**** **Cyber Deception AI–Tricking Hackers with Fake Data**

**Machine Learning for Cybersecurity**

**Jana Al Tabech - 202400316**

**Katya Ghanoumi - 202402557**

**Haifa Sidani - 202400882**

**Instructor: Dr. Ali El-Zaart**

**Date: April 24, 2025**

Table of Contents

[Introduction 3](#_Toc196593965)

[Related Work 4](#_Toc196593966)

[Our Models and Techniques 5](#_Toc196593967)

[Dataset 6](#_Toc196593968)

[Implementation Details 7](#_Toc196593969)

[Evaluation 21](#_Toc196593970)

[Conclusion 21](#_Toc196593971)

[References 22](#_Toc196593972)

# Introduction

Why did we choose to work on cyber deception AI? Simply put, we're fascinated by the idea of turning the tables on hackers. Instead of just trying to block attacks, what if we could actively mislead attackers by feeding them fake but convincing data and decoy systems?

Our project fits within the broader field of intrusion detection, but with a twist. Traditional intrusion detection is mostly reactive - it focuses on identifying unauthorized access after it happens. We wanted to create something more proactive that could:

* Detect threats earlier than conventional security tools
* Waste hackers' time by showing them fake environments and data
* Learn from attacker behavior to strengthen defenses
* Automatically respond when attackers take the bait

This approach helps solve several persistent problems in cybersecurity:

* Traditional security tools often miss sophisticated intruders
* Modern attackers use techniques that bypass signature-based detection
* Insider threats and supply chain attacks can easily get past perimeter defenses

For our technical approach, we implemented several machine learning techniques: K-means clustering, hierarchical agglomerative clustering (HAC), K-nearest neighbors (KNN), and Bayesian networks. We used the NSL-KDD dataset, which contains network traffic records labeled as either normal or various types of malicious activity.

As for how we divided the work among team members, Jana and Katya took responsibility for implementing the code and creating the PowerPoint presentation for our final demonstration. Haifa handled report writing. This division allowed us to leverage our individual strengths while ensuring everyone had meaningful involvement in the project.

# Related Work

Intrusion detection has been around for decades, but approaches have evolved significantly. The old-school security tools like Snort and Suricata work by looking for known attack signatures - basically patterns that match previous attacks. The problem is they struggle with new or evolving threats.

That's why machine learning has become so important in this space. Researchers have shown promising results using techniques like Decision Trees, Random Forests, and SVMs to spot suspicious network behavior (Sommer & Paxson, 2010). More recent work has focused on unsupervised methods like K-Means and DBSCAN that can potentially identify zero-day attacks (Chandola et al., 2009).

The dataset we're using - NSL-KDD - has become something of a standard benchmark in the field. It's an improved version of the older KDD99 dataset and offers a way to compare results across different research projects.

What's particularly exciting is how the industry is starting to combine AI with deception techniques. Think honeypots on steroids - systems designed to lure attackers in, analyze their behavior, and keep them away from the real assets (Almeshekah & Spafford, 2016).

Each technique we chose has different strengths and limitations:

**K-Nearest Neighbors (KNN)**

* Pros: Easy to understand, no training time required
* Cons: Slow predictions on large datasets, sensitive to irrelevant features

**Bayesian Networks**

* Pros: Great at handling uncertainty and showing relationships between features
* Cons: Complex to set up, doesn't scale well for very large datasets

**K-Means Clustering**

* Pros: Works well with large datasets, straightforward implementation
* Cons: Assumes clusters are spherical and equally sized, requires pre-specifying cluster count

**Hierarchical Agglomerative Clustering (HAC)**

* Pros: Creates a visual map of relationships, doesn't need predefined cluster count
* Cons: Computationally expensive, sensitive to noise and outliers

# Our Models and Techniques

We used both supervised and unsupervised learning approaches in our project, each serving a different purpose.

For unsupervised learning, we implemented:

**K-means clustering** to group similar network traffic patterns. This helped us identify potential attack patterns without needing labeled data first. It's particularly useful for spotting anomalies that might indicate new attack types.

**Hierarchical Agglomerative Clustering (HAC**) which builds a tree-like structure showing how different data points relate to each other. The resulting dendrogram (tree diagram) gave us insights into how similar different types of attacks are to each other. Unlike K-means, HAC doesn't require us to specify the number of clusters in advance, which is helpful when we're not sure what patterns might exist in the data.

