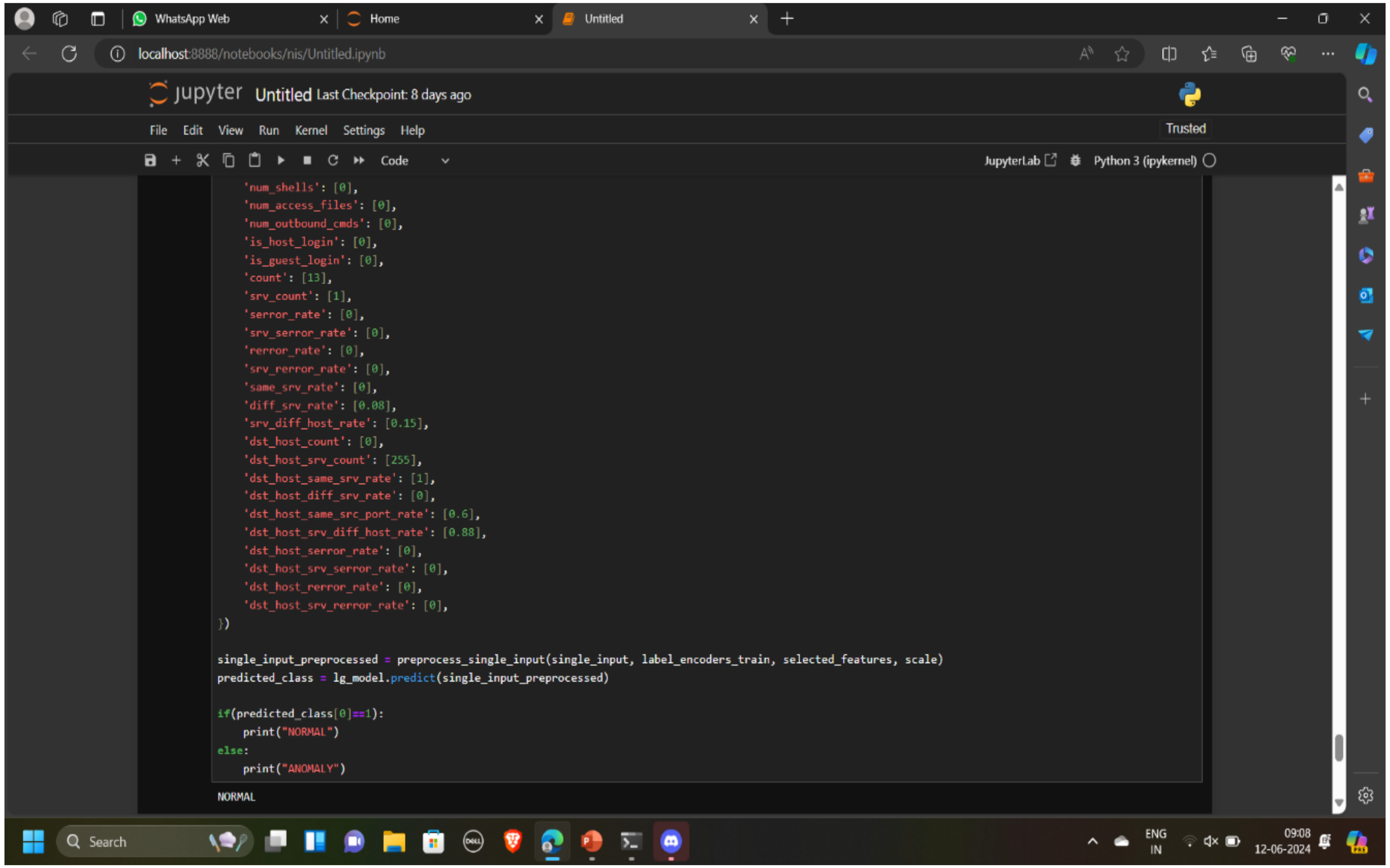
```
*************** KNeighborsClassifier Model Testing ***************
[[3435   63]
 [  65 3995]]
---------------
              precision    recall  f1-score   support

      normal       0.98      0.98      0.98      3498
     anamoly       0.98      0.98      0.98      4060

    accuracy                           0.98      7558
   macro avg       0.98      0.98      0.98      7558
weighted avg       0.98      0.98      0.98      7558

**************** LogisticRegression Model Testing ****************
[[3129  369]
 [ 210 3850]]
----------------
              precision    recall  f1-score   support

      normal       0.94      0.89      0.92      3498
     anamoly       0.91      0.95      0.93      4060

    accuracy                           0.92      7558
   macro avg       0.92      0.92      0.92      7558
weighted avg       0.92      0.92      0.92      7558

************** DecisionTreeClassifier Model Testing **************
[[3485   13]
 [  25 4035]]
--------------
              precision    recall  f1-score   support

      normal       0.99      1.00      0.99      3498
     anamoly       1.00      0.99      1.00      4060

    accuracy                           0.99      7558
   macro avg       0.99      1.00      0.99      7558
weighted avg       0.99      0.99      0.99      7558
```


