
CAPSTONE PROJECT

NETWORK INTRUSION DETECTION USING ML

Presented By:
Srikanth Jammula
Madanapalle Institute of Technology & Science
Computer Science & Engineering

OUTLINE

- Problem Statement
- Proposed System/Solution
- System Development Approach
- Algorithm & Deployment
- Result (Output Image)
- Conclusion
- Future Scope
- References

PROBLEM STATEMENT

The Challenge: Create a robust network intrusion detection system (NIDS) using machine learning. The system should be capable of analyzing network traffic data to identify and classify various types of cyber-attacks (e.g., DoS, Probe, R2L, U2R) and distinguish them from normal network activity. The goal is to build a model that can effectively secure communication networks by providing an early warning of malicious activities

PROPOSED SOLUTION

- To build a robust Network Intrusion Detection System (NIDS) capable of distinguishing normal traffic from multiple classes of attacks (e.g., DoS, Probe, R2L, U2R), we propose a machine-learning-driven pipeline that combines data preprocessing, feature engineering, supervised classification, and operational deployment with real-time alerting. The solution is designed for early warning, scalability, and adaptability to evolving threat patterns.
- **1. Overall Architecture**
- The system architecture consists of the following logical layers:
- **Data Ingestion Layer:** Captures raw network traffic (e.g., packet captures, flow records) or consumes labeled datasets (such as NSL-KDD, CICIDS) for training and testing.
- **Preprocessing & Feature Engineering Layer:** Cleans and transforms raw inputs into structured features suited for ML. This includes normalization, categorical encoding, handling class imbalance, and deriving domain-aware indicators (e.g., connection duration, error rates, protocol flags).
- **Modeling Layer:** Trains one or more machine learning models to classify each traffic instance into benign or attack categories. Multiple algorithms (e.g., Random Forest, XGBoost, SVM, or neural networks) can be evaluated, with potential use of ensemble voting or stacking to improve robustness.
- **Inference & Detection Layer:** Applies the trained model(s) on live or batched traffic to produce predictions. Suspicious activity is flagged, and attack type is classified.
- **Alerting & Dashboard Layer:** Generates real-time alerts for detected intrusions, logs decisions, and provides visual feedback (e.g., confusion matrix summaries, time-series of detected events) to operators.
- **Feedback Loop:** Operator validation and newly labeled data feed back into the system to retrain/refine models, adapting to new attack variants.

2. Data Preprocessing

- Cleaning:** Remove or impute missing values, validate types, and discard irrelevant identifiers.
- Normalization/Scaling:** Continuous numeric features (e.g., packet size, duration) are scaled to prevent dominance in distance-sensitive models.
- Encoding:** Categorical protocol or flag fields are one-hot encoded or otherwise transformed.
- Class Imbalance Handling:** Techniques like class weighting, SMOTE (synthetic oversampling), or undersampling are employed to mitigate skewed attack distribution, especially for rare attack types (e.g., U2R).

3. Feature Engineering

Domain-specific features are derived to amplify signal:

- Connection-level aggregates (e.g., average packet rate, failed connection counts)
- Temporal patterns (e.g., bursts of traffic, rapid successive requests)
- Ratios and flags combining protocol and behavior indicators
- Statistical summaries over sliding windows for streaming detection

4. Model Selection and Training

Multi-class Classification: Models are trained to distinguish among normal traffic and multiple attack types. Candidate algorithms include tree-based methods (Random Forest, XGBoost) for interpretability and gradient boosting performance, as well as SVM or lightweight neural architectures for comparison.

Evaluation Metrics: Accuracy, precision, recall, F1-score per class, and confusion matrices are used to assess performance. Special emphasis is placed on recall for attack classes to minimize missed detections.

Ensembling: Where beneficial, outputs from multiple models are combined (e.g., majority voting or meta-classifier) to reduce variance and improve generalization.

5. Detection & Early Warning

On deployment, the system processes incoming traffic in near real-time, scoring each instance. When an attack is detected:

- A classification label (attack type) and confidence score are generated.
- Threshold-based or rule-enhanced logic triggers alerts for human operators or automated mitigation systems.
- Logs are stored for audit, forensic analysis, and future model refinement.

6. Deployment Considerations

- **Scalability:** The prediction service is containerized and can be horizontally scaled to handle high-throughput environments.
- **Latency:** Models are optimized (e.g., via feature selection or lightweight inference paths) to meet real-time detection requirements.
- **Retraining Pipeline:** Periodic or feedback-triggered retraining is built to adapt to adversarial drift and new attack signatures.

7. Security & Reliability

- Inputs are validated to reduce adversarial manipulation.
- Model decisions can be logged with explainability metadata (e.g., feature importances) to aid trust and troubleshooting.
- Redundancy and health checks ensure the detection service remains operational under load.

8. Extensions and Future Enhancements

Unsupervised Anomaly Detection: Complement supervised models with autoencoders or clustering to catch novel, unseen attack types.

Graph-based Correlation: Correlate events across hosts or sessions to identify coordinated multi-vector attacks.

Integration: Tie into SIEMs or orchestration platforms for automated response (e.g., isolating compromised nodes).

