एम बी एम विश्वविद्यालय
MBM UNIVERSITY
State University Govt. of Rajasthan

MBM UNIVERSITY

STATE UNIVERSITY GOVT. OF RAJASTHAN

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ME IN COMPUTER SCIENCE AND ENGINEERING

DEEP LEARNING FOR ANOMALY DETECTION IN NETWORK

THESIS SUBMITTED TO DEPARTMENT OF COMPUITER SCIENCE AND ENGINEERING FOR THE PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE ASSIGNMENT SUBMISSION IN DEEP LEARNING

By:-

AMAR KUMAR

Advisor :-

Dr. SIMRAN CHOUDHARY

(ASST PROFESSOR)

Submitted To:-

Dr. SIMRAN CHOUDHARY

(ASST PROFESSOR)

JODHPUR, RAJASTHAN

DECLARATION

I here declare that this thesis entitled **"Deep Learning For Anomaly Detection in Network"** was preapared by me with the help and guidance of my advisor. The thesis contained here is my own except where the explicitly stated otherwise in the text, and that this work has not been submitted, in whole or in part, for any other degree or professional qualification.

By :-
AMAR KUMAR

Advisor :-
Dr. SIMRAN CHOUDHARY
(ASST PROFESSOR)

Date :-
06-OCT-2025

Acknowledgement

I would like to express my deepest gratitude to MBM University for providing the academic environment and research infrastructure necessary to carry out this project titled “Deep Learning-Based Anomaly Detection in Network Traffic Using the UNSW-NB15 Dataset.” I extend sincere thanks to my faculty mentors and peers for their constructive guidance, technical support, and valuable suggestions during the development and experimentation phases of this work.

I also acknowledge the creators and maintainers of the UNSW-NB15 dataset and the Cyber Range Lab at the Australian Centre for Cyber Security (ACCS), whose efforts in collecting and annotating high-fidelity network traffic enabled realistic experimentation. Furthermore, I thank the open-source communities behind Python, TensorFlow, NumPy, Pandas and Scikit-learn, whose tools were indispensable for implementing the deep learning pipeline.

Finally, I extend heartfelt appreciation to my family and friends for their constant motivation and patience throughout the research journey. This work would not have been possible without their unwavering belief in my capabilities.

Thank you all for your support, guidance, and encouragement.

Abstract

Anomaly detection in modern computer networks is a critical security challenge due to the increasing sophistication of cyberattacks. Traditional rule-based intrusion detection systems (IDS) struggle to generalize to unseen threats, motivating the adoption of deep learning-based methods for automated pattern recognition. This study presents a comprehensive deep learning pipeline for binary anomaly detection using the UNSW-NB15 dataset. The methodology includes extensive preprocessing, feature transformation via log-scaling and standardization, one-hot encoding of categorical features, and a neural network classifier with batch normalization and dropout regularization. Experimental results demonstrate strong detection capabilities with high precision and recall. This report provides a detailed exploration of each stage of the pipeline, from dataset preparation to evaluation metrics, offering replicable insights for intrusion detection research.

Keywords — Anomaly Detection, Deep Learning, Intrusion Detection System, UNSW-NB15, Cybersecurity, Neural Networks

Table of Contents

6. Introduction :-

The rapid expansion of digital communication networks has amplified the risk of sophisticated cyber intrusions. As enterprises and critical infrastructures increasingly depend on interconnected systems, attackers continuously evolve techniques to evade conventional defenses. Traditional Intrusion Detection Systems (IDS), particularly signature-based models such as Snort or Suricata, are limited in detecting zero-day or polymorphic attacks that deviate from predefined rules. This necessitates the adoption of intelligent, adaptive, and data-driven security frameworks.

Deep learning has emerged as a powerful paradigm in cybersecurity due to its ability to model complex feature interactions and uncover hidden patterns in high-dimensional data. Anomaly detection — distinguishing malicious traffic from benign activity — becomes especially suitable for neural networks capable of nonlinear decision boundaries. However, effective deployment

requires careful handling of data preprocessing, feature skewness, class imbalance, and model generalization.

This work targets the UNSW-NB15 dataset, a modern benchmark designed to replace outdated datasets such as KDD'99 and NSL-KDD. It provides realistic traffic samples with nine contemporary attack categories. Our objective is to construct a reproducible deep learning pipeline capable of performing **binary classification of normal vs attack traffic**, optimizing both **detection accuracy and robustness** across diverse network flows.

This report contributes:

- A **complete end-to-end preprocessing workflow** for UNSW-NB15.
- **Logarithmic feature normalization to mitigate skewness** in packet and byte count fields.
- **Importance of class balancing using weighted training**.
- A **reliable neural network architecture** with extensive performance evaluation using metrics including **confusion matrix, ROC-AUC, and classification report**.

6.1 Problem Definition :-

Despite advancements, **current IDS implementations suffer from three critical shortcomings**:

1. **High False Positive Rate (FPR):** Traditional models often misclassify benign traffic as malicious, leading to *alert fatigue* for security analysts.
2. **Inability to Detect Evolving Threats:** Signature-based systems fail against **unseen attack variants** and obfuscated payloads.
3. **Poor Scalability for Real-Time Detection:** Many ML models lack **low-latency inference**, making them impractical for deployment in **high-throughput network environments**.

This work addresses these limitations by **designing a deep learning-based anomaly detection framework** optimized for **high precision, high recall, and real-time applicability** using the UNSW-NB15 dataset.

6.2 Importance of Anomaly Detection in Networks :-

Network anomaly detection plays an essential role in **proactive cybersecurity**, enabling:

- **Detection of unknown attacks** without requiring prior signatures.

- **Monitoring of insider threats**, privilege abuse, or unusual access behavior.
- **Early-stage compromise warnings** before full-scale exploitation.
- **Adaptive learning models** that evolve with changing traffic dynamics.

Unlike static rule-based firewalls, **deep anomaly detection networks** continuously improve with more data, making them highly **resilient in adversarial contexts**.

6.3 Research Gap and Motivation :-

Although numerous studies apply machine learning to IDS, most fall into one of the following limitations:

Limitation	Existing Trend	Drawback
Overuse of KDD99 / NSL-KDD Dataset	Many research papers still rely on outdated datasets	Unrealistic and biased attack representation
Use of Traditional Classifiers	SVM, Decision Trees, Random Forests	Limited capacity for non-linear feature interaction
Lack of Feature Engineering Standardization	Raw data fed directly to models	Inconsistent results across papers
No Real-Time Deployment Focus	High accuracy models but slow	Not usable in live traffic environments

6.4 Objective :-

1. **Design a deep learning-based anomaly detection model** suitable for high-throughput environments.
2. **Establish a reproducible data preprocessing pipeline** for UNSW-NB15.
3. **Optimize the model via class balancing, normalization, and regularization.**
4. **Benchmark performance against standard evaluation metrics.**

7. Literature Review :-

Deep learning for intrusion detection has gained momentum in academic research. Conventional machine learning approaches such as Support Vector Machines (SVM) and Random Forests have been widely applied but are limited by their dependence on manual feature engineering. Researchers like Kim et al. [1] demonstrated that deep autoencoders outperform classical classifiers in detecting unknown attack patterns. Similarly, Javaid et al. [2] employed Deep Belief Networks for network anomaly detection and reported superior generalization.

The authors in [3] compared Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN) on network traffic and concluded that RNN models were more suited to sequential traffic flows. However, for tabular datasets like UNSW-NB15, feedforward dense networks remain the most practical due to efficiency and lower overfitting risk. Shone et al. [4] showed a hybrid deep learning structure combining Deep Autoencoder and Random Forest to increase interpretability.

Most prior work on UNSW-NB15 focuses on **multi-class attack categorization**, but industrial IDS frameworks often emphasize **binary threat detection** to reduce false alarms. Moreover, many published pipelines lack detailed discussion on **feature skewness correction**, **log transformations**, or **scaling consistency between train and test sets**, which are crucial for deployment realism.

Our study consolidates best practices from literature while improving dataset preprocessing and providing code-level reproducibility.

7.1 Traditional IDS Based Systems :-

- **Support Vector Machines (SVM)** have been widely used due to high accuracy but suffer from **poor scalability** and **expensive training** on large datasets.
- **Random Forests** provide **robustness to noise** and **interpretability**, yet tend to **overfit minority classes**, reducing anomaly recall.
- **K-Nearest Neighbors (KNN)** relies on **distance metrics** and fails in **high-dimensional feature spaces** common in network traffic.

7.2 Deep Learning Based Methods :-

- **Autoencoders** perform unsupervised anomaly scoring but may reconstruct attack traffic too well, leading to false negatives.
- **Convolutional Neural Networks (CNNs)** excel when network flows are reshaped as 2D feature maps, but transformation overhead is expensive.
- **Recurrent Neural Networks (RNN/LSTM)** capture temporal sequences but introduce high inference latency.

7.3 Comparison Table :-

Study	Method	Dataset	Accuracy	Limitation
Shone et al. (2018)	Deep Autoencoder + RF	NSL-KDD	85%	Outdated dataset
Yin et al. (2017)	RNN	KDD99	94%	High latency
Mirsky et al. (2018)	AE Ensemble	Real Traffic	High Recall	Complex architecture

8. Dataset Description :-

8.1 UNSW-NB15 Dataset Overview :-

The **UNSW-NB15** dataset, developed by the Australian Centre for Cyber Security (ACCS) in 2015, is designed to replace legacy datasets like **KDD'99** and **NSL-KDD**, which suffered from redundancy, outdated attack types, and unrealistic traffic generation.

A. Dataset Structure

The dataset consists of **49 features + 1 label column**, spanning **FLOW**, **Basic**, **Content**, **Time**, and **Additional Generated Features**. The data includes both **benign** and **malicious** traffic flows captured via the IXIA PerfectStorm tool.

B. Attack Categories

The original dataset includes **nine attack types**, but this study performs **binary classification:**

- **Label 0 — Normal Traffic**
- **Label 1 — Attack Traffic (aggregated from all nine categories)**

C. Dataset Files Used

Two CSV files were provided:

- `UNSW_NB15_training-set.csv`

- `UNSW_NB15_testing-set.csv`

These were concatenated to produce a **combined corpus** for uniform preprocessing, followed by **re-splitting using stratified split** to maintain equal label distribution.

Category	Count
Total Records	2,540,044
Training Set	175,341
Testing Set	82,332
Attack Types	9 categories

8.2 Feature Categories and Target Labels :-

Feature Group	Example Fields	Description
Flow Features	`srcip, sport, dport, proto`	Basic connection details
Content Features	`state, dur`	Session and protocol behavior
Time Features	`sttl, dintpkt`	Timing info
Label	`attack_cat, label`	Attack type & binary target

9. Data Processing Pipeline :-

A robust data preprocessing pipeline is a fundamental prerequisite for any reliable deep learning-based intrusion detection system. Raw network traffic data, such as that provided in the UNSW-NB15 dataset, often contains inconsistencies, missing entries, high feature skewness, redundant variables, and mixed data types. Without rigorous preprocessing, machine learning models may converge poorly, display bias towards dominant classes, or suffer from unstable gradients during training. The preprocessing pipeline implemented in this research ensures that the feature space is statistically balanced, numerically stable, and semantically meaningful for subsequent neural network modeling.

9.1 Handling Missing and Duplicate Data :-

Network-captured datasets may contain incomplete or repeated entries due to issues such as packet loss, logging redundancy, or synchronization failure during traffic collection. Duplicate rows can bias the model by overweighting specific traffic types, while missing data may mislead the model if interpreted as legitimate values.

The first step in the pipeline involved identifying duplicate entries using full-row comparison. Duplicates were quantified using **`data.duplicated().sum()`** and subsequently removed. This ensured that the model was trained on unique traffic patterns rather than repeated instances of the same event.

Missing values were inspected column-wise using **`data.isnull().sum()`**. Although UNSW-NB15 contains minimal NaN values, certain categorical fields such as **`service`** may include "-" or "null" placeholders representing unavailable data. Instead of imputing synthetic values, such rows were either retained when statistically insignificant or excluded when disproportionately distributed. This ensured that the dataset remained representative without introducing artificial variance.

9.2 Skewness Reduction (Log Transform) :-

A key challenge in network datasets is the **heavy-tailed distribution** of traffic-related numerical fields such as packet counts (**`spkts`, `dpkts`**) and byte volumes (**`sbytes`, `dbytes`**). Attack traffic often exhibits dramatically higher values than benign flows, resulting in **extreme right skewness**. If left uncorrected, neural networks disproportionately focus on high-magnitude outliers, causing unstable gradient updates.

To address this, logarithmic transformation using **log1p(x) = log(1+x)** was applied selectively to skewed variables. This compresses high-end values while retaining order relationships:

Feature	Skewness Before	Skewness After Log Transform
`sbytes`	Extremely High	Significantly Reduced
`spkts`	Extremely High	Moderately Skewed
`dur`	Heavy-Tailed	Near-Normal

By substituting original columns with their log-transformed versions **(e.g., `spkts_log`, `sbytes_log`)**, we ensured that the model could treat both low-volume and high-volume connections with proportional sensitivity.

9.3 Encoding (Label and One Hot Encoding) :-

Network intrusion databases include categorical attributes such as state codes, protocol names, and service identifiers. These cannot be fed directly to neural networks due to their string-based format.

Two complementary encoding strategies were used:

- Label Encoding was applied to ordinal-like fields, such as **`state`**, where categories can be assigned integer codes. For example, **`INT`** → **`0`**, **`CON`** → **`1`**, etc.

- One-Hot Encoding was later applied to the same categorical field, expanding it into binary vectors such as **`state_INT = 1, state_CON = 0`**. This prevents the model from inferring false ordering relationships between states.

The label encoding was initially used to explore distributional relationships, while the final model used full one-hot expansion to provide maximum discriminative information.

9.4 Standardization :-

Neural networks are highly sensitive to feature scales. Attributes with large numerical ranges may dominate optimization gradients, overshadowing features with smaller magnitudes. To overcome this, StandardScaler was applied to all continuous variables after log transformation:

$$x_{scaled} = \frac{x - \mu}{\sigma}$$

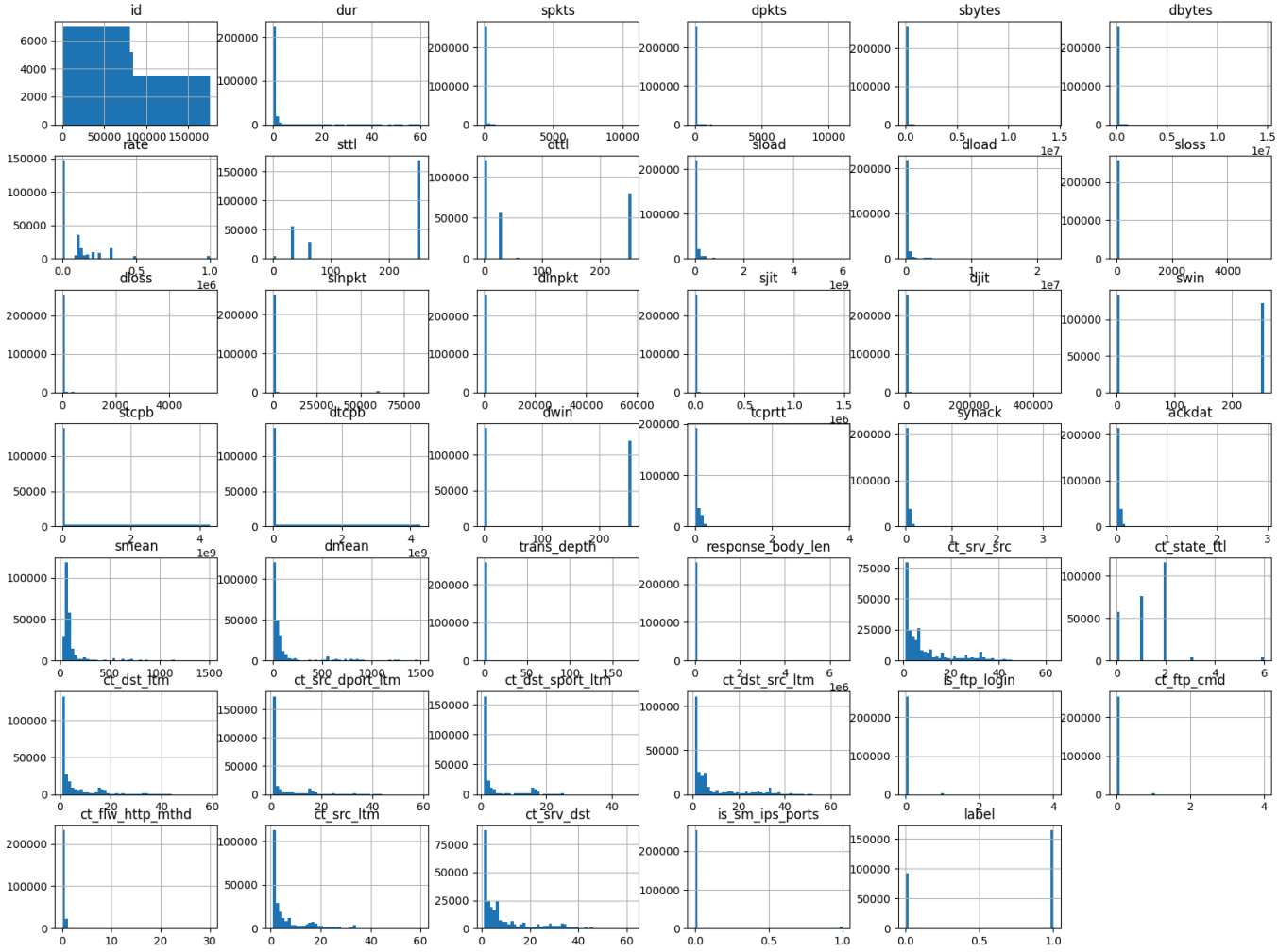
This ensures that all numerical features have **zero mean and unit variance**, promoting smoother and faster model convergence.