localhost:8888/notebooks/nis/Untitled.ipynb
jupyter Untitled Last Checkpoint: 8 days ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)

```
    'serror_rate': [0],
    'srv_serror_rate': [0],
    'rerror_rate': [0],
    'srv_rerror_rate': [0],
    'same_srv_rate': [0],
    'diff_srv_rate': [0.08],
    'srv_diff_host_rate': [0.15],
    'dst_host_count': [0],
    'dst_host_srv_count': [255],
    'dst_host_same_srv_rate': [1],
    'dst_host_diff_srv_rate': [0],
    'dst_host_same_src_port_rate': [0.6],
    'dst_host_srv_diff_host_rate': [0.88],
    'dst_host_serror_rate': [0],
    'dst_host_srv_serror_rate': [0],
    'dst_host_rerror_rate': [0],
    'dst_host_srv_rerror_rate': [0],
})

# Preprocess and predict
single_input_preprocessed = preprocess_single_input(single_input, label_encoders_train, selected_features, scaler)
predicted_class = lg_model.predict(single_input_preprocessed)

# Output the prediction
if predicted_class[0] == 1:
    print("NORMAL")
else:
    print("ANOMALY")
```

```
ANOMALY
```

```
[ ]:
```



WhatsApp Web
Home
Untitled
localhost:8888/notebooks/nis/Untitled.ipynb
jupyter Untitled Last Checkpoint: 8 days ago
File Edit View Run Kernel Settings Help
Trusted
Code
JupyterLab
Python 3 (ipykernel)
Search
ENG IN
09:08
12-06-2024

```
    'num_shells': [0],
    'num_access_files': [0],
    'num_outbound_cmds': [0],
    'is_host_login': [0],
    'is_guest_login': [0],
    'count': [13],
    'srv_count': [1],
    'serror_rate': [0],
    'srv_serror_rate': [0],
    'rerror_rate': [0],
    'srv_rerror_rate': [0],
    'same_srv_rate': [0],
    'diff_srv_rate': [0.08],
    'srv_diff_host_rate': [0.15],
    'dst_host_count': [0],
    'dst_host_srv_count': [255],
    'dst_host_same_srv_rate': [1],
    'dst_host_diff_srv_rate': [0],
    'dst_host_same_src_port_rate': [0.6],
    'dst_host_srv_diff_host_rate': [0.88],
    'dst_host_serror_rate': [0],
    'dst_host_srv_serror_rate': [0],
    'dst_host_rerror_rate': [0],
    'dst_host_srv_rerror_rate': [0],
})

single_input_preprocessed = preprocess_single_input(single_input, label_encoders_train, selected_features, scale)
predicted_class = lg_model.predict(single_input_preprocessed)

if(predicted_class[0]==1):
    print("NORMAL")
else:
    print("ANOMALY")
```

```
NORMAL
```