SYSTEM APPROACH

1. System Requirements

To ensure smooth development and execution of the bike rental prediction model, the following **hardware and software requirements** are recommended:

a. Hardware Requirements

Processor: Intel i5/i7 or equivalent (quad-core or higher)

RAM: Minimum 8 GB (16 GB recommended for large datasets)

Storage: At least 20 GB free disk space

GPU (Optional): NVIDIA GPU for accelerated training of complex models

b. Software Requirements

Operating System: Windows 10 / Linux / macOS

Python Version: 3.8 or higher

IDE/Notebook: Jupyter Notebook, PyCharm, or VS Code

Package Manager: pip or conda for installing dependencies

2. Libraries Required to Build the Model

The following Python libraries are required for **data processing, model building, and evaluation**:

Data Handling and Analysis:

pandas → Data loading and manipulation

numpy → Numerical operations and array handling

Data Visualization:

matplotlib → Line plots, histograms, and bar charts

seaborn → Statistical visualizations and heatmaps

Machine Learning and Model Building:

scikit-learn → Linear Regression, Random Forest, Gradient Boosting, model evaluation

xgboost or lightgbm (optional) → Advanced gradient boosting models

Data Preprocessing:

scikit-learn.preprocessing → StandardScaler, MinMaxScaler, OneHotEncoder

scikit-learn.model_selection → Train-test split, cross-validation

Model Evaluation:

scikit-learn.metrics → R^2 score, Mean Absolute Error (MAE), Root Mean Squared Error (RMSE)

ALGORITHM & DEPLOYMENT

■ Algorithm Selection

For the Network Intrusion Detection System (NIDS), a supervised multi-class classification approach is used to distinguish between normal traffic and various attack types (DoS, Probe, R2L, U2R). Tree-based ensemble methods such as XGBoost (and optionally Random Forest as a baseline/comparison) are chosen because they:

- Handle nonlinear relationships and heterogeneous feature types well.
- Are robust to noise and outliers, common in network data.
- Provide feature importance for interpretability.
- Scale reasonably for medium-sized network datasets.

■ Data Input

Input features are derived from network traffic records and include:

- Connection-level metrics: duration, bytes transferred, number of packets.
- Protocol information: protocol type, service, flag statuses.
- Behavioral indicators: failed connection counts, connection rate, unusual port usage.
- Environment/context features: source/destination IP aggregates (if featurized), time-based patterns (e.g., bursts in traffic).
- Labels: Ground truth classes – Normal, DoS, Probe, R2L, U2R.

ALGORITHM & DEPLOYMENT

- **Data Input:**

Input features are derived from network traffic records and include:

- **Connection-level metrics:** duration, bytes transferred, number of packets.
- **Protocol information:** protocol type, service, flag statuses.
- **Behavioral indicators:** failed connection counts, connection rate, unusual port usage.
- **Environment/context features:** source/destination IP aggregates (if featurized), time-based patterns (e.g., bursts in traffic).
- **Labels:** Ground truth classes – Normal, DoS, Probe, R2L, U2R.

- **Training Process**

1. **Preprocessing:** Clean data, encode categorical fields (one-hot or label encoding), scale numeric features as needed. Address class imbalance using techniques like **SMOTE**, class weighting, or stratified sampling, since some attacks (e.g., U2R) are rare.
2. **Feature Engineering:** Derive composite indicators (e.g., error rate = failed connections / total attempts), sliding-window statistics for temporal context.
3. **Model Training:** Train XGBoost with cross-validation to validate generalization; tune hyperparameters via Randomized/Grid search optimizing for recall/F1 on attack classes to reduce false negatives.
4. **Evaluation:** Use confusion matrices, per-class precision/recall/F1, and AUC (where applicable) to assess performance, placing priority on catching attacks (higher recall for malicious classes).

ALGORITHM & DEPLOYMENT

Prediction Process

- Incoming network traffic records are preprocessed identically to training (same feature pipeline).
- The trained model predicts a class label per connection/flow, outputting attack type or “Normal” with confidence scores.
- Thresholding and rule-enhancements can be applied to adjust sensitivity (e.g., flag low-confidence suspicious as “review”).
- Predictions feed into alert generation, tagging events with metadata (timestamp, source/dest, predicted class, confidence)

Deployment

1. Architecture Overview.

The deployed NIDS is structured into modular services:

Data Collector: Captures live traffic (e.g., via packet sniffers, NetFlow/sFlow exporters) or ingests batched logs.

Preprocessing Service: Normalizes and encodes incoming traffic into model-ready feature vectors.

Inference Engine: Hosts the trained ML model (e.g., XGBoost) exposed via a REST/gRPC API for low-latency classification.

Alerting & Dashboard: Receives predictions, logs incidents, triggers alerts (email/SIEM integration), and visualizes metrics.

Retraining Pipeline: Periodically or feedback-driven retraining using newly labeled data to adapt to drift.

2. Technology Stack

- **Model Serving:** Containerized inference service (e.g., Flask/FastAPI or lightweight model server) serving the trained model.
- **Orchestration:** Docker for packaging; optional Kubernetes or serverless platform for scaling (e.g., IBM Cloud Code Engine, if aligning with your earlier cloud usage).
- **Data Pipeline:** Kafka or lightweight message queue for streaming traffic into preprocessing/inference.
- **Storage:** Time-series DB or log store(e.g., Elasticsearch, PostgreSQL) for alerts and historical results.
- **Visualization:** Grafana or custom dashboard for operator insight.
- **Alerting:** Integration with monitoring/SIEM for real-time response.

3. Scalability & Reliability

- Stateless inference service scaled horizontally behind a load balancer to handle traffic bursts.
- Circuit-breaker and retry logic to ensure resilience if downstream systems (e.g., logging) are temporarily unavailable.
- Health checks and auto-restart for containers.

4. Real-time Detection & Feedback Loop

- The system ingests traffic in near real-time; predictions are made with minimal latency.
- Detected anomalies/attacks are reviewed; confirmed labels are fed back to update the training corpus.
- Scheduled or triggered retraining refreshes the model to capture new threat patterns.

5. Security Considerations

- Input validation to mitigate adversarial manipulation of features.
- Secure communication (TLS) between components.
- Audit logging of decisions, including feature importance snapshots for explainability in critical alerts.
- Role-based access control on dashboards and alert management.