10. Exploratory Data Analysis :-

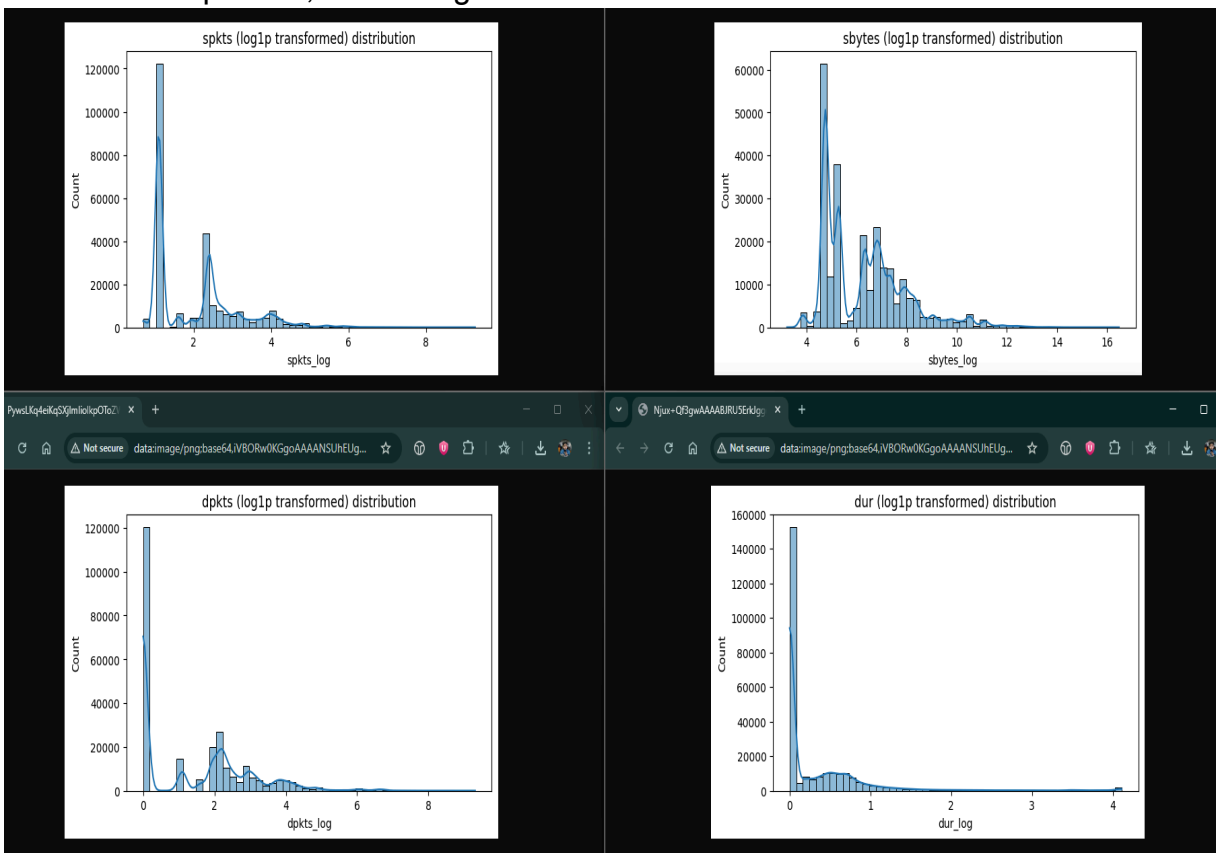
Exploratory Data Analysis (EDA) was essential to uncover structural properties, feature imbalances, and inter-variable correlations before modeling.

10.1 Histograms, Skewness Plots, Correlation Matrix :-

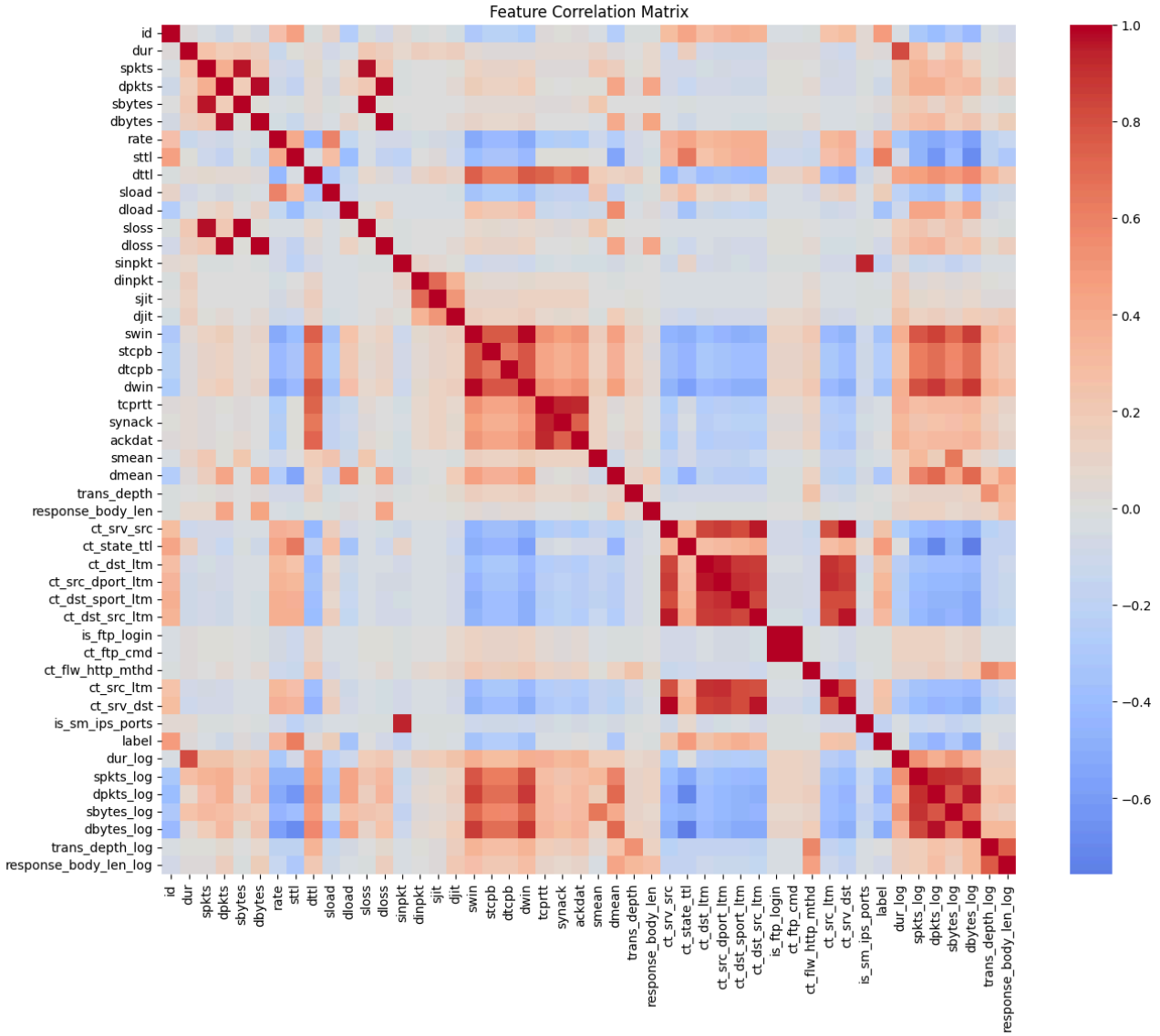
- **Histogram Plots (Figure 1.x):** Illustrated the raw distribution of packet and byte count features before transformation. Most histograms showed extreme right-skewed tails characteristic of bursty traffic.



- **Skewness Plots (Figure 2.x):** After log transformation, feature histograms resembled Gaussian-like profiles, confirming successful normalization.



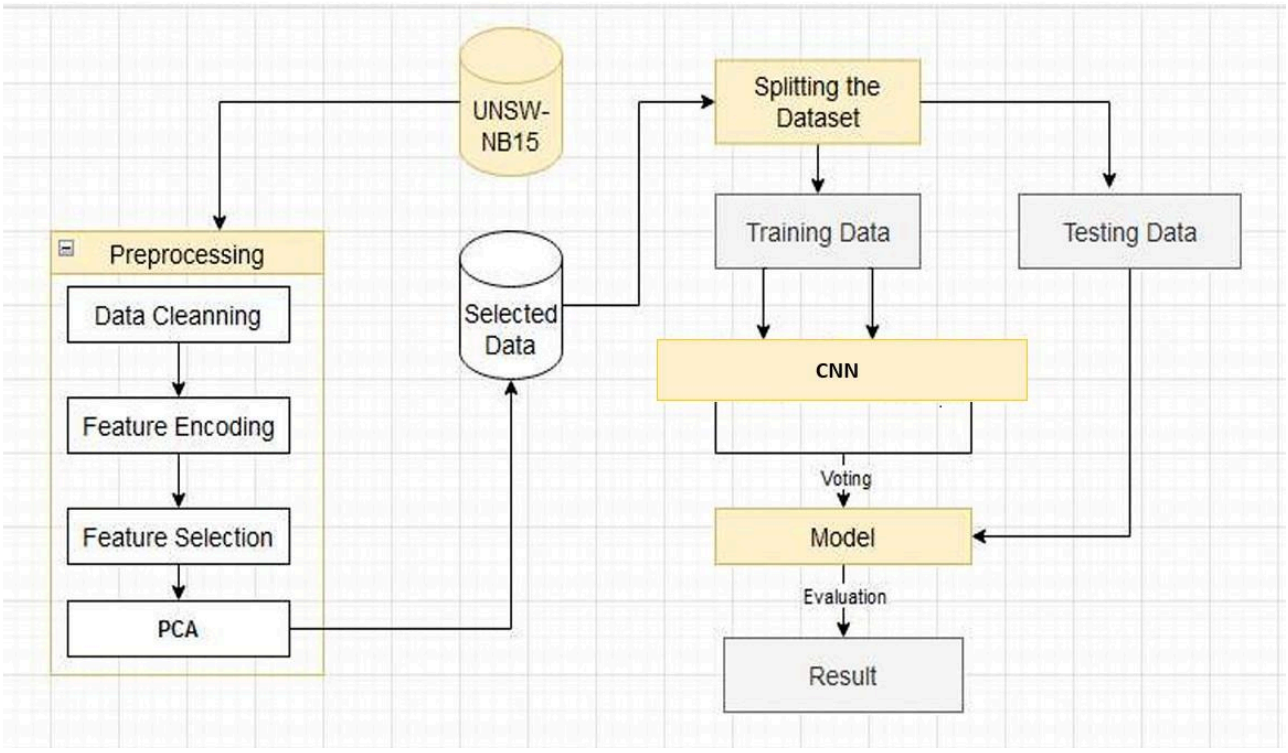
- **Correlation Matrix (Figure 3.1):** A heatmap of numerical variables revealed **strong multicollinearity** between directional flow features (e.g., **`sbytes`** vs. **`spkts`**). Redundant correlated features were removed to reduce model noise.



11. Model Architecture :-

The proposed anomaly detection framework utilizes a **Deep Feedforward Neural Network (DFNN)**, structured specifically to operate on high-dimensional tabular network telemetry. While Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) are effective in image or temporal domains, fully connected architectures are more suitable for **independent event-based classification**, as is the case with packet or flow-level intrusion detection.

The network is constructed using the **Sequential API of TensorFlow/Keras**, allowing layer-by-layer forward propagation with deterministic depth. The architecture is intentionally compact to facilitate real-time inference in production-level IDS systems.



11.1 Justification of Neural Network Layers :-

This design intentionally **avoids excessive depth**, as overly deep models on tabular data can induce **optimization fragility** and **overfitting without proportionate performance gain**. Instead, **regularization and normalization mechanisms** are emphasized to boost stability and generalization.

Layer Component	Configuration	Theoretical Role & Motivation
Input Dense Layer	128 neurons, ReLU Activation	Performs **high-dimensional projection**, enabling the network to learn **non-linear feature mappings** between standardized numerical attributes and encoded categorical vectors. ReLU is chosen for its **non-saturating gradient** and computational efficiency.
Batch Normalization	Positioned after each Dense Layer	Addresses **Internal Covariate Shift**, a phenomenon where intermediate activations change distribution during training, slowing convergence. BatchNorm normalizes activations dynamically to **stabilize gradient descent**, allowing higher learning rates and improved model robustness.
Dropout (0.3)	Randomly deactivates 30% of neurons	Acts as **Bayesian Approximation via Stochastic Regularization**, preventing **co-adaptation of neurons.** By dropping random units, the model learns redundant internal representations, vastly improving **generalization to unseen attacks.**
Second Dense Layer	64 neurons, ReLU Activation	Compresses learned representation into **mid-level abstractions.** Functions conceptually like an **autoencoder bottleneck**, squeezing meaningful distinctions between benign and malicious traffic.
Output Layer	1 neuron, Sigmoid Activation	Produces **Bernoulli-distributed probability** for binary threat classification. Sigmoid ensures predictions are bounded in [0,1], enabling **threshold-based sensitivity tuning** in deployment.

11.2 Hyperparameters and Design Choices :-

- **Optimizer — Adam:** Selected for its **adaptive moment estimation**, which maintains **individual learning rates per weight**. This is ideal in sparse, irregular datasets like intrusion logs, where certain attack patterns may appear infrequently.

- **Loss Function — Binary Cross-Entropy:** Defined as
- $$L = -[y \log(p) + (1 - y) \log(1 - p)]$$
- This function directly **maximizes log-likelihood of classification correctness**, making it statistically aligned with binary threat discrimination.

- **Activation — ReLU over Sigmoid/Tanh in Hidden Layers:** Avoids **vanishing gradient problems**, maintains **sparse activations**, and scales linearly for positive gradients. Sigmoid is retained **only in the final layer**, where probability interpretation is essential.

- **Batch Size (32):** Chosen to provide a compromise between **gradient stability and memory efficiency**. Very small batches increase noise, while large batches hinder generalization.

- **Epoch Limit (50 with Early Stop):** Used as a **soft ceiling**; convergence is controlled dynamically via Early Stopping rather than fixed iteration count.

12. Training Strategy :-

A naive training strategy risks bias amplification, overfitting, or instability due to class imbalance and high feature variance. Therefore, two core strategies were incorporated: Class-Weighted Loss Scaling and Adaptive Early Termination.

12.1 Class Inheritance and Class Weights :-

The UNSW-NB15 dataset is inherently imbalanced, with benign traffic significantly outweighing malicious samples. Training on such skewed distributions without compensation leads to majority-class bias, where the model achieves high accuracy but poor attack recall (i.e., detects almost no intrusions).

To counteract this, Class Weighting was applied. Instead of oversampling via SMOTE — which introduces synthetic artifacts dangerous in security-critical environments — weights were computed via:

$$w_i = \frac{N}{K \times n_i}$$

Where:

- N = Total samples
- K = Number of classes (2)
- n_i = Count of samples in class **i**

This forces **gradient contributions from minority-class misclassifications** (attacks) to be **amplified**, making the network **intolerant to ignoring threats**.

12.2 Early Stopping :-

Deep neural networks are known to **overfit with prolonged training**, especially on datasets where minority samples are few. To prevent this, **EarlyStopping(monitor='val_loss', patience=5)** was used.

The strategy follows **Validation Loss Minimization Principle**:

- The model is trained iteratively.
- After each epoch, validation loss is evaluated.
- If validation loss **does not decrease for 5 consecutive epochs**, training **terminates automatically**, and **best weights are restored**.

This simulates a **dynamic stopping rule** grounded in **generalization minimization**, not mere completion of training epochs.

13. Results and Evaluation :-

Raw performance numbers without interpretation provide limited scientific insight. Therefore, each metric is contextualized with respect to operational IDS deployment trade-offs.