CONCLUSION

In this project, we have developed a powerful and adaptive Network Intrusion Detection System (NIDS) by employing a wide range of machine learning algorithms, including Random Forest, Gradient Boosting, XGBoost, Logistic Regression, Decision Tree, and K-Nearest Neighbors (KNN). The primary objective was to overcome the shortcomings of traditional rule-based intrusion detection systems, which often fail to detect new and sophisticated attacks, by introducing a data-driven approach capable of learning from patterns in network traffic. We began by collecting and analyzing network traffic data, followed by thorough preprocessing to clean and transform the data for optimal model performance. Feature selection techniques such as Recursive Feature Elimination (RFE) were used to identify the most relevant attributes, thereby improving the efficiency and accuracy of our models. Each algorithm was trained and tested on labeled datasets to evaluate its performance in detecting both normal and malicious network activity. Our ensemble approach—combining the strengths of multiple classifiers—helped in capturing complex and nonlinear patterns, leading to higher detection rates and lower false positives. The models were evaluated using key performance metrics like accuracy, precision, recall, F1-score, and confusion matrices, which demonstrated the effectiveness of our system across various attack types. Moreover, cross-validation techniques were applied to ensure the robustness and generalizability of the models. The adaptability of our system makes it suitable for dynamic and real-time intrusion detection environments, while its scalability allows for deployment in both small and large-scale networks. In conclusion, this project not only highlights the potential of machine learning in enhancing cybersecurity but also provides a solid foundation for future improvements through the integration of deep learning techniques, real-time monitoring, and automated threat response mechanisms.

FUTURE SCOPE

- The future scope of this Network Intrusion Detection System (NIDS) project is vast and promising, reflecting the ever-evolving landscape of cybersecurity threats and the continuous advancements in machine learning technologies. One significant direction for future work is the integration of more advanced machine learning and deep learning techniques. By incorporating models such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs), the NIDS can enhance its ability to detect complex and novel attacks. These sophisticated models can provide deeper insights into the temporal patterns and spatial relationships inherent in network traffic data, thereby improving detection accuracy and robustness.
- Another critical area for future development is the enhancement of real-time detection and response capabilities. By leveraging streaming data processing frameworks and integrating the NIDS with real-time data sources, the system can provide immediate alerts and automated responses to mitigate threats as they occur. This requires optimizing the machine learning models for faster inference times and ensuring that the system can scale effectively to handle large volumes of network traffic without compromising performance.
- Adaptive learning and online training represent another promising avenue for future work. Implementing adaptive learning mechanisms will allow the NIDS to update its models based on new data and emerging threats, significantly improving its resilience and effectiveness. Online training techniques can enable the system to learn continuously from new patterns without the need for extensive retraining cycles, thus keeping the detection capabilities up-to-date with minimal manual intervention.
- Incorporating external threat intelligence feeds can also enhance the NIDS's detection capabilities by providing additional context. By correlating internal network activity with external threat data, the system can identify indicators of compromise (IoCs) and respond more effectively to sophisticated attacks. Enhanced feature engineering techniques can further improve the system's performance by creating new features that accurately represent the behavior of applications, protocols, and user activities. Additionally, implementing User Behavior Analytics (UBA) can help detect anomalies based on deviations from typical user behavior, providing another layer of defense against insider threats and sophisticated external attacks.

REFRENCES

- Buczak, A. L., & Guven, E. (2015). A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection. IEEE Communications Surveys & Tutorials, 18(2), 1153-1176. doi:10.1109/COMST.2015.2494502
- Chandola, V., Banerjee, A., & Kumar, V. (2009). Anomaly Detection: A Survey. ACM Computing Surveys, 41(3), 15. doi:10.1145/1541880.1541882.
- Dua, S., & Du, X. (2011). Data Mining and Machine Learning in Cybersecurity. CRC Press. ISBN: 978-1-4398-5052-7
- García-Teodoro, P., Díaz-Verdejo, J., Maciá-Fernández, G., & Vázquez, E. (2009). Anomaly-based network intrusion detection: Techniques, systems and challenges. Computers & Security, 28(1-2), 18-28. doi:10.1016/j.cose.2008.08.003
- Gu, G., Perdisci, R., Zhang, J., & Lee, W. (2008). BotMiner: Clustering Analysis of Network Traffic for Protocol- and Structure-Independent Botnet Detection. Proceedings of the 17th USENIX Security Symposium, 139-154. USENIX Association.
- Javaid, A., Niyaz, Q., Sun, W., & Alam, M. (2016). A Deep Learning Approach for Network Intrusion Detection System. Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS), 21-26. doi:10.4108/eai.3-12-2015.2262516.
- Kim, G., Lee, S., & Kim, S. (2014). A novel hybrid intrusion detection method integrating anomaly detection with misuse detection. Expert Systems with Applications, 41(4), 1690-1700. doi:10.1016/j.eswa.2013.08.066
- Kruegel, C., Mutz, D., Robertson, W., & Vigna, G. (2003). Bayesian Event Classification for Intrusion Detection. Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003), 14-23. doi:10.1109/CSAC.2003.1254303.