6. Deployment Automation

- CI/CD pipeline automates testing, container builds, and rolling updates of the inference model.
- Versioning of models to allow rollback if newly deployed models degrade.

```
[ ] %pip install numpy pandas seaborn matplotlib optuna sklearn xgboost catboost lightgbm > /dev/null 2>&1
```

```
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.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC, LinearSVC
from sklearn.feature_selection import RFE
import itertools
import optuna
```

```
[ ] from xgboost import XGBClassifier
    from tabulate import tabulate
```

```
[ ] train = pd.read_csv('/content/Train_data.csv')
    test = pd.read_csv('/content/Test_data.csv')
```

```
train
```





	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	...	dst_host_srv_count	dst_host_same_srv_rate	dst_host_diff_srv_rate	dst_hos
0	0	tcp	ftp_data	SF	491	0	0	0	0	0	...	25	0.17	0.03	
1	0	udp	other	SF	146	0	0	0	0	0	...	1	0.00	0.60	
2	0	tcp	private	S0	0	0	0	0	0	0	...	26	0.10	0.05	
3	0	tcp	http	SF	232	8153	0	0	0	0	...	255	1.00	0.00	
4	0	tcp	http	SF	199	420	0	0	0	0	...	255	1.00	0.00	
...	
25187	0	tcp	exec	RSTO	0	0	0	0	0	0	...	7	0.03	0.06	
25188	0	tcp	ftp_data	SF	334	0	0	0	0	0	...	39	1.00	0.00	
25189	0	tcp	private	REJ	0	0	0	0	0	0	...	13	0.05	0.07	
25190	0	tcp	nnspp	S0	0	0	0	0	0	0	...	20	0.08	0.06	
25191	0	tcp	finger	S0	0	0	0	0	0	0	...	49	0.19	0.03	

25192 rows × 42 columns

```
[ ] train.info()
```



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 25192 entries, 0 to 25191
Data columns (total 42 columns):
#   Column                Non-Null Count  Dtype
---  ---                ---
0   duration              25192 non-null  int64
1   protocol_type         25192 non-null  object
2   service               25192 non-null  object
3   flag                 25192 non-null  object
4   src_bytes             25192 non-null  int64
5   dst_bytes             25192 non-null  int64
6   land                 25192 non-null  int64
7   wrong_fragment        25192 non-null  int64
```



```
[ ] train.head()
```



	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	...	dst_host_srv_count	dst_host_same_srv_rate	dst_host_diff_srv_rate	dst_host_same...
0	0	tcp	ftp_data	SF	491	0	0	0	0	0	...	25	0.17	0.03	
1	0	udp	other	SF	146	0	0	0	0	0	...	1	0.00	0.60	
2	0	tcp	private	S0	0	0	0	0	0	0	...	26	0.10	0.05	
3	0	tcp	http	SF	232	8153	0	0	0	0	...	255	1.00	0.00	
4	0	tcp	http	SF	199	420	0	0	0	0	...	255	1.00	0.00	

5 rows × 42 columns

```
train.describe()
```



	duration	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	num_failed_logins	logged_in	num_compromised	...	dst_host_count	dst_host_s...
count	25192.000000	2.519200e+04	2.519200e+04	25192.000000	25192.000000	25192.000000	25192.000000	25192.000000	25192.000000	25192.000000	...	25192.000000	2519
mean	305.054104	2.433063e+04	3.491847e+03	0.000079	0.023738	0.00004	0.198039	0.001191	0.394768	0.227850	...	182.532074	11
std	2686.555640	2.410805e+06	8.883072e+04	0.008910	0.260221	0.00630	2.154202	0.045418	0.488811	10.417352	...	98.993895	11
min	0.000000	0.000000e+00	0.000000e+00	0.000000	0.000000	0.00000	0.000000	0.000000	0.000000	0.000000	...	0.000000	
25%	0.000000	0.000000e+00	0.000000e+00	0.000000	0.000000	0.00000	0.000000	0.000000	0.000000	0.000000	...	84.000000	1
50%	0.000000	4.400000e+01	0.000000e+00	0.000000	0.000000	0.00000	0.000000	0.000000	0.000000	0.000000	...	255.000000	6
75%	0.000000	2.790000e+02	5.302500e+02	0.000000	0.000000	0.00000	0.000000	0.000000	1.000000	0.000000	...	255.000000	25
max	42862.000000	3.817091e+08	5.151385e+06	1.000000	3.000000	1.00000	77.000000	4.000000	1.000000	884.000000	...	255.000000	25

8 rows × 38 columns



```
[ ] train.describe(include='object')
```



	protocol_type	service	flag	class
count	25192	25192	25192	25192
unique	3	66	11	2
top	tcp	http	SF	normal
freq	20526	8003	14973	13449

▼ Missing Data

```
[ ] 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)}%)")
```

▼ Duplicates