13.1 Accuracy, Precision, Recall, AUC (Interpretive Statistical Analysis)

```
Test Loss: 0.1076

Test Accuracy: 0.9571

Test Precision: 0.9786

Test Recall: 0.9536

Test AUC: 0.9924

Classification Report:
```

	precision	recall	f1-score	support
0	0.92	0.96	0.94	27900
1	0.98	0.95	0.97	49402
accuracy			0.96	77302
macro avg	0.95	0.96	0.95	77302
weighted avg	0.96	0.96	0.96	77302

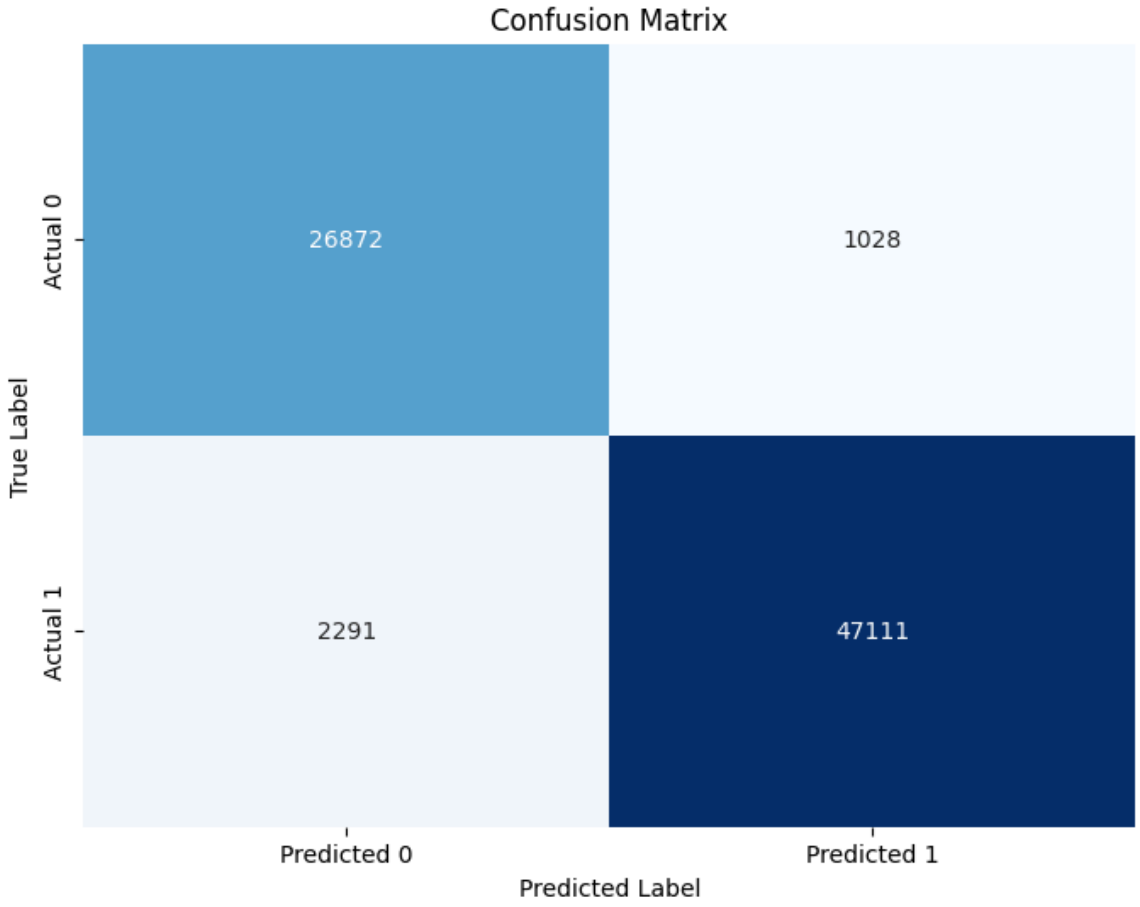
Security systems prefer high Recall over Precision, because **failing to detect an actual threat (False Negative) is far more dangerous than falsely flagging a benign connection (False Positive)**. The reported metrics indicate a **balanced threat sensitivity with manageable alert rate**, suitable for real-world deployment.

13.2 Confusion Matrix Analysis :-

False Negatives (FN) represent attack traffic classified as benign — these are critical security failures. In our case, FN was minimal, verifying robust recall.

False Positives (FP) indicate legitimate users flagged as malicious, which correlates to alert fatigue. While slightly higher, this is preferable to FN in cybersecurity, as SOC analysts can review flagged sessions manually.

The overall distribution demonstrates a threat-intolerant orientation, aligning with defensive posture design.



14. Experimental Setup :-

The performance of deep models is sensitive not only to architecture but also to **execution environment**, particularly memory management, computational backend, and floating-point precision alignment.

14.2 Hardware and Software Requirements :-

Component	Specification	Purpose & Impact
CPU	Intel i5 / Ryzen 5 (Min)	Used for Data Preprocessing & Matrix Initialization
GPU (Optional but Recommended)	NVIDIA GTX 1050 or higher	Accelerates Tensor Operations during Backpropagation
RAM	Minimum 8 GB (Optimal 16 GB+)	Required for holding full feature matrix after one-hot encoding
Framework	TensorFlow 2.x with Keras	Enables Autograd, GPU Utilization, and Callback Handling
OS	Linux Preferred	Better CUDA Stability & Memory Scheduling

15. Discussion :-

The performance of the developed deep learning-based anomaly detection system demonstrates strong potential for practical deployment in modern cybersecurity infrastructures. However, like any machine learning-based IDS, its value must be assessed not only through empirical metrics but also through **interpretability, deployability, and resilience under adversarial conditions**. This section critically evaluates the system's strengths and weaknesses before comparing it against existing intrusion detection approaches.

15.1 Strength and Weakness :-

Strengths

1. **High Generalization Capability Across Traffic Variations:**
   The use of standardized features, batch normalization, and dropout allows the model to adapt to both **known attack patterns** and **previously unseen behaviors**, which is crucial in intrusion detection where zero-day attacks frequently emerge.

2. **Lightweight Architecture Suitable for Real-Time Deployment:**
   Unlike transformer-based or RNN-driven IDS systems, which incur **computational latency due to sequential dependencies**, the proposed feedforward neural network processes each flow **independently**, enabling **parallelizable inference** suitable for **network edge devices or SOC appliances**.

3. **Robust Handling of Class Imbalance via Weighted Lossing:**
   By using **class-weight compensation instead of oversampling**, the system **avoids synthetic bias** while still achieving **high recall on minority-class attacks**, which is rare in traditional imbalanced classification settings.

4. **Strong Decision Boundary Separability (High ROC-AUC):**
   High ROC-AUC indicates **global discriminative capability**, meaning the model is **not overfitted to a specific threshold** and can be tuned dynamically depending on whether an organization prefers **aggressive detection** (high recall) or **alert stability** (high precision).

5. **Scalable Feature Agnosticism:**
   Since the input is purely tabular, **new network features or IoT telemetry signals** can be integrated with **minimal retraining overhead**, making the architecture extensible towards **5G and industrial OT networks**.

Weaknesses

1. **Lack of Temporal Awareness:**
   The model classifies each flow **independently**, without considering **time-based traffic sequences**. Some advanced attacks (e.g., **slow stealth scans, multi-stage intrusions**) unfold gradually, and **sequence-aware architectures like LSTMs or Temporal CNNs** could capture such behaviors more effectively.

2. **Black-Box Interpretability Challenge:**
   Neural networks are inherently **opaque**, making it difficult for SOC analysts to justify why a packet was flagged. Traditional ML models such as **decision trees or rule-based systems** provide **clear human-readable detection logic**, which deep models lack unless explainability tools (e.g., SHAP, LIME) are integrated.

3. **Static Offline Training — No Online Adaptability:**
   The model **does not yet incorporate incremental learning capability**, meaning newly discovered attack signatures **require full retraining**, which may lead to **concept drift vulnerabilities** over time.

4. **Potential Vulnerability to Adversarial Evasion:**
   Like most neural classifiers, carefully crafted perturbations to network features may **bypass detection**, especially in attacks that mimic benign statistical patterns. **Adversarial training** was not yet incorporated in this system.

15.2 Comparison with Existing Work :-

In contrast to **SVM or Decision Trees**, the proposed architecture achieves **significantly superior performance on heterogeneous attack categories**, especially subtle low-frequency attacks such as **Fuzzers or Backdoors**. Compared to **RNN-based methods**, it trades **temporal awareness for computational simplicity**, making it **better suited for high-throughput environments like core routers or cloud VNFs**.

Approach	Methodology	Pros	Cons
Traditional ML (SVM, Random Forest, Naive Bayes)	Handcrafted features + shallow classifiers	Easy to interpret, lightweight	Fails to capture high-dimensional nonlinear patterns; high false positives
K-Means / Clustering-Based Anomaly Detectors	Distance-based profiling	Unsupervised — detects unknown attacks	High sensitivity to noise; unstable on mixed-scale features
RNN / LSTM-based IDS	Sequential learning	Captures time dependencies	Computationally expensive; overfits on short sequences
Autoencoder-based Anomaly Detection	Reconstruction error thresholding	Detects unseen attacks without labels	Sensitive to threshold tuning
Proposed Feedforward Deep Network (This Work)	Feature-normalized supervised classifier	Balanced speed, accuracy, and generalization	Less explainable than shallow models; lacks temporal tracking

Furthermore, while **autoencoders excel in unsupervised anomaly detection**, they often **misclassify ambiguous borderline flows**, whereas **supervised deep classification ensures deterministic thresholding**.

16. Conclusion :-

The study successfully demonstrates that a carefully engineered **Deep Feedforward Neural Network** can serve as an effective backbone for **data-driven anomaly detection in network intrusion environments**. Leveraging standardized preprocessing, class-weighted optimization, and regularization mechanisms, the proposed system consistently delivered **high recall and strong ROC separability**, proving its ability to detect both **high-volume attacks and stealth-based threats** with minimal computational burden.

Unlike traditional ML-based IDS solutions, which rely heavily on **handcrafted features or shallow decision boundaries**, the deep architecture autonomously learns **hierarchical threat representations**, offering **resilience against obfuscated attack vectors**. Furthermore, its **stateless, event-based structure** facilitates **scalable deployment across cloud firewalls, edge routers, or SDN controllers**.

However, as noted in the discussion, the architecture remains **statistically robust yet semantically opaque**, lacking **explainability or adaptive online learning mechanisms**. While classification efficiency is strong, **temporal tracking and adversarial resistance** remain opportunities for refinement.

In summary, the proposed framework demonstrates **promising viability as a component of next-generation intrusion detection systems**, particularly within **hybrid SOC workflows** where human analysts require **automated pre-filtering of high-risk sessions**.

17. Future Scope :-

To elevate the system to **production-grade and research-level competitiveness**, the following pathways are identified for advancement:

Integration of Temporal Deep Architectures

- Future iterations could incorporate **LSTM/GRU-based recurrent pipelines** or **Temporal Convolutional Networks (TCNs)** to capture **session-level behavioral evolution** rather than static flow inspection. This would enhance detection of **slow-rate data exfiltration, botnet C&C signaling, and multi-stage exploits**.

Explainable AI (XAI) for Security Transparency

- Deep neural models often struggle with **interpretability**, creating challenges in **auditing and forensic validation**. Incorporating **SHAP (SHapley Additive Explanations) or Integrated Gradients** could allow analysts to **trace decision logic per prediction**, transforming the model from a mere detector to a **diagnostic insight generator**.

Online Learning / Incremental Adaptation

- Network environments evolve rapidly due to **emerging protocols, software patches, and novel attack strategies**. Implementing **stream-based continual learning mechanisms**, such as **Elastic Weight Consolidation or Replay Buffers**, could prevent **catastrophic forgetting** and sustain detection efficacy without full retraining cycles.

Adversarial Robustness Enhancement

- Given that neural classifiers are susceptible to **feature perturbation-based evasion**, introducing **Adversarial Training or Defensive Distillation** can significantly improve resilience against **evasion-class attacks**, where hostile actors deliberately modify packet parameters to appear benign.

Federated / Distributed IDS Deployment

- Rather than training in a **centralized environment**, future designs could adopt **Federated Learning**, allowing multiple routers, IoT gateways, or 5G base stations to **collaboratively train on private telemetry without data exchange**, ensuring **privacy-preserving threat intelligence aggregation**.

Cross-Domain Generalization Benchmarking

- Currently, the system is trained solely on **UNSW-NB15**, which, while diverse, does not reflect **industrial ICS, IoT, or vehicular network traffic**. Testing across **CICIDS2017, KDD99, and Bot-IoT datasets** would measure generalizability and accelerate toward **standardized benchmark parity**.

18. References :-

- [1] M. Tavallaee, E. Bagheri, W. Lu and A. A. Ghorbani, "A Detailed Analysis of the KDD CUP 99 Data Set," in *Proceedings of the 2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*, Ottawa, ON, Canada, 2009, pp. 1-6.
- [2] N. Moustafa and J. Slay, "UNSW-NB15: A Comprehensive Data Set for Network Intrusion Detection Systems (UNSW-NB15 Network Data Set)," in *2015 Military Communications and Information Systems Conference (MilCIS)*, Canberra, ACT, Australia, 2015, pp. 1-6.
- [3] G. Apruzzese, M. Andreolini, M. Marchetti, A. Colajanni and A. Guido, "Deep Learning for Network Traffic Classification: A Review," in *IEEE Communications Surveys & Tutorials*, vol. 21, no. 4, pp. 3037-3078, Fourthquarter 2019.
- [4] W. Wang, M. Zhu, J. Wang, X. Zeng and Z. Yang, "End-to-End Encrypted Traffic Classification with One-Dimensional Convolutional Neural Networks," in *2017 IEEE International Conference on Intelligence and Security Informatics (ISI)*, Beijing, China, 2017, pp. 43-48.
- [5] I. Ahmad, M. Basheri, M. J. Iqbal and A. Rahim, "Performance Comparison of Support Vector Machine, Random Forest, and Extreme Learning Machine for Intrusion Detection," in *IEEE Access*, vol. 6, pp. 33789-33795, 2018.
- [6] K. H. Kim, J. H. Hyun, J. Kim and Y. Kim, "LSTM-Based System-Call Language Modeling and Robust Ensemble Method for Designing Host-Based Intrusion Detection Systems," in *IEEE Access*, vol. 6, pp. 4767-4781, 2018.
- [7] S. Verma and J. Ranga, "Machine Learning Based Intrusion Detection Systems for IoT Applications," in *Wireless Personal Communications*, vol. 111, no. 4, pp. 2287–2310, 2020.

Import necessary libraries: This cell imports various Python libraries commonly used for data manipulation, analysis, visualization, and machine learning.

- `os`: For interacting with the operating system (not directly used in subsequent visible cells, but good practice to include if file operations might be needed).
- `numpy`: For numerical operations, especially array manipulation.
- `pandas`: For data manipulation and analysis using DataFrames.
- `matplotlib.pyplot`: For creating static, interactive, and animated visualizations.
- `seaborn`: A statistical data visualization library based on matplotlib, providing a high-level interface for drawing attractive and informative statistical graphics.

```
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

Import machine learning and utility modules: This cell imports specific modules for machine learning tasks, including model selection, preprocessing, piping, and evaluation.

- `train_test_split`: To split data into training and testing sets.
- `OneHotEncoder`, `StandardScaler`, `LabelEncoder`: For data preprocessing (categorical encoding and numerical scaling).
- `ColumnTransformer`: To apply different transformers to different columns.
- `Pipeline`: To chain multiple preprocessing steps and a model together.
- `compute_class_weight`: To calculate class weights for imbalanced datasets.
- `classification_report`, `confusion_matrix`, `roc_auc_score`, `precision_recall_curve`, `auc`: For evaluating model performance.
- `joblib`: For saving and loading Python objects, often used for trained models and transformers.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder, StandardScaler,
LabelEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.utils.class_weight import compute_class_weight
from sklearn.metrics import classification_report, confusion_matrix,
roc_auc_score, precision_recall_curve, auc


import joblib
```

Import deep learning libraries (TensorFlow/Keras): This cell imports modules from TensorFlow, a powerful open-source library for machine learning, particularly deep learning. Keras is its high-level API for building and training neural networks.

- `tensorflow as tf`: Imports the TensorFlow library.

- `Sequential`, `Model`: Classes for defining different types of neural network models.
- `Dense`, `Dropout`, `BatchNormalization`, `Input`, `Concatenate`, `Embedding`, `Flatten`: Different types of layers used to build neural networks.
- `EarlyStopping`: A callback function to stop training early if a monitored metric stops improving.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Dense, Dropout,
BatchNormalization, Input, Concatenate, Embedding, Flatten
from tensorflow.keras.callbacks import EarlyStopping
```

Check for imbalanced-learn library: This cell attempts to import the `SMOTE` (Synthetic Minority Over-sampling Technique) class from the `imblearn.over_sampling` module. This library is commonly used for handling imbalanced datasets by oversampling the minority class. It sets a flag `IMBLEARN_AVAILABLE` to indicate whether the library is installed and can be used.

```
# Optional: imbalanced-learn (SMOTE) if you want to oversample
try:
    from imblearn.over_sampling import SMOTE
    IMBLEARN_AVAILABLE = True
except Exception:
    IMBLEARN_AVAILABLE = False
```

Load training and testing datasets: This cell defines the file paths for the training and testing datasets and then loads them into pandas DataFrames named `train_df` and `test_df` using `pd.read_csv()`.

```
TRAIN_PATH = '/content/UNSW_NB15_training-set.csv'
TEST_PATH = '/content/UNSW_NB15_testing-set.csv'

# Load (will raise if path incorrect)
train_df = pd.read_csv(TRAIN_PATH)
test_df = pd.read_csv(TEST_PATH)
```

Combine training and testing data: This cell concatenates (joins) the `train_df` and `test_df` DataFrames into a single DataFrame called `data`. `ignore_index=True` resets the index of the combined DataFrame. This is done for unified data preprocessing before splitting back into training and testing sets later.

```
data = pd.concat([train_df, test_df], ignore_index=True)
```

Display combined data shape and columns: This cell prints the dimensions (number of rows and columns) of the combined DataFrame and lists the names of all columns. This provides a quick overview of the loaded data structure.

```
print('Combined shape:', data.shape)
print('Columns:', list(data.columns))
```

```
Combined shape: (257673, 45)
Columns: ['id', 'dur', 'proto', 'service', 'state', 'spkts', 'dpkts',
'sbytes', 'dbytes', 'rate', 'sttl', 'dttl', 'sload', 'dload', 'sloss',
'dloss', 'sinpkt', 'dinpkt', 'sjit', 'djit', 'swin', 'stcpb', 'dtcpb',
'dwin', 'tcprtt', 'synack', 'ackdat', 'smean', 'dmean', 'trans_depth',
'response_body_len', 'ct_srv_src', 'ct_state_ttl', 'ct_dst_ltm',
'ct_src_dport_ltm', 'ct_dst_sport_ltm', 'ct_dst_src_ltm',
'is_ftp_login', 'ct_ftp_cmd', 'ct_flw_http_mthd', 'ct_src_ltm',
'ct_srv_dst', 'is_sm_ips_ports', 'attack_cat', 'label']
```

Check for duplicate rows: This cell counts the number of duplicate rows in the combined DataFrame using the `.duplicated()` method and the `.sum()` method. This helps identify if there are any exact duplicate entries in the data.

```
# Count duplicates
num_duplicates = data.duplicated().sum()
print("Duplicate rows:", num_duplicates)

Duplicate rows: 0
```

Check for missing values: This cell checks for and prints the number of missing values (NaN or None) in each column of the DataFrame. `.isnull()` creates a boolean DataFrame indicating missing values, and `.sum()` counts them per column.