```
[ ] print(f"Number of duplicate rows: {train.duplicated().sum()}")
```

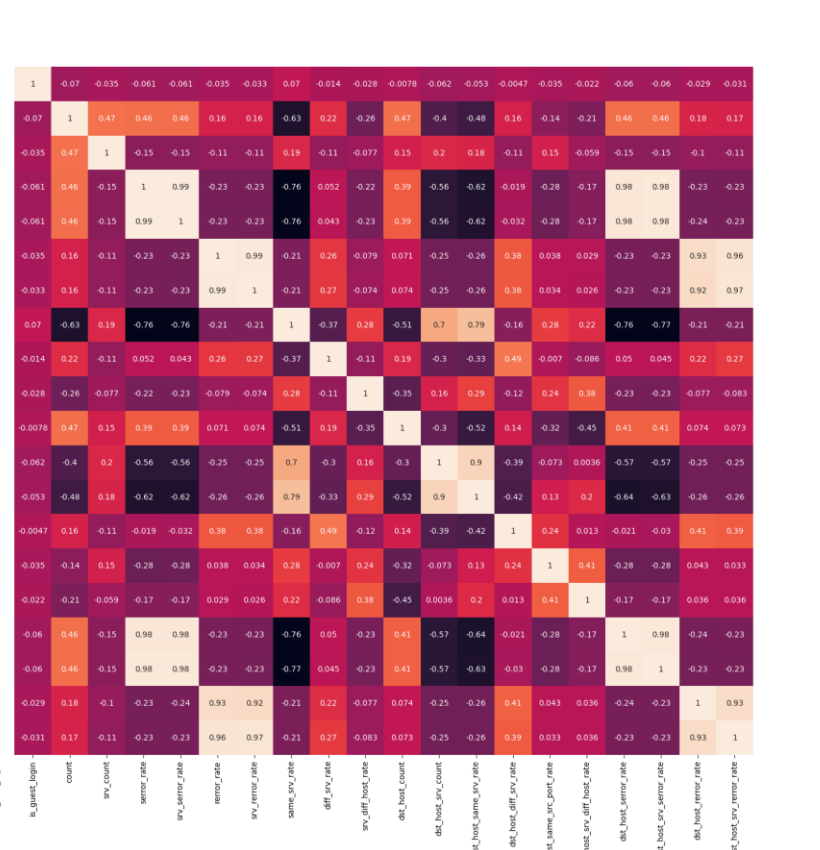
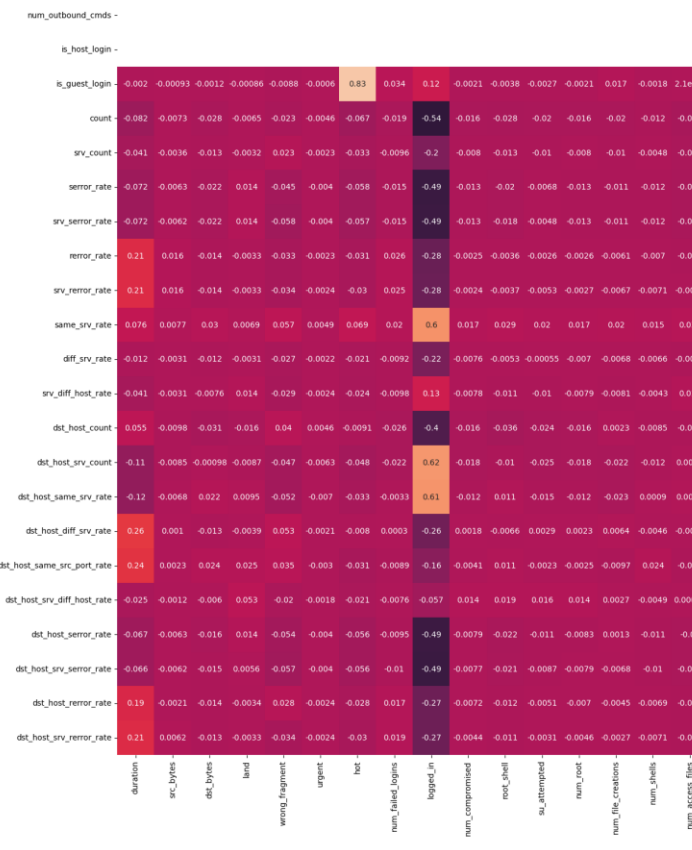


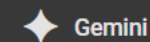
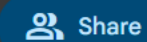
```
Number of duplicate rows: 0
```



```
numeric_df = train.select_dtypes(include='number')



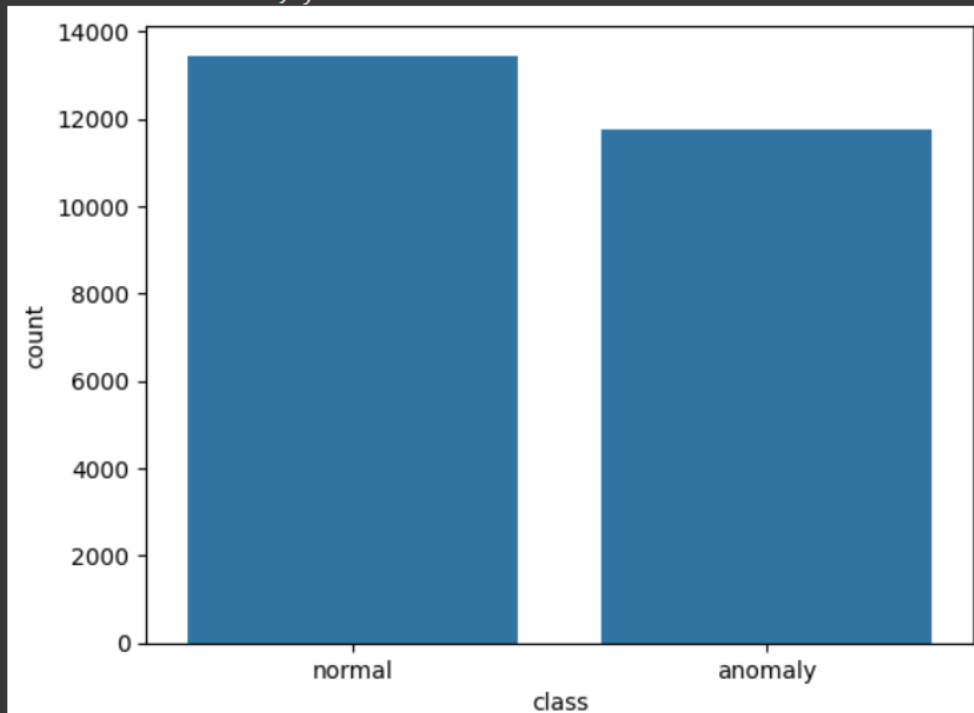
plt.figure(figsize=(40, 30))
sns.heatmap(numeric_df.corr(), annot=True)
```





Q Commands | + Code + Text | ▶ Run all ▼

Connect ▼ ^

 sns.countplot(x=train['class']) <Axes: xlabel='class', ylabel='count'>

▼ 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])
```



```
df[col] = label_encoder.fit_transform(df[col])
```

```
le(train)  
le(test)
```

```
train.drop(['num_outbound_cmds'], axis=1, inplace=True)  
test.drop(['num_outbound_cmds'], axis=1, inplace=True)  
train.head()
```



	duration	protocol_type	service	flag	src_bytes	dst_bytes	land	wrong_fragment	urgent	hot	...	dst_host_srv_count	dst_host_same_srv_rate	dst_host_diff_srv_rate	dst_host_sam...
0	0	1	19	9	491	0	0	0	0	0	...	25	0.17	0.03	
1	0	2	41	9	146	0	0	0	0	0	...	1	0.00	0.60	
2	0	1	46	5	0	0	0	0	0	0	...	26	0.10	0.05	
3	0	1	22	9	232	8153	0	0	0	0	...	255	1.00	0.00	
4	0	1	22	9	199	420	0	0	0	0	...	255	1.00	0.00	

5 rows × 41 columns

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]  
    selected_features
```



```
['protocol_type',  
 'flag',  
 'src_bytes',  
 'dst_bytes',  
 'count',  
 'same_srv_rate',  
 'diff_srv_rate',  
 'dst_host_srv_count',  
 'dst_host_same_srv_rate',  
 'dst_host_same_src_port_rate']
```

```
[ ] X_train = X_train[selected_features]
```

▼ Split and scale data

```
[ ] scale = StandardScaler()  
X_train = scale.fit_transform(X_train)  
test = scale.fit_transform(test)
```

```
[ ] x_train, x_test, y_train, y_test = train_test_split(X_train, Y_train, train_size=0.70, random_state=2)
```

▼ Logistic Regression Model

```
▶ lg_model = LogisticRegression(random_state = 42)  
lg_model.fit(x_train, y_train)
```



▼ LogisticRegression ⓘ ?
LogisticRegression(random_state=42)





```
[ ] 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}")
```

```
Training Score: 0.9417035272768516
Test Score: 0.938872717650172
```

▼ Decision Tree Classifier

```
[ ] 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)
    accuracy = classifier_obj.score(x_test, y_test)
    return accuracy
```

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