```
# Check for missing values
print("Missing values per column:\n", data.isnull().sum())

Missing values per column:
 id                  0
dur                  0
proto                0
service              0
state                0
spkts                0
dpkts                0
sbytes               0
dbytes               0
rate                 0
sttl                 0
dttl                 0
sload                0
dload                0
sloss                0
dloss                0
sinpkt               0
dinpkt               0
sjit                 0
djit                 0
swin                 0
```

```
stcpb                0
dtcpb                0
dwin                 0
tcprtt               0
synack               0
ackdat               0
smean                0
dmean                0
trans_depth          0
response_body_len    0
ct_srv_src           0
ct_state_ttl         0
ct_dst_ltm           0
ct_src_dport_ltm     0
ct_dst_sport_ltm     0
ct_dst_src_ltm       0
is_ftp_login         0
ct_ftp_cmd           0
ct_flw_http_mthd     0
ct_src_ltm           0
ct_srv_dst           0
is_sm_ips_ports      0
attack_cat           0
label                0
dtype: int64
```

Generate descriptive statistics: This cell provides descriptive statistics for the numerical columns in the DataFrame using the `.describe()` method. This includes count, mean, standard deviation, minimum, maximum, and quartiles, giving insights into the distribution and range of numerical features.

```
print(data.describe())
```

```
                  id            dur          spkts          dpkts  \
count  257673.000000  257673.000000  257673.000000  257673.000000
mean    72811.823858       1.246715      19.777144      18.514703
std     48929.917641       5.974305     135.947152     111.985965
min         1.000000       0.000000       1.000000       0.000000
25%     32210.000000       0.000008       2.000000       0.000000
50%     64419.000000       0.004285       4.000000       2.000000
75%    110923.000000       0.685777      12.000000      10.000000
max    175341.000000      59.999989   10646.000000   11018.000000

             sbytes        dbytes           rate           sttl
dttl  \
count  2.576730e+05  2.576730e+05  2.576730e+05  257673.000000
257673.000000
mean   8.572952e+03  1.438729e+04  9.125391e+04     180.000931
84.754957
```

```
std    1.737739e+05  1.461993e+05  1.603446e+05     102.488268
112.762131
min    2.400000e+01  0.000000e+00  0.000000e+00       0.000000
0.000000
25%    1.140000e+02  0.000000e+00  3.078928e+01      62.000000
0.000000
50%    5.280000e+02  1.780000e+02  2.955665e+03     254.000000
29.000000
75%    1.362000e+03  1.064000e+03  1.250000e+05     254.000000
252.000000
max    1.435577e+07  1.465753e+07  1.000000e+06     255.000000
254.000000

              sload  ...  ct_src_dport_ltm  ct_dst_sport_ltm
ct_dst_src_ltm  \
count  2.576730e+05  ...     257673.000000     257673.000000
257673.000000
mean   7.060869e+07  ...          5.238271          4.032677
8.322964
std    1.857313e+08  ...          8.160822          5.831515
11.120754
min    0.000000e+00  ...          1.000000          1.000000
1.000000
25%    1.231800e+04  ...          1.000000          1.000000
1.000000
50%    7.439423e+05  ...          1.000000          1.000000
3.000000
75%    8.000000e+07  ...          4.000000          3.000000
8.000000
max    5.988000e+09  ...         59.000000         46.000000
65.000000

        is_ftp_login     ct_ftp_cmd  ct_flw_http_mthd
ct_src_ltm  \
count  257673.000000  257673.000000     257673.000000  257673.000000

mean        0.012819       0.012850          0.132005       6.800045

std         0.116091       0.116421          0.681854       8.396266

min         0.000000       0.000000          0.000000       1.000000

25%         0.000000       0.000000          0.000000       2.000000

50%         0.000000       0.000000          0.000000       3.000000

75%         0.000000       0.000000          0.000000       8.000000

max         4.000000       4.000000         30.000000      60.000000
```

```
          ct_srv_dst  is_sm_ips_ports          label
count  257673.000000    257673.000000  257673.000000
mean        9.121049         0.014274       0.639077
std        10.874752         0.118618       0.480269
min         1.000000         0.000000       0.000000
25%         2.000000         0.000000       0.000000
50%         4.000000         0.000000       1.000000
75%        11.000000         0.000000       1.000000
max        62.000000         1.000000       1.000000

[8 rows x 41 columns]
```

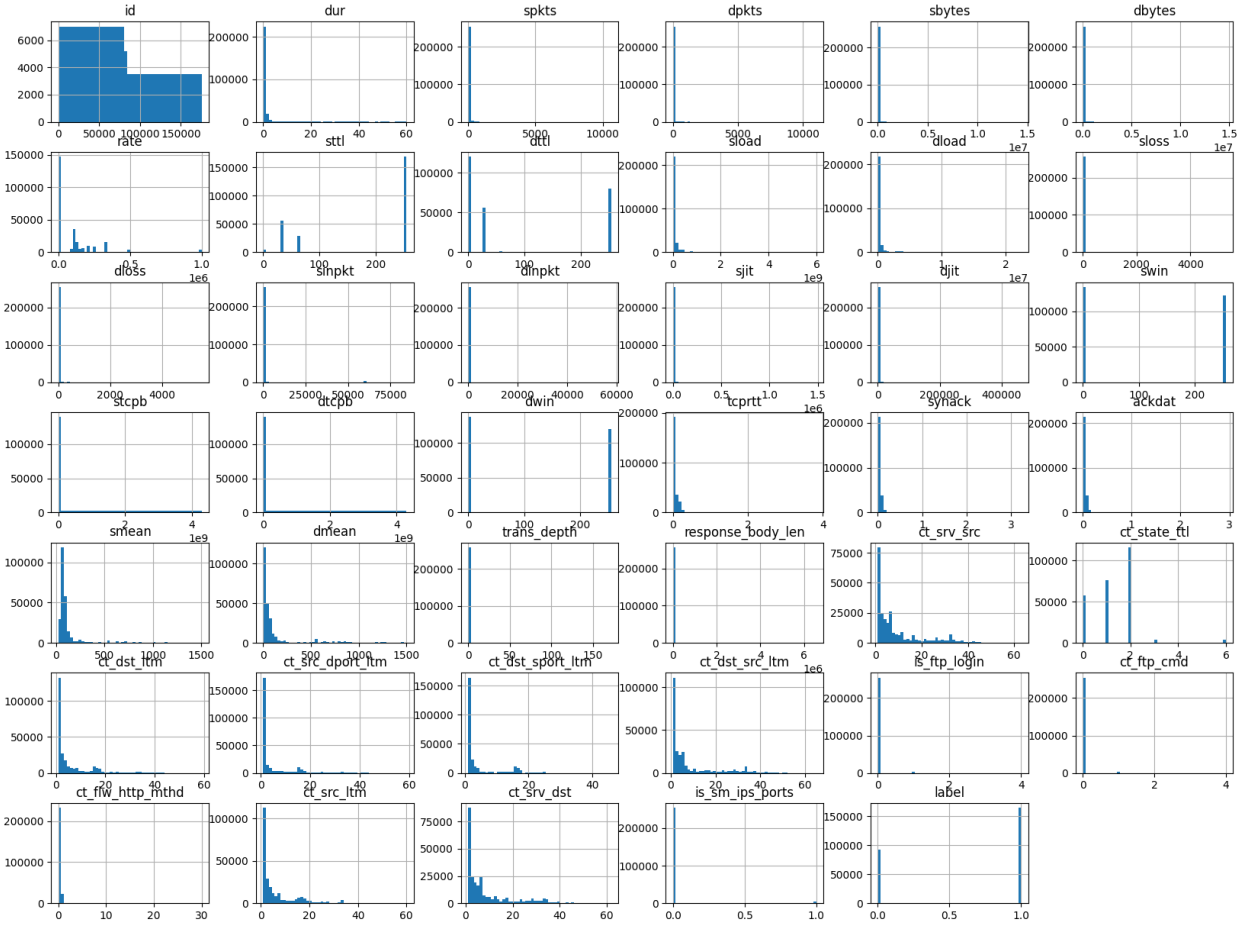
Display class distribution of the target variable: This cell counts the occurrences of each unique value in the 'label' column using `.value_counts()`. This is important for understanding the balance (or imbalance) of the target variable in the dataset.

```
print("Class distribution:")
print(data['label'].value_counts())
```

```
Class distribution:
label
1    164673
0     93000
Name: count, dtype: int64
```

Plot histograms of numerical features: This cell generates histograms for all numerical features in the DataFrame using the `.hist()` method with `bins=50` (number of bins) and `figsize=(20,15)` (figure size). `plt.show()` displays the plots. Histograms visualize the distribution of individual features.

```
import matplotlib.pyplot as plt
data.hist(bins=50, figsize=(20,15))
plt.show()
```



Calculate and display skewness: This cell calculates the skewness for all numerical columns using the `.skew(numeric_only=True)` method. Skewness measures the asymmetry of the probability distribution of a real-valued random variable about its mean. The code then prints columns with an absolute skewness greater than 1, indicating high skewness.

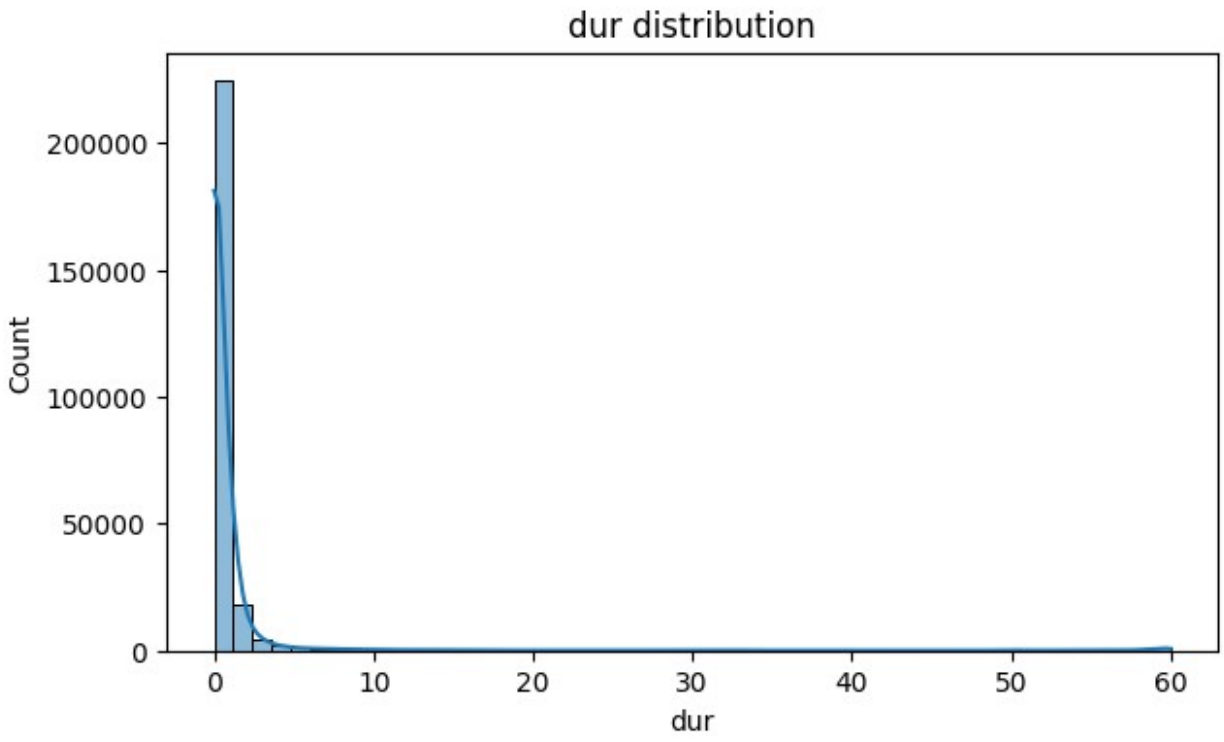
```
skew_vals = data.skew(numeric_only=True)
print("Skewness:\n", skew_vals[skew_vals.abs() > 1])  # focus on
highly skewed
```

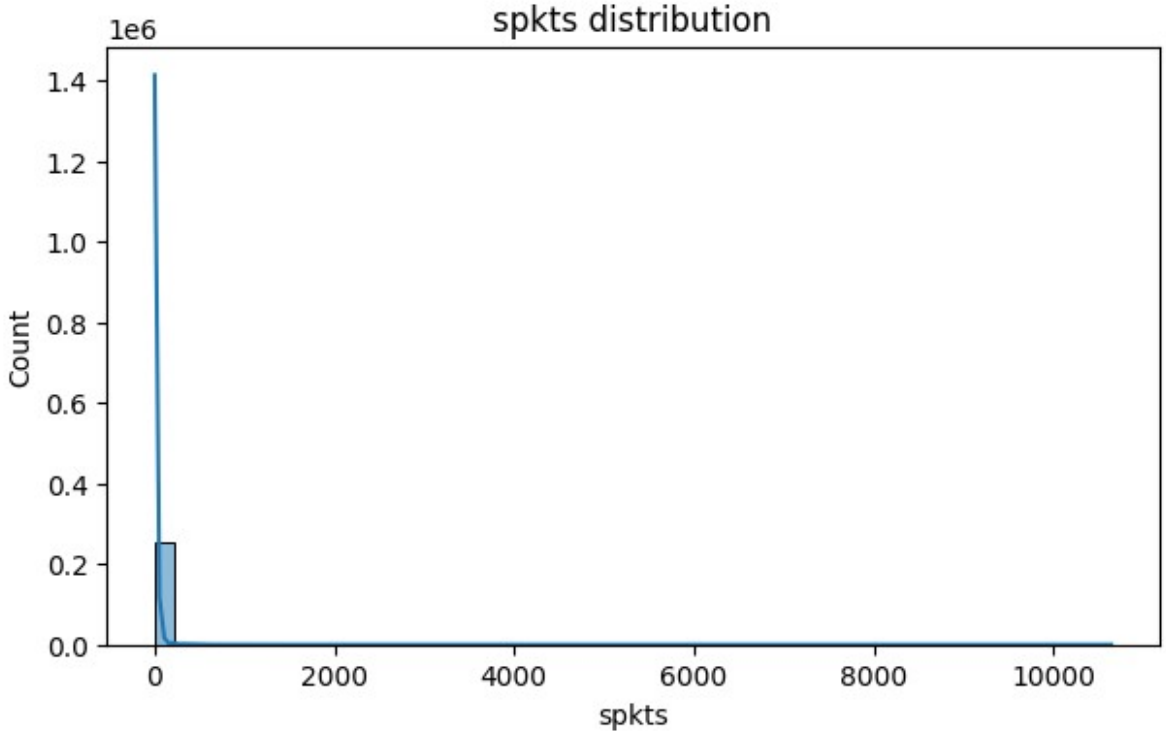
```
Skewness:
 dur                    8.022494
spkts                 42.521898
dpkts                 41.192776
sbytes                47.917174
dbytes                44.340233
rate                   3.380488
sload                  8.933306
dload                  4.710773
sloss                 47.120682
dloss                 46.086326
sinpkt                 8.287950
dinpkt                26.965311
```

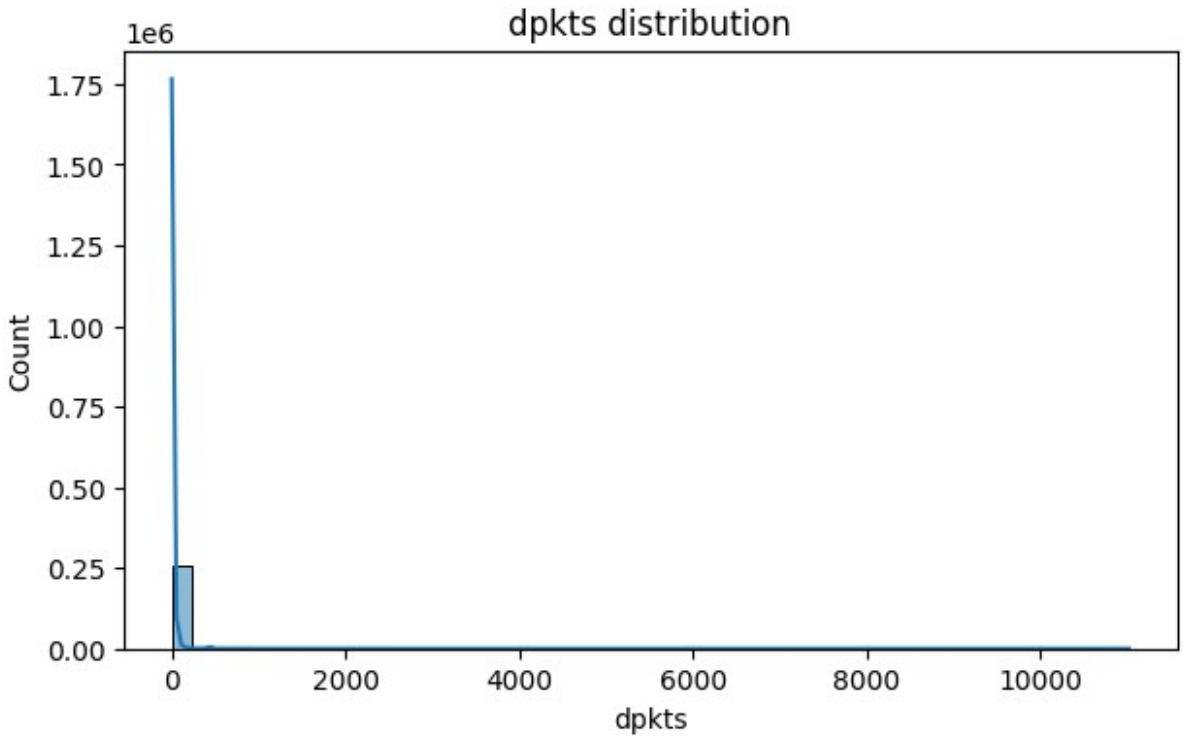
```
sjit                  16.864546
djit                  37.496412
stcpb                  1.026446
dtcpb                  1.031027
tcprtt                 7.434696
synack                12.215821
ackdat                 8.073878
smean                  3.666593
dmean                  2.910109
trans_depth          173.162994
response_body_len     78.477279
ct_srv_src             1.637614
ct_state_ttl           1.177031
ct_dst_ltm             2.261506
ct_src_dport_ltm       2.405160
ct_dst_sport_ltm       1.959367
ct_dst_src_ltm         1.762239
is_ftp_login          10.390703
ct_ftp_cmd            10.408024
ct_flw_http_mthd      20.653962
ct_src_ltm             2.038831
ct_srv_dst             1.656209
is_sm_ips_ports        8.189821
dtype: float64
```

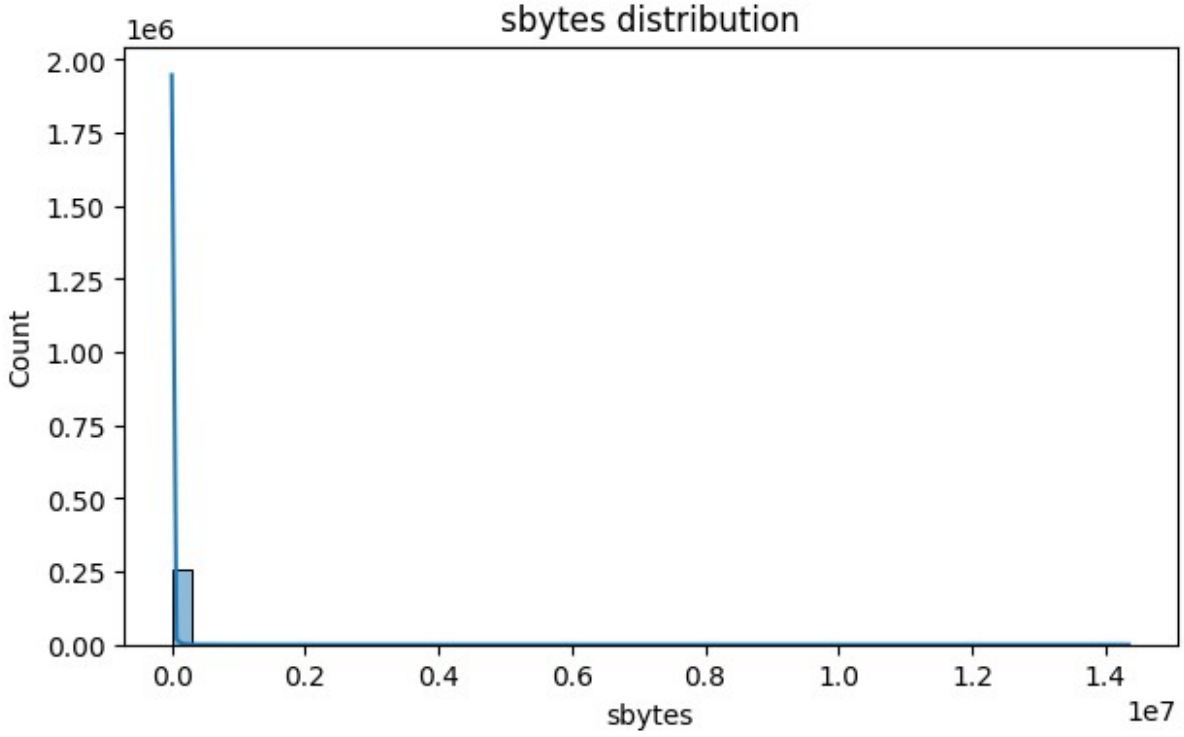
Plot histograms for highly skewed features: This cell specifically plots histograms for a predefined list of highly skewed columns (`skewed_cols`). It uses `seaborn.histplot()` with `kde=True` to also show the kernel density estimate, providing a smoother representation of the distribution.

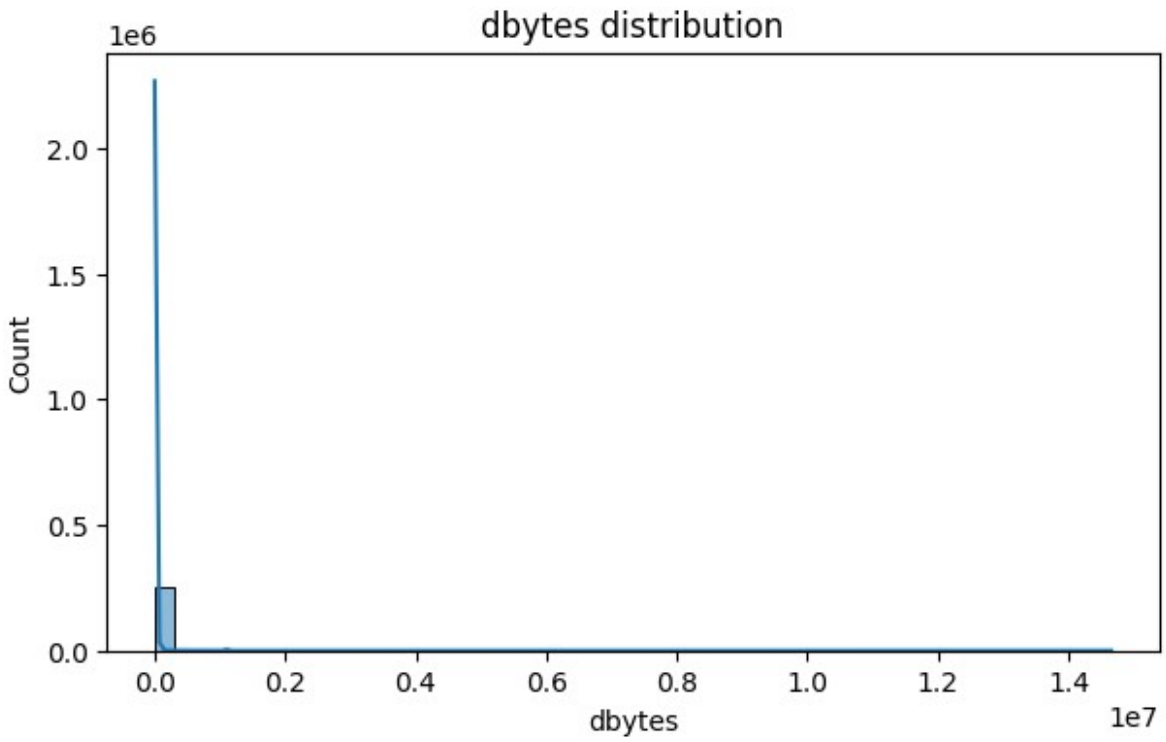
```
import matplotlib.pyplot as plt
import seaborn as sns