```
Training Score: 0.9417035272768516
Test Score: 0.938872717650172

[I 2025-07-31 11:21:39,738] A new study created in memory with name: no-name-4970b442-3e9f-4ba4-a25d-9b368a8e9403
[I 2025-07-31 11:21:39,781] Trial 0 finished with value: 0.9933844932521831 and parameters: {'dt_max_depth': 10, 'dt_max_features': 7}. Best is trial 0 with value: 0.9933844932521831.
[I 2025-07-31 11:21:39,835] Trial 1 finished with value: 0.9951045250066155 and parameters: {'dt_max_depth': 31, 'dt_max_features': 9}. Best is trial 1 with value: 0.9951045250066155.
[I 2025-07-31 11:21:39,873] Trial 2 finished with value: 0.9863720560994972 and parameters: {'dt_max_depth': 6, 'dt_max_features': 10}. Best is trial 1 with value: 0.9951045250066155.
[I 2025-07-31 11:21:39,926] Trial 3 finished with value: 0.9949722148716592 and parameters: {'dt_max_depth': 18, 'dt_max_features': 10}. Best is trial 1 with value: 0.9951045250066155.
[I 2025-07-31 11:21:39,958] Trial 4 finished with value: 0.992590632442445 and parameters: {'dt_max_depth': 18, 'dt_max_features': 5}. Best is trial 1 with value: 0.9951045250066155.
[I 2025-07-31 11:21:39,969] Trial 5 finished with value: 0.9442974331833819 and parameters: {'dt_max_depth': 2, 'dt_max_features': 3}. Best is trial 1 with value: 0.9951045250066155.
[I 2025-07-31 11:21:40,003] Trial 6 finished with value: 0.9937814236570521 and parameters: {'dt_max_depth': 12, 'dt_max_features': 6}. Best is trial 1 with value: 0.9951045250066155.
[I 2025-07-31 11:21:40,030] Trial 7 finished with value: 0.9865043662344536 and parameters: {'dt_max_depth': 6, 'dt_max_features': 6}. Best is trial 1 with value: 0.9951045250066155.
[I 2025-07-31 11:21:40,056] Trial 8 finished with value: 0.9830643027255888 and parameters: {'dt_max_depth': 7, 'dt_max_features': 4}. Best is trial 1 with value: 0.9951045250066155.
[I 2025-07-31 11:21:40,067] Trial 9 finished with value: 0.9442974331833819 and parameters: {'dt_max_depth': 2, 'dt_max_features': 4}. Best is trial 1 with value: 0.9951045250066155.
[I 2025-07-31 11:21:40,122] Trial 10 finished with value: 0.9941783540619211 and parameters: {'dt_max_depth': 31, 'dt_max_features': 8}. Best is trial 1 with value: 0.9951045250066155.
[I 2025-07-31 11:21:40,187] Trial 11 finished with value: 0.9944429743318338 and parameters: {'dt_max_depth': 32, 'dt_max_features': 10}. Best is trial 1 with value: 0.9951045250066155.
```

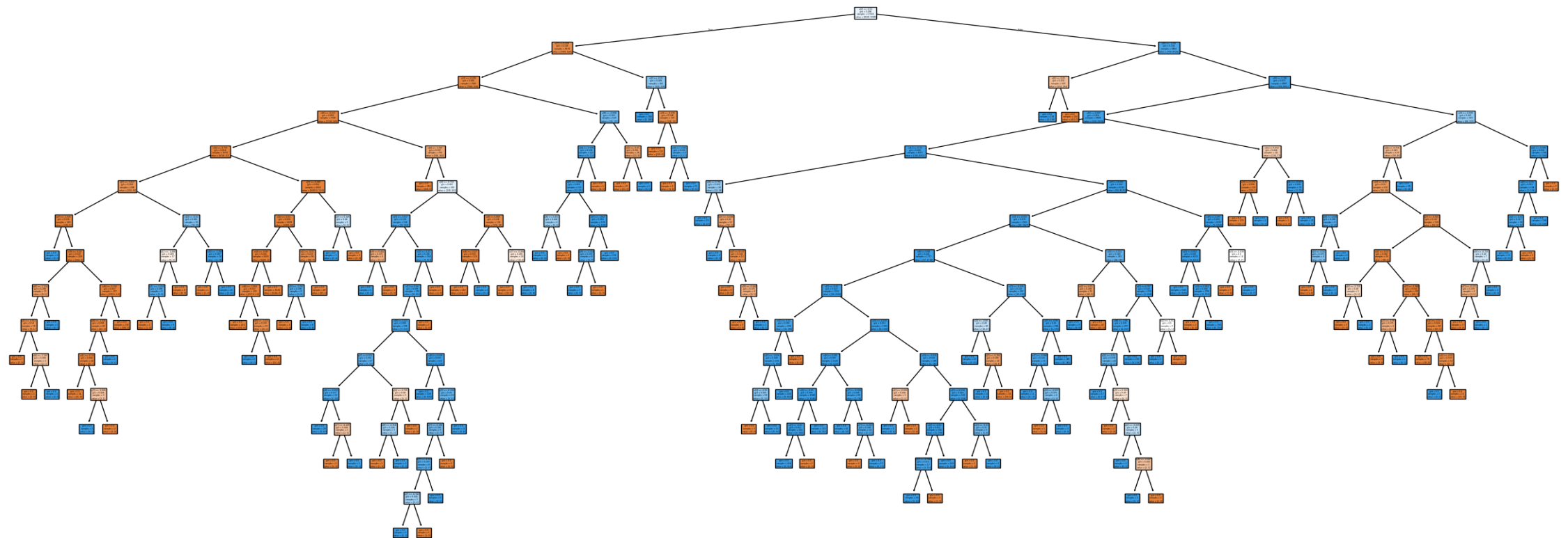
```
[ ] dt = DecisionTreeClassifier(max_features = study_dt.best_trial.params['dt_max_features'], max_depth = study_dt.best_trial.params['dt_max_depth'])
dt.fit(x_train, y_train)

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}")
```

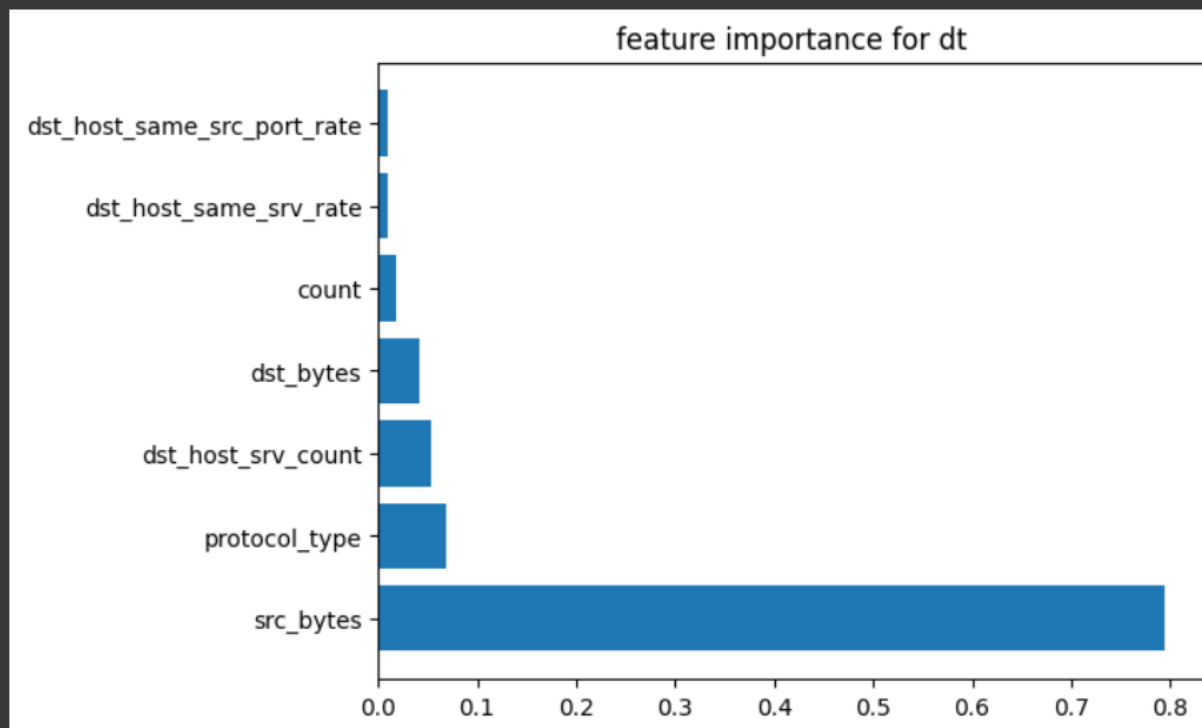
➡ Train Score: 1.0
Test Score: 0.9935168033871394

▶ fig = plt.figure(figsize = (30,12))
tree.plot_tree(dt, filled=True);
plt.show()





```
def f_importance(coef, names, top=-1):  
    imp = coef  
    imp, names = zip(*sorted(list(zip(imp, names))))  
    if top == -1:  
        top = len(names)  
    plt.barh(range(top), imp[::-1][0:top], align='center')  
    plt.yticks(range(top), names[::-1][0:top])  
    plt.title('feature importance for dt')  
    plt.show()  
features_names = selected_features  
f_importance(abs(dt.feature_importances_), features_names, top=7)
```



Random Forest Classifier

```
[ ] def objective(trial):
    rf_max_depth = trial.suggest_int('rf_max_depth', 2, 32, log=False)
    rf_max_features = trial.suggest_int('rf_max_features', 2, 10, log=False)
    rf_n_estimators = trial.suggest_int('rf_n_estimators', 3, 20, log=False)
    classifier_obj = RandomForestClassifier(max_features = rf_max_features, max_depth = rf_max_depth, n_estimators = rf_n_estimators)
    classifier_obj.fit(x_train, y_train)
    accuracy = classifier_obj.score(x_test, y_test)
    return accuracy
```

```
[ ] study_rf = optuna.create_study(direction='maximize')
    study_rf.optimize(objective, n_trials=30)
    print(study_rf.best_trial)
```

→ [I 2025-07-31 11:21:46,769] A new study created in memory with name: no-name-6256b0b0-7480-46b6-9d92-de7b41bcf169
[I 2025-07-31 11:21:47,585] Trial 0 finished with value: 0.9947075946017465 and parameters: {'rf_max_depth': 13, 'rf_max_features': 10, 'rf_n_estimators': 16}. Best is trial 0 with value: 0.9947075946017465
[I 2025-07-31 11:21:47,971] Trial 1 finished with value: 0.9947075946017465 and parameters: {'rf_max_depth': 9, 'rf_max_features': 10, 'rf_n_estimators': 11}. Best is trial 0 with value: 0.9947075946017465

```
[ ] rf = RandomForestClassifier(max_features = study_rf.best_trial.params['rf_max_features'], max_depth = study_rf.best_trial.params['rf_max_depth'], n_estimators = study_rf.best_trial.params['rf_n_estimators'])
    rf.fit(x_train, y_train)

    rf_train, rf_test = rf.score(x_train, y_train), rf.score(x_test, y_test)

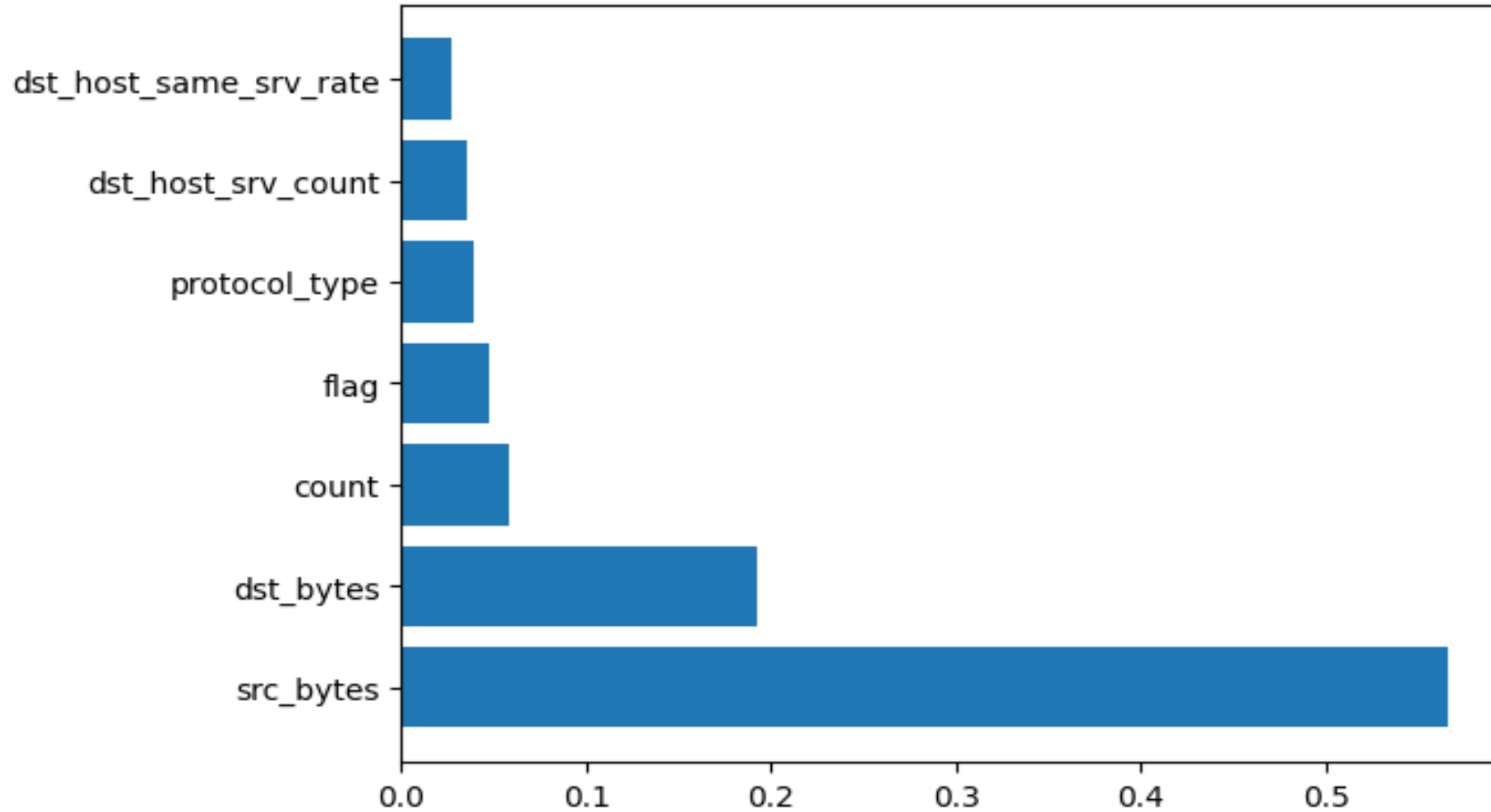
    print(f"Train Score: {rf_train}")
    print(f"Test Score: {rf_test}")
```

→ Train Score: 0.9999432913689463
Test Score: 0.9951045250066155