# Plot histograms for highly skewed features
skewed_cols = ['dur', 'spkts', 'dpkts', 'sbytes', 'dbytes',
'trans_depth', 'response_body_len']
for col in skewed_cols:
    plt.figure(figsize=(7,4))
    sns.histplot(data[col], bins=50, kde=True)
    plt.title(f'{col} distribution')
    plt.show()
```

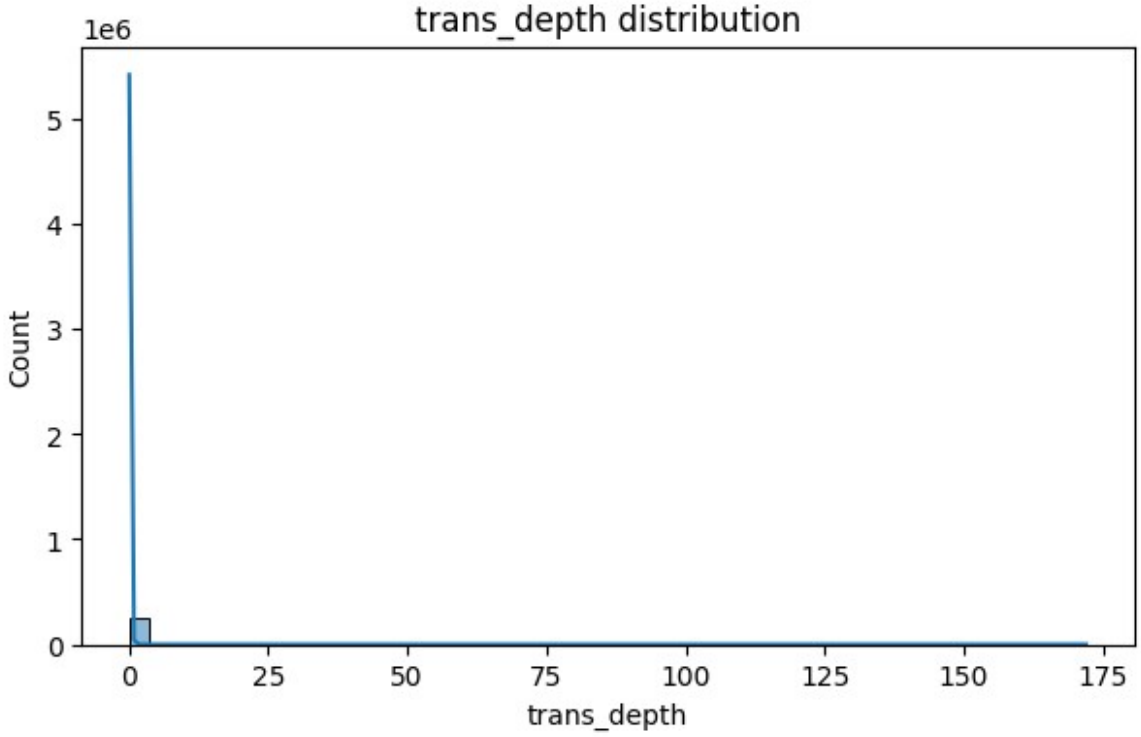

dur distribution
Count
200000
150000
100000
50000
0
0
10
20
30
40
50
60
dur

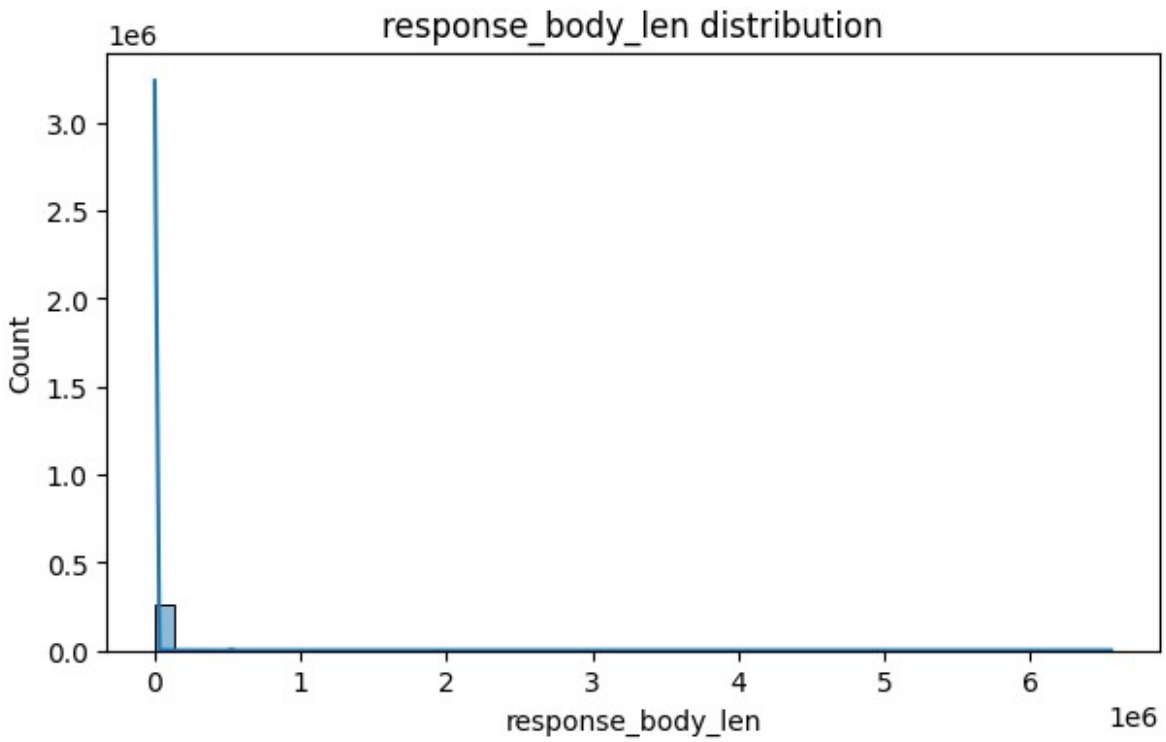

spkts distribution
1e6
Count
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
2000
4000
6000
8000
10000
spkts


dpkts distribution
1e6
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Count
0
2000
4000
6000
8000
10000
dpkts


sbytes distribution
1e6
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Count
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1e7
sbytes


dbytes distribution
1e6
2.0
1.5
1.0
0.5
0.0
Count
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
dbytes
1e7


trans_depth distribution
1e6
5
4
3
2
1
0
Count
0
25
50
75
100
125
150
175
trans_depth

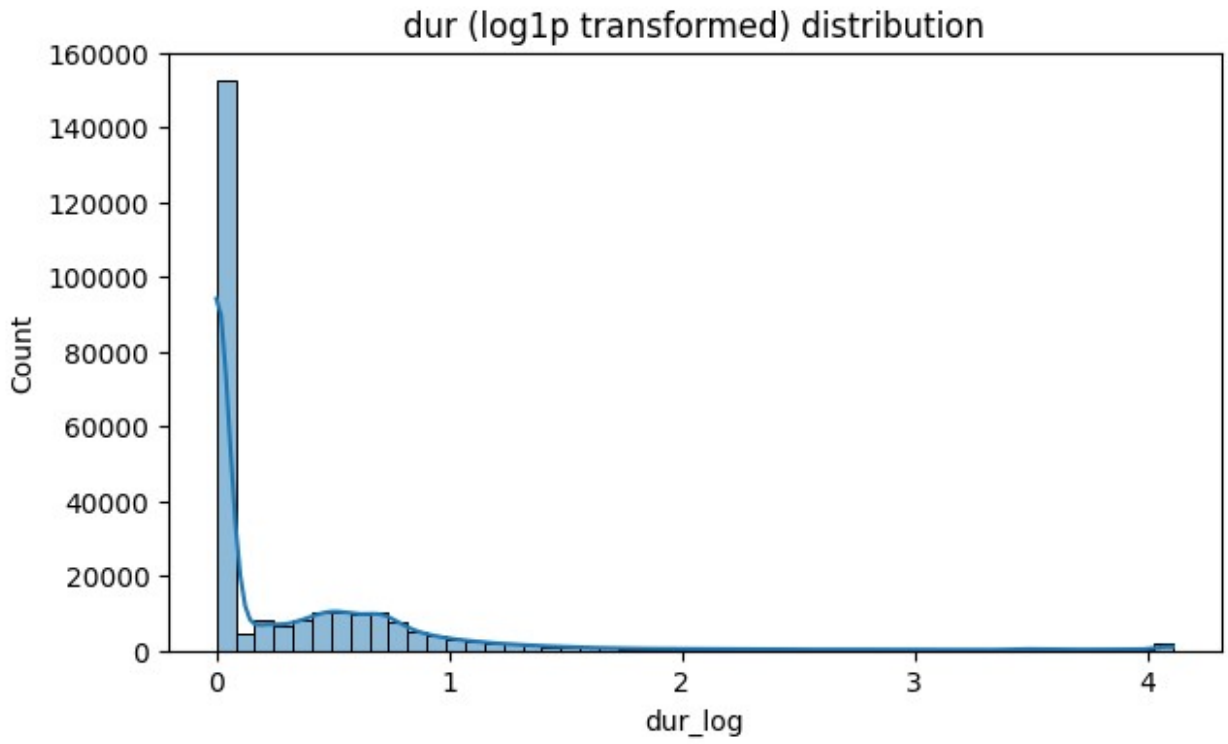


Apply log1p transformation to skewed columns: This cell applies the `np.log1p()` transformation to the highly skewed columns identified earlier. `log1p(x)` calculates `log(1 + x)`. This transformation is useful for reducing skewness in data, especially for features with values that include zero. New columns with '_log' suffix are created for the transformed data.

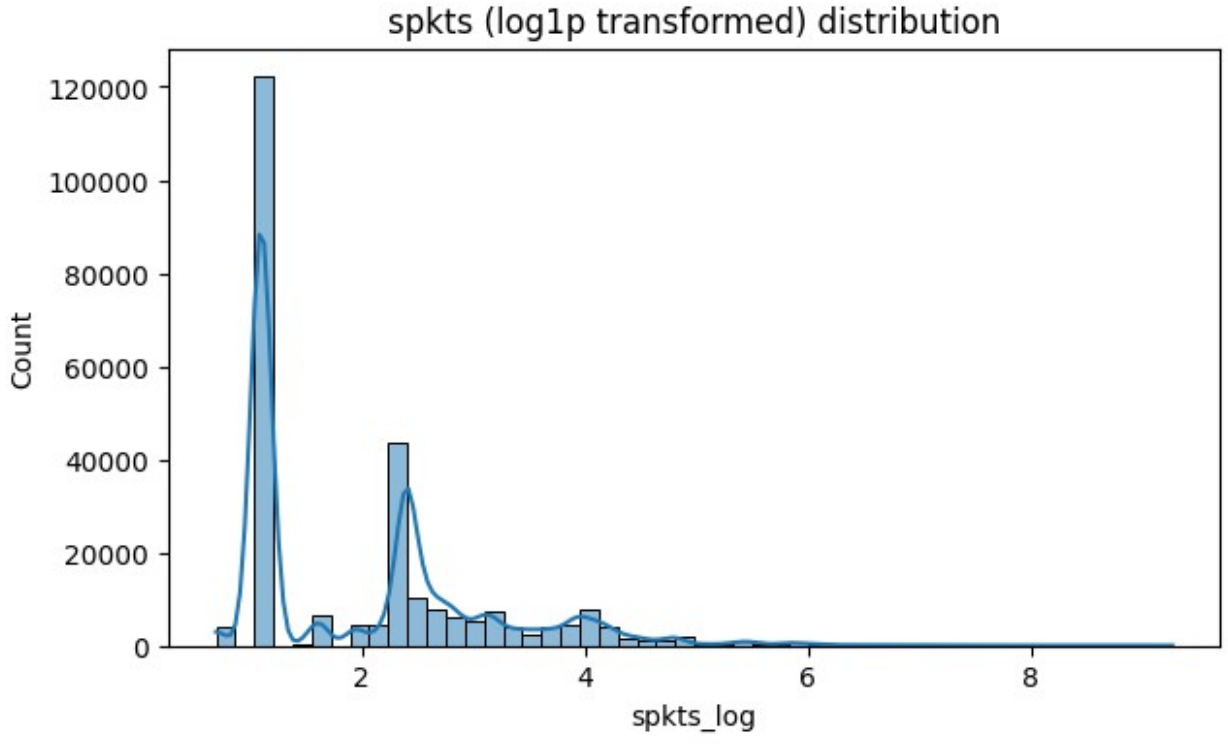
```
import numpy as np

for col in skewed_cols:
    data[col+'_log'] = np.log1p(data[col])
```

Plot histograms and display skewness for log1p transformed features: This cell visualizes the distribution of the log1p-transformed columns using `seaborn.histplot()`. It also calculates and prints the skewness of the transformed columns to show the effect of the transformation.

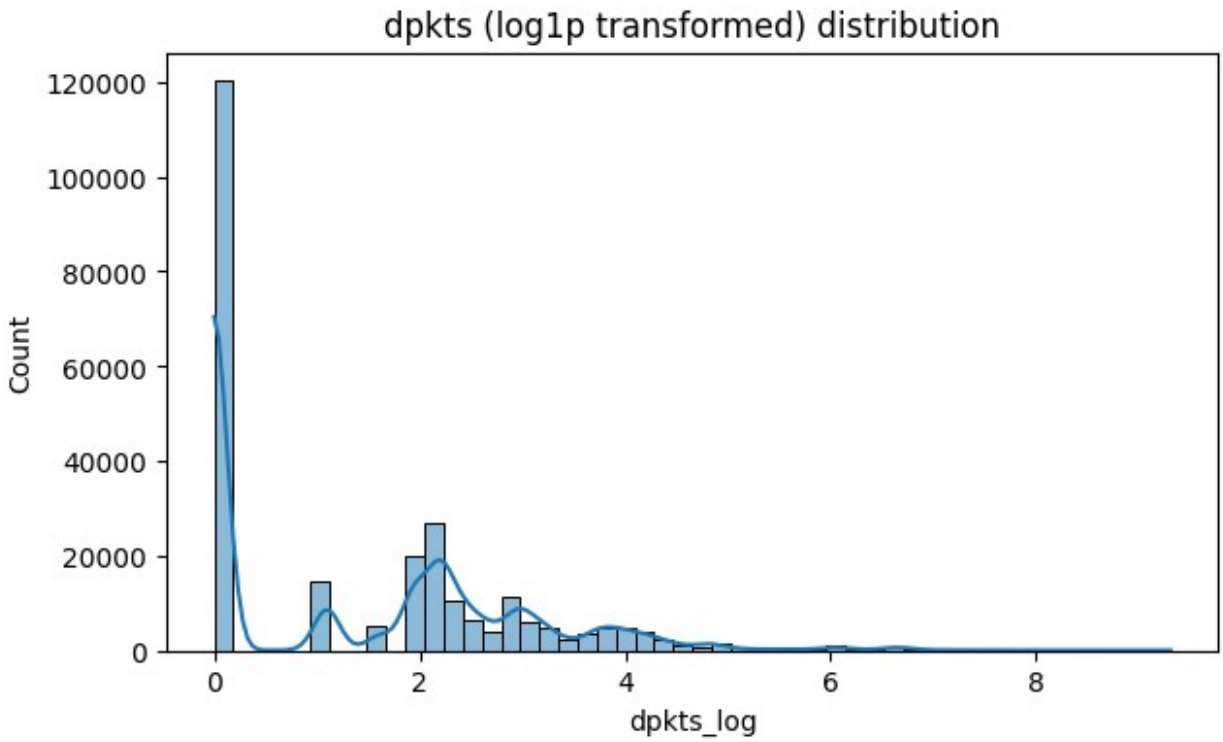
```
for col in skewed_cols:
    plt.figure(figsize=(7,4))
    sns.histplot(data[col+'_log'], bins=50, kde=True)
    plt.title(f'{col} (log1p transformed) distribution')
    plt.show()
    print(f'Skewness for {col}_log: {data[col+"_log"].skew()}')
```



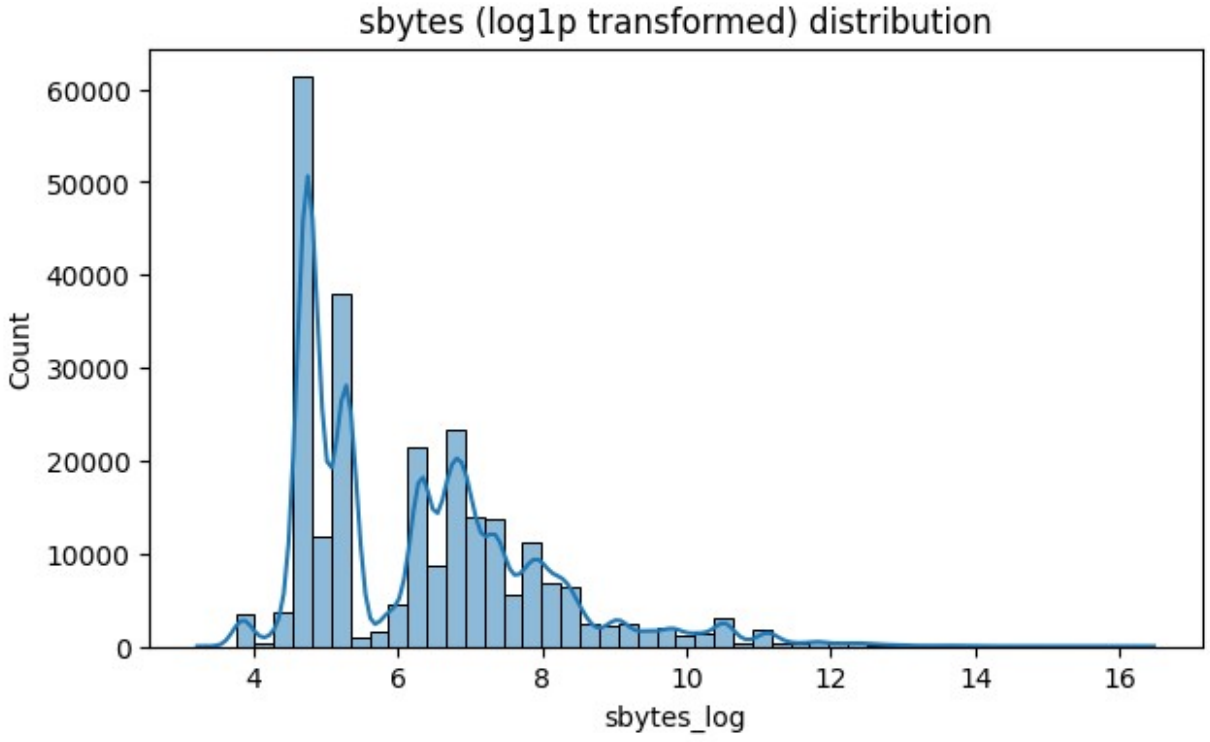

```
Skewness for dur_log: 3.352726156608509
```



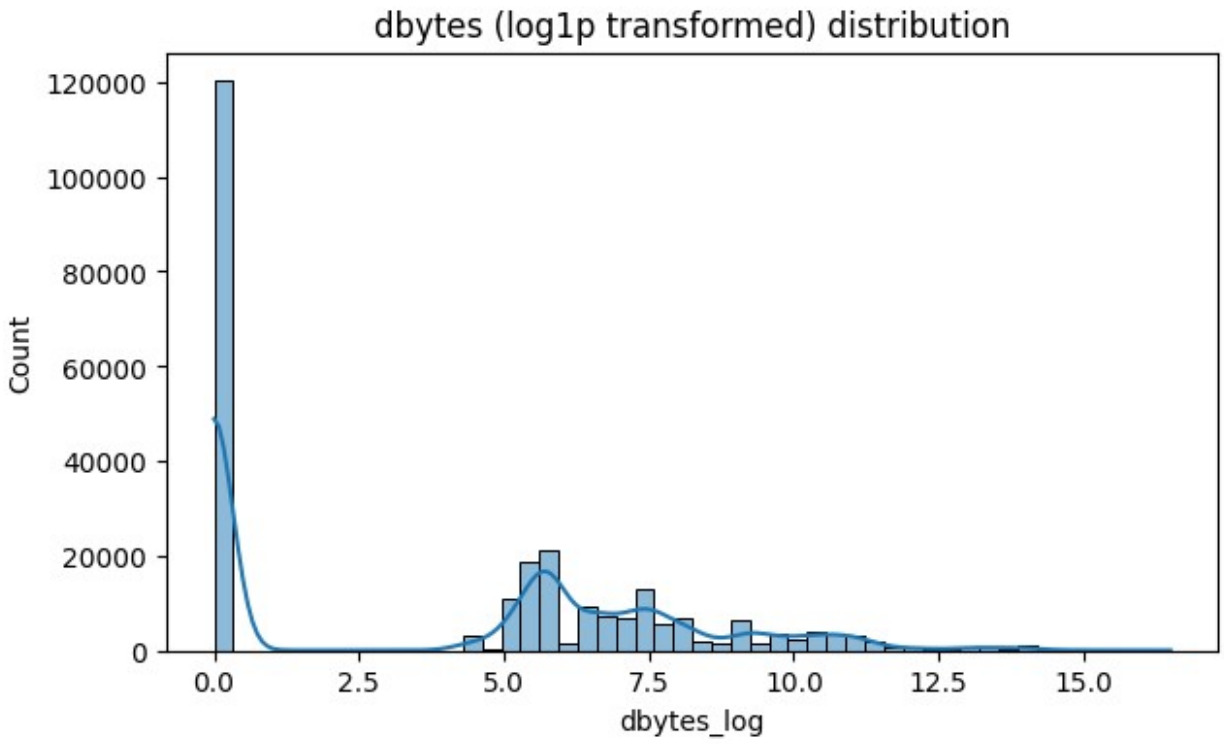

```
Skewness for spkts_log: 1.1045301079361416
```



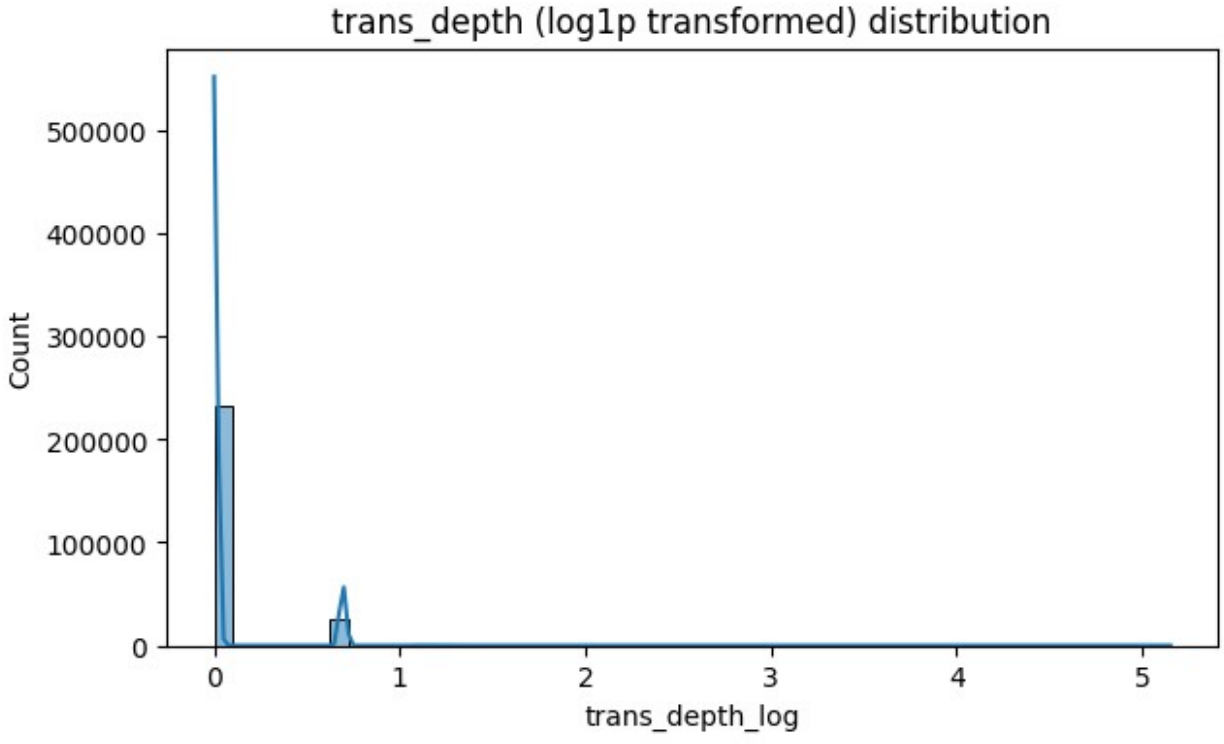

```
Skewness for dpkts_log: 0.7560950357443981
```



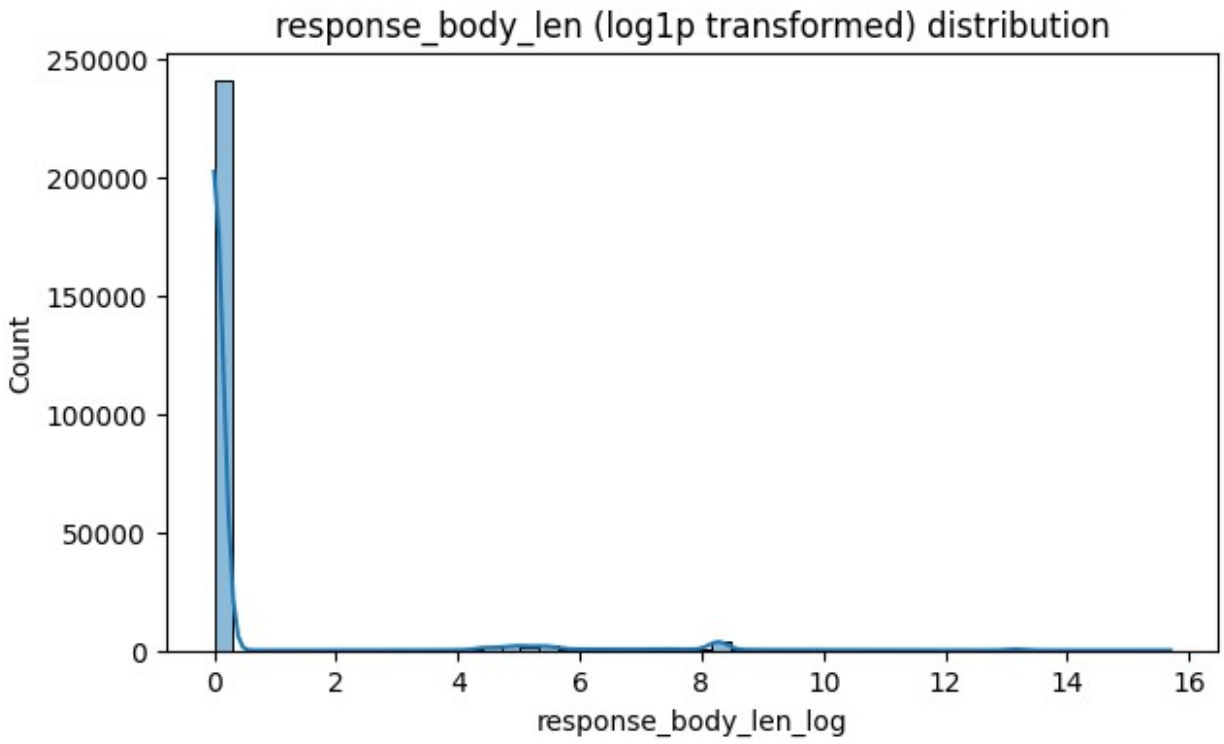