```
[ ] def f_importance(coef, names, top=-1):
    imp = coef
    imp, names = zip(*sorted(list(zip(imp, names))))
    if top == -1:
        top = len(names)
    plt.barh(range(top), imp[::-1][0:top], align='center')
    plt.yticks(range(top), names[::-1][0:top])
    plt.title('feature importance for dt')
    plt.show()

    features_names = selected_features
    f_importance(abs(rf.feature_importances_), features_names, top=7)
```

feature importance for dt



RESULT

```
[ ] data = ["Logistic Regression", lg_train, lg_test],  
          ["Decision Tree", dt_train, dt_test],  
          ["Random Forest", rf_train, rf_test]  
col_names = ["Model", "Train Score", "Test Score"]  
print(tabulate(data, headers=col_names, tablefmt="fancy_grid"))
```



Model	Train Score	Test Score
Logistic Regression	0.941704	0.938873
Decision Tree	1	0.993517
Random Forest	0.999943	0.995105

CONCLUSION

- The proposed Network Intrusion Detection System leverages machine learning to provide an adaptive, multi-class solution for identifying malicious network activity, including DoS, Probe, R2L, and U2R attacks. By combining thorough preprocessing, feature engineering, and supervised modeling (with XGBoost as the primary classifier), the system effectively discriminates between normal and anomalous traffic, offering actionable early warnings. The modular pipeline—from data acquisition through real-time inference and alert generation—ensures scalability, low-latency detection, and continuous improvement via a feedback loop that incorporates newly labeled events. Deployment considerations such as containerized serving, automated retraining, and integration with monitoring/alerting infrastructure enhance operational robustness and maintainability.
- Key challenges include class imbalance among attack types, potential concept drift as adversaries evolve, and ensuring resilience against adversarial inputs. These were mitigated through techniques like imbalance handling, cross-validation, and careful input validation. Future enhancements include incorporating unsupervised anomaly detection to surface novel threats, enriching context with correlation across sessions, and integrating with broader security orchestration systems for automated response. Overall, the system delivers a practical, extensible foundation for strengthening network security through intelligent, data-driven intrusion detection.

FUTURE SCOPE

- The future scope of this Network Intrusion Detection System is broad and impactful. A key direction is incorporating **unsupervised and semi-supervised learning** (e.g., autoencoders, clustering, and anomaly scoring) to detect novel or zero-day attacks that lack labeled examples. **Deep learning architectures** such as LSTM or Transformer-based sequence models can capture temporal dependencies in traffic for more nuanced detection, especially in encrypted or stealthy flows. **Graph-based correlation** of events across hosts and sessions would support identification of coordinated, multi-stage attacks. Integrating **threat intelligence feeds** and **user behavior analytics** can enrich context, enabling adaptive risk scoring and reducing false positives.
- Deployment can evolve toward **edge and federated learning**, allowing distributed inference while preserving privacy and minimizing central bandwidth. Enhancing **explainability** (e.g., SHAP values or surrogate models) will build operator trust and accelerate incident triage. **Adversarial robustness** training and input sanitization can harden the system against evasion attempts. Automation layers—such as closed-loop **response orchestration** with SOAR/SIEM integration—could enable real-time mitigation. Finally, implementing **active learning** and continuous feedback from analysts will keep models current amid evolving threat landscapes, making the system increasingly self-improving and resilient.

REFERENCES

- <https://www.kaggle.com/datasets/sampadab17/network-intrusion-detection>

IBM CERTIFICATIONS

In recognition of the commitment to achieve
professional excellence



Srikanth Jammula

Has successfully satisfied the requirements for:

Getting Started with Artificial Intelligence



Issued on: Jul 15, 2025
Issued by: IBM SkillsBuild

Verify: <https://www.credly.com/badges/967e44d3-0be6-49e3-b5c4-37b026c8f689>



IBM CERTIFICATIONS

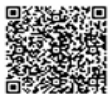
In recognition of the commitment to achieve professional excellence



Srikanth Jammula

Has successfully satisfied the requirements for:

Journey to Cloud: Envisioning Your Solution



Issued on: Jul 31, 2025
Issued by: IBM SkillsBuild

Verify: <https://www.credly.com/badges/192f3da9-751c-4193-8f9f-d309e4c7ea91>



IBM CERTIFICATIONS

IBM **SkillsBuild**

Completion Certificate



This certificate is presented to

Srikanth Jammula

for the completion of

**Lab: Retrieval Augmented Generation with
LangChain**

(ALM-COURSE_3824998)

According to the Adobe Learning Manager system of record

Completion date: 31 Jul 2025 (GMT)

Learning hours: 20 mins



THANK YOU