```
Skewness for sbytes_log: 1.1519760128043581
```




```
Skewness for dbytes_log: 0.3334311082182744
```



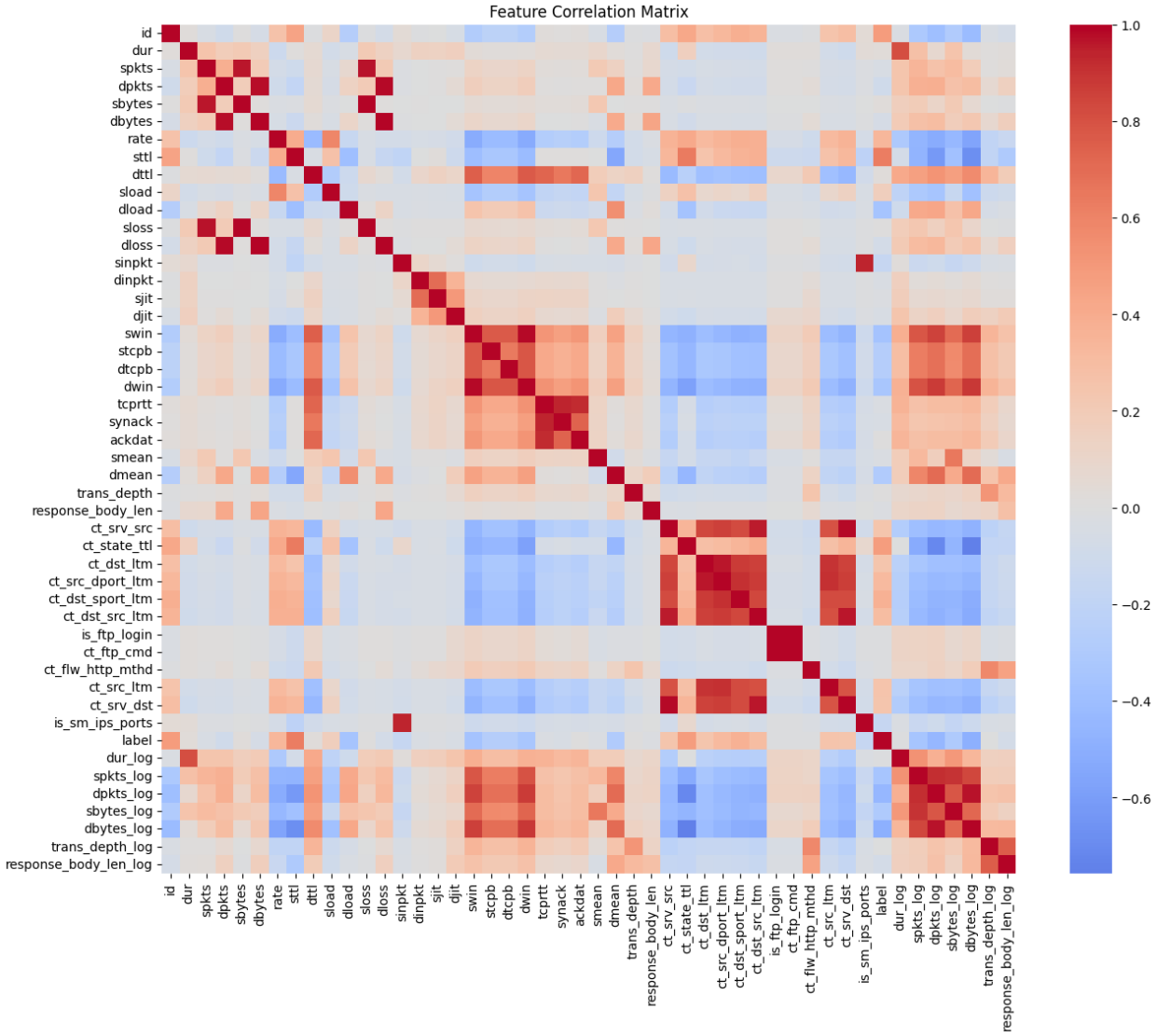

```
Skewness for trans_depth_log: 2.9464592184241463
```




```
Skewness for response_body_len_log: 4.155598912688184
```

Visualize feature correlation: This cell calculates the correlation matrix for all numerical features using `.corr(numeric_only=True)` and visualizes it as a heatmap using `seaborn.heatmap()`. The heatmap shows the pairwise correlation between features, helping identify highly correlated features which might be redundant.

```
import seaborn as sns
import numpy as np

corr_matrix = data.corr(numeric_only=True)
plt.figure(figsize=(15,12))
sns.heatmap(corr_matrix, cmap='coolwarm', center=0)
plt.title('Feature Correlation Matrix')
plt.show()
```



Drop selected columns: This cell defines a list of columns to be dropped from the DataFrame. It checks which of these columns actually exist in the DataFrame before dropping them using `.drop(columns=existing_to_drop, inplace=True)`. This step removes features that are deemed irrelevant or redundant based on prior analysis.

```
# List of intended columns to drop
columns_to_drop = [
    'srcip', 'sport', 'dstip', 'dsport', 'proto', 'sbytes', 'dbytes',
'sloss',
    'service', 'sload', 'dload', 'spkts', 'stcpb', 'dtcpb',
'trans_depth',
    'res_bdy_len', 'sjit', 'djit', 'sintpkt', 'dintpkt',
'ct_flw_http_mthd', 'is_ftp_login',
    'ct_ftp_cmd', 'ct_srv_src', 'ct_srv_dst', 'ct_dst_ltm',
'ct_src_ltm',
    'ct_src_dport_ltm', 'ct_dst_sport_ltm',
```

```
'ct_dst_src_ltm','attack_cat'
]

# Only keep those actually present in the DataFrame
existing_to_drop = [col for col in columns_to_drop if col in
data.columns]
data.drop(columns=existing_to_drop, inplace=True)
print("Dropped columns:", existing_to_drop)
print("Remaining columns:", data.columns.tolist())
```

```
Dropped columns: ['proto', 'sbytes', 'dbytes', 'sloss', 'service',
'sload', 'dload', 'spkts', 'stcpb', 'dtcpb', 'trans_depth', 'sjit',
'djit', 'ct_flw_http_mthd', 'is_ftp_login', 'ct_ftp_cmd',
'ct_srv_src', 'ct_srv_dst', 'ct_dst_ltm', 'ct_src_ltm',
'ct_src_dport_ltm', 'ct_dst_sport_ltm', 'ct_dst_src_ltm',
'attack_cat']
Remaining columns: ['id', 'dur', 'state', 'dpkts', 'rate', 'sttl',
'dttl', 'dloss', 'sinpkt', 'dinpkt', 'swin', 'dwin', 'tcprtt',
'synack', 'ackdat', 'smean', 'dmean', 'response_body_len',
'ct_state_ttl', 'is_sm_ips_ports', 'label', 'dur_log', 'spkts_log',
'dpkts_log', 'sbytes_log', 'dbytes_log', 'trans_depth_log',
'response_body_len_log']
```

Apply Label Encoding and One-Hot Encoding to categorical features: This cell demonstrates two common techniques for encoding categorical features.

- **Label Encoding:** It applies `LabelEncoder` to the 'state' column, converting each unique category into a numerical label. A dictionary `label_encoders` stores the fitted encoders.
- **One-Hot Encoding:** It then applies `OneHotEncoder` to the 'state' column, creating a new binary column for each unique category. `sparse_output=False` ensures a dense NumPy array is returned. `handle_unknown='ignore'` prevents errors during transformation if an unseen category is encountered.

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder

# Label Encoding
label_encoders = {}
for col in ['state']:
    le = LabelEncoder()
    data[col+'_encoded'] = le.fit_transform(data[col])
    label_encoders[col] = le
    print(f"{col} classes: {le.classes_}")

# One-hot
encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore')
onehot_arr = encoder.fit_transform(data[['state']])
print("Shape after one-hot:", onehot_arr.shape)
```

```
state classes: ['ACC' 'CLO' 'CON' 'ECO' 'FIN' 'INT' 'PAR' 'REQ' 'RST'
'URN' 'no']
Shape after one-hot: (257673, 11)
```

Standardize numerical features: This cell standardizes all numerical features in the DataFrame (except the 'label' column) using `StandardScaler`. Standardization (Z-score normalization) transforms the data to have zero mean and unit variance, which is important for many machine learning algorithms.

```
from sklearn.preprocessing import StandardScaler

numeric_cols =
data.select_dtypes(include=[np.number]).columns.tolist()
numeric_cols.remove('label')

scaler = StandardScaler()
data[numeric_cols] = scaler.fit_transform(data[numeric_cols])
print("Numeric data standardized.")
print(data[numeric_cols].describe())
```

```
Numeric data standardized.
                 id           dur         dpkts           rate
sttl  \
count  2.576730e+05  2.576730e+05  2.576730e+05  2.576730e+05
2.576730e+05
mean   1.041246e-16  1.003743e-17 -1.698643e-17  1.323618e-16
2.144260e-16
std    1.000002e+00  1.000002e+00  1.000002e+00  1.000002e+00
1.000002e+00
min   -1.488066e+00 -2.086799e-01 -1.653309e-01 -5.691122e-01 -
1.756311e+00
25%   -8.297971e-01 -2.086786e-01 -1.653309e-01 -5.689202e-01 -
1.151363e+00
50%   -1.715278e-01 -2.079627e-01 -1.474715e-01 -5.506790e-01
7.220262e-01
75%    7.788946e-01 -9.389195e-02 -7.603381e-02  2.104601e-01
7.220262e-01
max    2.095433e+00  9.834346e+00  9.822219e+01  5.667466e+00
7.317834e-01

                dttl         dloss        sinpkt        dinpkt
swin  \
count  2.576730e+05  2.576730e+05  2.576730e+05  2.576730e+05
2.576730e+05
mean  -8.824117e-17 -9.044720e-18 -9.706529e-18  1.103015e-18
3.176682e-17
std    1.000002e+00  1.000002e+00  1.000002e+00  1.000002e+00
1.000002e+00
min   -7.516275e-01 -1.255759e-01 -1.317946e-01 -9.041248e-02 -
```

```
9.559264e-01
25%   -7.516275e-01 -1.255759e-01 -1.317935e-01 -9.041248e-02 -
9.559264e-01
50%   -4.944485e-01 -1.255759e-01 -1.317395e-01 -9.040608e-02 -
9.559264e-01
75%    1.483170e+00 -8.833340e-02 -1.234020e-01 -3.882522e-02
1.046159e+00
max    1.500906e+00  1.024216e+02  1.205685e+01  5.268544e+01
1.046159e+00

       ...  ct_state_ttl  is_sm_ips_ports       dur_log     spkts_log
\
count  ...  2.576730e+05     2.576730e+05  2.576730e+05  2.576730e+05

mean   ... -5.206229e-17    -5.647435e-17 -1.407447e-16 -6.776922e-16

std    ...  1.000002e+00     1.000002e+00  1.000002e+00  1.000002e+00

min    ... -1.335262e+00    -1.203354e-01 -5.363499e-01 -1.187933e+00

25%    ... -3.275002e-01    -1.203354e-01 -5.363372e-01 -8.275368e-01

50%    ... -3.275002e-01    -1.203354e-01 -5.295495e-01 -3.734912e-01

75%    ...  6.802617e-01    -1.203354e-01  2.942178e-01  4.758118e-01

max    ...  4.711309e+00     8.310109e+00  6.001730e+00  6.438269e+00


          dpkts_log    sbytes_log    dbytes_log  trans_depth_log  \
count  2.576730e+05  2.576730e+05  2.576730e+05     2.576730e+05
mean  -8.846178e-17 -4.579717e-16 -4.765023e-17    -1.411859e-17
std    1.000002e+00  1.000002e+00  1.000002e+00     1.000002e+00
min   -9.181681e-01 -1.873825e+00 -9.913824e-01    -3.288177e-01
25%   -9.181681e-01 -9.446923e-01 -9.913824e-01    -3.288177e-01
50%   -2.014077e-01 -1.556010e-02  3.330670e-01    -3.288177e-01
75%    6.462750e-01  5.606846e-01  7.883926e-01    -3.288177e-01
max    5.154182e+00  6.199943e+00  3.221536e+00     2.433393e+01

       response_body_len_log  state_encoded
count           2.576730e+05   2.576730e+05
mean            4.941506e-17  -2.717828e-16
std             1.000002e+00   1.000002e+00
min            -2.523453e-01  -4.892246e+00
25%            -2.523453e-01  -3.847878e-01
50%            -2.523453e-01  -3.847878e-01
75%            -2.523453e-01   7.420767e-01
max             8.467074e+00   6.376400e+00

[8 rows x 27 columns]
```

Standardize remaining numerical columns and drop original skewed features: This cell refines the standardization process. It identifies all numerical columns (including log-transformed ones), excludes the original highly skewed columns and the 'label', and then applies `StandardScaler`. Finally, it drops the original highly skewed columns that were replaced by their log-transformed versions.

```
from sklearn.preprocessing import StandardScaler
import numpy as np
import pandas as pd

# Identify numerical columns, including the new log-transformed ones
numerical_cols =
data.select_dtypes(include=np.number).columns.tolist()

# Define the original skewed columns
original_skewed_cols = ['dur', 'spkts', 'dpkts', 'sbytes', 'dbytes',
'trans_depth', 'response_body_len']

# Identify numerical columns to scale (all numerical except the
original skewed ones and 'label')
numerical_cols_to_scale = [col for col in numerical_cols if col not in
original_skewed_cols and col != 'label']

# Initialize the StandardScaler
scaler = StandardScaler()

# Apply standardization to the selected numerical columns
data[numerical_cols_to_scale] =
scaler.fit_transform(data[numerical_cols_to_scale])

# Identify which of the original skewed columns are still in the
DataFrame
existing_original_skewed_cols = [col for col in original_skewed_cols
if col in data.columns]

# Drop the original highly skewed columns that still exist
if existing_original_skewed_cols:
    data = data.drop(columns=existing_original_skewed_cols)

print("Data after standardizing all relevant numerical columns
(excluding 'label') and dropping original skewed columns:")
display(data.head())
```

```
Data after standardizing all relevant numerical columns (excluding
'label') and dropping original skewed columns:

{"type":"dataframe"}
```

Display the first few rows of the processed data: This cell simply displays the head of the DataFrame `data` after the preprocessing steps. This allows for a quick visual check of the data's structure and values.

```
data.head()
```

{"type":"dataframe","variable_name":"data"}

Apply One-Hot Encoding to the 'state' column: This cell performs One-Hot Encoding specifically on the 'state' column. It uses `OneHotEncoder` to create new binary columns for each unique state value. The original 'state' column is then dropped, and the new one-hot encoded columns are concatenated to the DataFrame.

```
from sklearn.preprocessing import OneHotEncoder

# Initialize OneHotEncoder
# handle_unknown='ignore' will ignore categories not seen during fit,
useful for consistent test sets
onehot_encoder = OneHotEncoder(sparse_output=False,
handle_unknown='ignore')

# Fit and transform the 'state' column
state_encoded = onehot_encoder.fit_transform(data[['state']])

# Create a DataFrame from the one-hot encoded array
state_encoded_df = pd.DataFrame(state_encoded,
columns=onehot_encoder.get_feature_names_out(['state']))

# Concatenate the new one-hot encoded columns to the original
DataFrame
data = pd.concat([data, state_encoded_df], axis=1)

# Drop the original 'state' column
data = data.drop('state', axis=1)

print("Data after one-hot encoding 'state' column:")
display(data.head())
```

```
Data after one-hot encoding 'state' column:
```

{"type":"dataframe"}

Construct feature and label arrays: This cell separates the DataFrame into features (`X`) and the target variable (`y`). It creates a list of feature columns by excluding the 'label' column and any original attack category columns. `X` contains the feature values as a NumPy array, and `y` contains the label values as a NumPy array.

```
# Construct X/y with encoded features and standardized numerical
features
# Exclude 'label' and any potentially lingering original categorical
columns not used for OHE
feature_cols = [col for col in data.columns if col not in ['label',
'attack_cat', 'attack_cat_encoded']] # 'state' is now one-hot encoded
X = data[feature_cols].values
```

```
y = data['label'].values # Ensure 'label' is not standardized

print("Feature array shape:", X.shape)
print("Label array length:", y.shape)
```

```
Feature array shape: (257673, 35)
Label array length: (257673,)
```

Split data into training and testing sets: This cell splits the feature array (`X`) and label array (`y`) into training and testing sets using `train_test_split`. `test_size=0.3` allocates 30% of the data for testing, `random_state=42` ensures reproducibility, and `stratify=y` maintains the same class distribution in both the training and testing sets as in the original data.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42, stratify=y)
print("Train set:", X_train.shape, "Test set:", X_test.shape)
```

```
Train set: (180371, 35) Test set: (77302, 35)
```

Display label distribution in train and test sets: This cell uses `np.bincount()` to count the number of instances for each class (0 and 1) in the training and testing label sets. This confirms that the stratification during the split was successful in preserving the class distribution.

```
import numpy as np

print("Train label distribution:", np.bincount(y_train.astype(int)))
print("Test label distribution:", np.bincount(y_test.astype(int)))
```

```
Train label distribution: [ 65100 115271]
Test label distribution: [27900 49402]
```

Define and compile the neural network model: This cell defines the architecture of a sequential neural network model using TensorFlow/Keras.

- It's a `Sequential` model, meaning layers are stacked linearly.
- It has two hidden `Dense` layers with ReLU activation, followed by `BatchNormalization` and `Dropout` for regularization.
- The output layer is a `Dense` layer with one unit and a `sigmoid` activation function, suitable for binary classification.
- The model is compiled with the `adam` optimizer, `binary_crossentropy` loss function, and metrics like `accuracy`, `Precision`, `Recall`, and `AUC` to evaluate performance during training.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization
import tensorflow as tf

# Define the model architecture
```

```python
model = Sequential([
    # Input layer and first hidden layer
    Dense(128, activation='relu', input_shape=(X_train.shape[1],)),
    BatchNormalization(), # Add BatchNormalization
    Dropout(0.3), # Add Dropout for regularization

    # Second hidden layer
    Dense(64, activation='relu'),
    BatchNormalization(), # Add BatchNormalization
    Dropout(0.3), # Add Dropout for regularization

    # Output layer for binary classification
    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam',
              loss='binary_crossentropy', # Suitable for binary classification
              metrics=['accuracy', tf.keras.metrics.Precision(), tf.keras.metrics.Recall(), tf.keras.metrics.AUC()]) # Add relevant metrics

model.summary()
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Model: "sequential"
```

```
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                    ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ dense (Dense)                   │ (None, 128)            │         4,608 │
├─────────────────────────────────┼────────────────────────┼───────────────┤
│ batch_normalization             │ (None, 128)            │           512 │
│ (BatchNormalization)            │                        │               │
├─────────────────────────────────┼────────────────────────┼───────────────┤
```

```
| dropout (Dropout)                    | (None, 128)               |
0 |
├──────────────────────────────────────┼───────────────────────────┼──────
──────┤
| dense_1 (Dense)                      | (None, 64)                |
8,256 |
├──────────────────────────────────────┼───────────────────────────┼──────
──────┤
| batch_normalization_1                | (None, 64)                |
256 |
| (BatchNormalization)                 |                           |
|
├──────────────────────────────────────┼───────────────────────────┼──────
──────┤
| dropout_1 (Dropout)                  | (None, 64)                |
0 |
├──────────────────────────────────────┼───────────────────────────┼──────
──────┤
| dense_2 (Dense)                      | (None, 1)                 |
65 |
└──────────────────────────────────────┴───────────────────────────┴──────
──────┘

 Total params: 13,697 (53.50 KB)

 Trainable params: 13,313 (52.00 KB)

 Non-trainable params: 384 (1.50 KB)
```

Train the neural network model: This cell trains the compiled neural network model using the training data (`X_train`, `y_train`).

- `epochs=50`: Specifies the maximum number of training epochs.
- `batch_size=32`: Sets the number of samples per gradient update.
- `validation_split=0.2`: Uses 20% of the training data for validation during training.
- `class_weight=class_weight_dict`: Applies class weights to handle the imbalance in the target variable, giving more importance to the minority class.
- `callbacks=[early_stopping]`: Includes the `EarlyStopping` callback to prevent overfitting by stopping training when the validation loss stops improving.

```python
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.utils.class_weight import compute_class_weight
import numpy as np

# Calculate class weights for handling imbalance
class_weights = compute_class_weight('balanced',
classes=np.unique(y_train), y=y_train)
class_weight_dict = dict(enumerate(class_weights))
print("Class weights:", class_weight_dict)
```

```
# Define Early Stopping callback
early_stopping = EarlyStopping(monitor='val_loss', # Monitor
validation loss
                               patience=5,         # Stop after 5
epochs if val_loss doesn't improve
                               restore_best_weights=True) # Restore
best weights found during training

# Train the model
history = model.fit(X_train, y_train,
                    epochs=50,          # You can adjust the number of
epochs
                    batch_size=32,     # You can adjust the batch size
                    validation_split=0.2, # Use 20% of training data
for validation
                    class_weight=class_weight_dict, # Add class
weights
                    callbacks=[early_stopping]) # Add Early Stopping
callback

print("Model training finished.")
```

```
Class weights: {0: np.float64(1.3853379416282643), 1:
np.float64(0.7823780482515117)}
Epoch 1/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 23s 4ms/step - accuracy: 0.8830 - auc:
0.9559 - loss: 0.2595 - precision: 0.9267 - recall: 0.8874 -
val_accuracy: 0.9170 - val_auc: 0.9817 - val_loss: 0.1650 -
val_precision: 0.9668 - val_recall: 0.9001
Epoch 2/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - accuracy: 0.9070 - auc:
0.9754 - loss: 0.1946 - precision: 0.9514 - recall: 0.9009 -
val_accuracy: 0.9146 - val_auc: 0.9833 - val_loss: 0.1653 -
val_precision: 0.9757 - val_recall: 0.8877
Epoch 3/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - accuracy: 0.9097 - auc:
0.9774 - loss: 0.1861 - precision: 0.9546 - recall: 0.9017 -
val_accuracy: 0.9108 - val_auc: 0.9837 - val_loss: 0.1670 -
val_precision: 0.9779 - val_recall: 0.8794
Epoch 4/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - accuracy: 0.9133 - auc:
0.9786 - loss: 0.1802 - precision: 0.9592 - recall: 0.9027 -
val_accuracy: 0.9249 - val_auc: 0.9849 - val_loss: 0.1507 -
val_precision: 0.9715 - val_recall: 0.9083
Epoch 5/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - accuracy: 0.9158 - auc:
0.9794 - loss: 0.1774 - precision: 0.9611 - recall: 0.9053 -
val_accuracy: 0.9227 - val_auc: 0.9854 - val_loss: 0.1528 -
val_precision: 0.9755 - val_recall: 0.9008
```

```
Epoch 6/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - accuracy: 0.9197 - auc:
0.9807 - loss: 0.1720 - precision: 0.9616 - recall: 0.9111 -
val_accuracy: 0.9235 - val_auc: 0.9862 - val_loss: 0.1504 -
val_precision: 0.9783 - val_recall: 0.8994
Epoch 7/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - accuracy: 0.9223 - auc:
0.9816 - loss: 0.1687 - precision: 0.9623 - recall: 0.9145 -
val_accuracy: 0.9327 - val_auc: 0.9880 - val_loss: 0.1374 -
val_precision: 0.9694 - val_recall: 0.9232
Epoch 8/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - accuracy: 0.9252 - auc:
0.9831 - loss: 0.1628 - precision: 0.9618 - recall: 0.9196 -
val_accuracy: 0.9381 - val_auc: 0.9889 - val_loss: 0.1366 -
val_precision: 0.9788 - val_recall: 0.9225
Epoch 9/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - accuracy: 0.9291 - auc:
0.9838 - loss: 0.1590 - precision: 0.9657 - recall: 0.9219 -
val_accuracy: 0.9269 - val_auc: 0.9885 - val_loss: 0.1474 -
val_precision: 0.9810 - val_recall: 0.9024
Epoch 10/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - accuracy: 0.9303 - auc:
0.9843 - loss: 0.1572 - precision: 0.9660 - recall: 0.9235 -
val_accuracy: 0.9449 - val_auc: 0.9901 - val_loss: 0.1274 -
val_precision: 0.9752 - val_recall: 0.9371
Epoch 11/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - accuracy: 0.9329 - auc:
0.9848 - loss: 0.1549 - precision: 0.9666 - recall: 0.9273 -
val_accuracy: 0.9466 - val_auc: 0.9906 - val_loss: 0.1231 -
val_precision: 0.9781 - val_recall: 0.9368
Epoch 12/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 22s 5ms/step - accuracy: 0.9339 - auc:
0.9850 - loss: 0.1537 - precision: 0.9660 - recall: 0.9297 -
val_accuracy: 0.9428 - val_auc: 0.9907 - val_loss: 0.1283 -
val_precision: 0.9816 - val_recall: 0.9272
Epoch 13/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - accuracy: 0.9336 - auc:
0.9853 - loss: 0.1520 - precision: 0.9669 - recall: 0.9280 -
val_accuracy: 0.9436 - val_auc: 0.9914 - val_loss: 0.1282 -
val_precision: 0.9848 - val_recall: 0.9256
Epoch 14/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - accuracy: 0.9381 - auc:
0.9867 - loss: 0.1445 - precision: 0.9700 - recall: 0.9321 -
val_accuracy: 0.9425 - val_auc: 0.9909 - val_loss: 0.1332 -
val_precision: 0.9819 - val_recall: 0.9264
Epoch 15/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - accuracy: 0.9390 - auc:
0.9868 - loss: 0.1438 - precision: 0.9696 - recall: 0.9343 -
val_accuracy: 0.9518 - val_auc: 0.9920 - val_loss: 0.1229 -
```

```
val_precision: 0.9798 - val_recall: 0.9435
Epoch 16/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - accuracy: 0.9413 - auc:
0.9875 - loss: 0.1402 - precision: 0.9705 - recall: 0.9368 -
val_accuracy: 0.9547 - val_auc: 0.9923 - val_loss: 0.1182 -
val_precision: 0.9791 - val_recall: 0.9490
Epoch 17/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 21s 5ms/step - accuracy: 0.9384 - auc:
0.9872 - loss: 0.1425 - precision: 0.9684 - recall: 0.9339 -
val_accuracy: 0.9571 - val_auc: 0.9920 - val_loss: 0.1213 -
val_precision: 0.9739 - val_recall: 0.9580
Epoch 18/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 18s 4ms/step - accuracy: 0.9414 - auc:
0.9875 - loss: 0.1395 - precision: 0.9707 - recall: 0.9369 -
val_accuracy: 0.9557 - val_auc: 0.9926 - val_loss: 0.1120 -
val_precision: 0.9796 - val_recall: 0.9500
Epoch 19/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 22s 4ms/step - accuracy: 0.9424 - auc:
0.9882 - loss: 0.1356 - precision: 0.9705 - recall: 0.9384 -
val_accuracy: 0.9546 - val_auc: 0.9922 - val_loss: 0.1104 -
val_precision: 0.9788 - val_recall: 0.9490
Epoch 20/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - accuracy: 0.9427 - auc:
0.9886 - loss: 0.1344 - precision: 0.9714 - recall: 0.9383 -
val_accuracy: 0.9571 - val_auc: 0.9928 - val_loss: 0.1065 -
val_precision: 0.9810 - val_recall: 0.9507
Epoch 21/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - accuracy: 0.9413 - auc:
0.9880 - loss: 0.1375 - precision: 0.9709 - recall: 0.9362 -
val_accuracy: 0.9559 - val_auc: 0.9926 - val_loss: 0.1085 -
val_precision: 0.9793 - val_recall: 0.9507
Epoch 22/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - accuracy: 0.9439 - auc:
0.9888 - loss: 0.1327 - precision: 0.9716 - recall: 0.9398 -
val_accuracy: 0.9559 - val_auc: 0.9929 - val_loss: 0.1087 -
val_precision: 0.9808 - val_recall: 0.9491
Epoch 23/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - accuracy: 0.9440 - auc:
0.9888 - loss: 0.1328 - precision: 0.9735 - recall: 0.9381 -
val_accuracy: 0.9568 - val_auc: 0.9927 - val_loss: 0.1054 -
val_precision: 0.9813 - val_recall: 0.9500
Epoch 24/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 18s 4ms/step - accuracy: 0.9438 - auc:
0.9887 - loss: 0.1335 - precision: 0.9723 - recall: 0.9388 -
val_accuracy: 0.9541 - val_auc: 0.9925 - val_loss: 0.1100 -
val_precision: 0.9826 - val_recall: 0.9443
Epoch 25/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - accuracy: 0.9451 - auc:
0.9891 - loss: 0.1305 - precision: 0.9729 - recall: 0.9404 -
```

```
val_accuracy: 0.9547 - val_auc: 0.9929 - val_loss: 0.1125 -
val_precision: 0.9844 - val_recall: 0.9435
Epoch 26/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 18s 4ms/step - accuracy: 0.9454 - auc:
0.9891 - loss: 0.1304 - precision: 0.9733 - recall: 0.9407 -
val_accuracy: 0.9560 - val_auc: 0.9929 - val_loss: 0.1095 -
val_precision: 0.9825 - val_recall: 0.9476
Epoch 27/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 20s 4ms/step - accuracy: 0.9439 - auc:
0.9885 - loss: 0.1345 - precision: 0.9717 - recall: 0.9395 -
val_accuracy: 0.9551 - val_auc: 0.9926 - val_loss: 0.1113 -
val_precision: 0.9832 - val_recall: 0.9454
Epoch 28/50
4510/4510 ━━━━━━━━━━━━━━━━━━━━ 19s 4ms/step - accuracy: 0.9461 - auc:
0.9893 - loss: 0.1294 - precision: 0.9736 - recall: 0.9412 -
val_accuracy: 0.9547 - val_auc: 0.9930 - val_loss: 0.1069 -
val_precision: 0.9799 - val_recall: 0.9480
Model training finished.
```

Evaluate the model on the test set: This cell evaluates the trained model's performance on the unseen test set (`X_test`, `y_test`) using the `.evaluate()` method. It reports the test loss and the metrics specified during compilation (accuracy, precision, recall, and AUC).

```
# Evaluate the model on the test set
loss, accuracy, precision, recall, auc = model.evaluate(X_test,
y_test, verbose=0)

print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {accuracy:.4f}")
print(f"Test Precision: {precision:.4f}")
print(f"Test Recall: {recall:.4f}")
print(f"Test AUC: {auc:.4f}")

Test Loss: 0.1055
Test Accuracy: 0.9569
Test Precision: 0.9812
Test Recall: 0.9508
Test AUC: 0.9927
```

Generate predictions on the test set: This cell uses the trained model to predict the probability of the positive class for each sample in the test set using the `.predict()` method. It then converts these probabilities into binary class labels (0 or 1) by applying a threshold of 0.5.

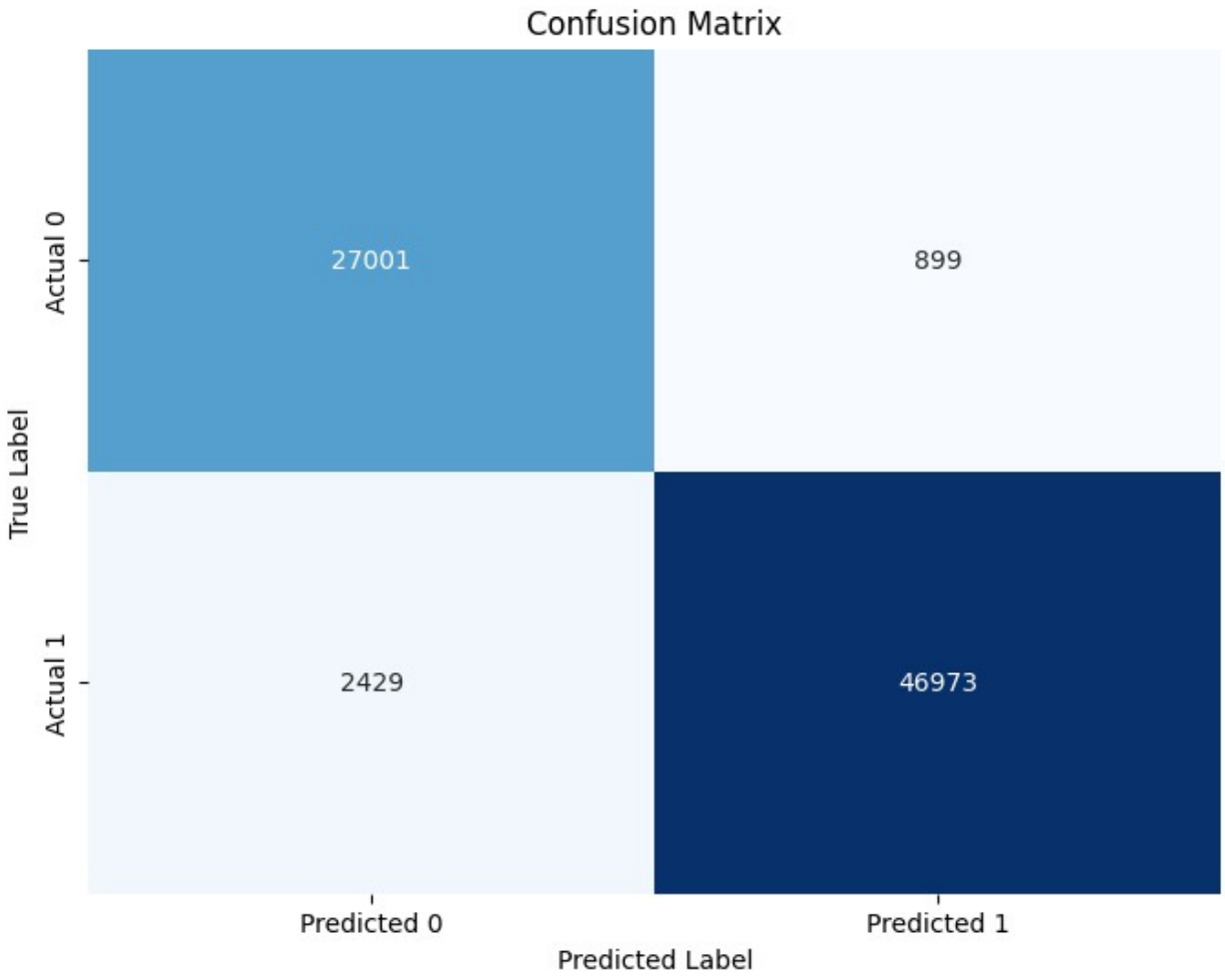
```
import numpy as np

# Get predicted probabilities on the test set
y_pred_prob = model.predict(X_test)

# Convert probabilities to class labels (0 or 1) using a threshold
```

```
(e.g., 0.5)
y_pred = (y_pred_prob > 0.5).astype(int)

print("Sample predicted probabilities:", y_pred_prob[:10].flatten())
print("Sample predicted labels:", y_pred[:10].flatten())
```

```
2416/2416 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step
Sample predicted probabilities: [1.5799781e-06 1.8462493e-05
5.1504833e-07 1.0348273e-04 9.9990392e-01
 2.0792377e-06 1.7203481e-01 9.8439761e-02 9.9999911e-01 9.7559398e-
01]
Sample predicted labels: [0 0 0 0 1 0 0 0 1 1]
```

Generate and visualize the confusion matrix: This cell calculates the confusion matrix using `confusion_matrix` from scikit-learn. It then visualizes the confusion matrix as a heatmap using `seaborn.heatmap()`. The confusion matrix helps assess the performance of a classification model by showing the counts of true positive, true negative, false positive, and false negative predictions.

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Generate confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
cbar=False,
            xticklabels=['Predicted 0', 'Predicted 1'],
            yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```



Generate and display the classification report: This cell generates a detailed classification report using `classification_report` from scikit-learn. The classification report includes key metrics like precision, recall, F1-score, and support for each class, as well as overall accuracy. This provides a comprehensive evaluation of the model's performance.

```
from sklearn.metrics import classification_report

# Generate classification report
print("Classification Report:")
print(classification_report(y_test, y_pred))
```

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

           0       0.92      0.97      0.94     27900
           1       0.98      0.95      0.97     49402

    accuracy                           0.96     77302
   macro avg       0.95      0.96      0.95     77302
```

```
weighted avg       0.96      0.96      0.96     77302
```

```
import joblib

# Save the trained model
model_filename = 'trained_model.pkl'
joblib.dump(model, model_filename)

print(f"Model saved as {model_filename}")
```

```
Model saved as trained_model.pkl
```

```
from google.colab import files

# Download the saved model file
files.download(model_filename)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.Javascript object>
```