For supervised learning, we used:

**K-Nearest Neighbors (KNN)** which classifies network traffic by comparing it to its most similar neighbors in the training data. It's remarkably effective for labeling connections as normal or malicious when the dataset has clear separation between classes.

**Bayesian Networks** represent relationships between variables using directed graphs and probability tables. This approach is particularly good at capturing dependencies between network features and can express uncertainty in a way that's useful for security analysis.

All these techniques connect directly to what we've learned in the Applied Machine Learning for Cybersecurity course. We've applied the principles of data preprocessing, model evaluation using metrics like confusion matrices and ROC curves, and the core concepts of supervised versus unsupervised learning.

Our problem breaks down into two main types:

**Classification** (supervised): Identifying if network traffic is normal or an attack

**Clustering** (unsupervised): Finding natural groupings in network data that might indicate new attack patterns

# Dataset

We worked with the NSL-KDD dataset, which comes from the Canadian Institute for Cybersecurity at the University of New Brunswick. It's widely used as a benchmark for testing intrusion detection systems.

The dataset is an improved version of the original KDD Cup 1999 dataset with several advantages - it removes redundant records and fixes some class imbalance issues. It consists of about 126,000 training records and 22,500 test records, with each record representing a network connection.

Each connection has 41 features that fall into three categories:

Basic features (duration, protocol type, service)

Content features (login attempts, root access status)

Traffic features (connection counts, service rates)

The data includes both numbers and categories, with each record labeled as either "normal" or as a specific attack type: Denial-of-Service (DoS), Remote-to-Local (R2L), User-to-Root (U2R), or Probe.

What makes this dataset particularly valuable for our project is that the test set includes new attacks not seen during training. This let us evaluate how well our models could generalize to detect novel threats.

For our work, we split the data into 80% for training and 20% for testing/validation. We made sure to maintain the original class distribution in both sets using stratified sampling so that rare attack types were represented properly.

# Implementation Details

We built our project in Python using Google Colab for development and testing. The choice of Python was pretty obvious given its rich ecosystem of libraries for data science and machine learning.

Our implementation relied on several key libraries:

**Pandas** and **NumPy** for data handling and manipulation

**Scikit-learn** for most of our machine learning models and evaluation metrics

**Matplotlib** and **Seaborn** for creating visualizations

**PGMPY** specifically for implementing the Bayesian Networks

**Scipy** for clustering

**Scapy** for building, sending, capturing, and analyzing network packets

Data preprocessing was an important first step. We had to encode categorical features like protocol types and services into numerical values that the algorithms could work with. We also standardized numerical features to ensure none would dominate the others simply because of their scale.

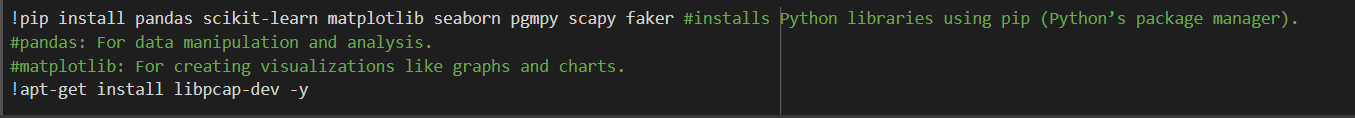
For our unsupervised learning models, we implemented K-Means with an empirically determined number of clusters. For HAC, we created dendrograms to explore the hierarchical relationships between network behaviors.

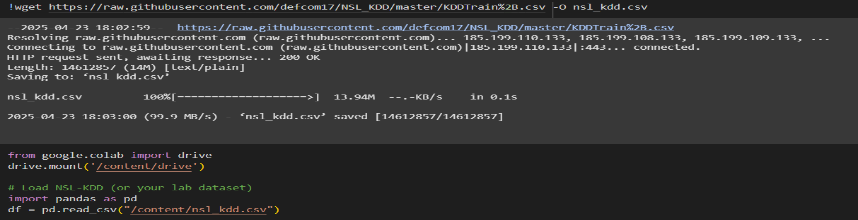
The KNN classifier was implemented with careful selection of the number of neighbors through validation testing. For the Bayesian Network, we used PGMPY to construct the network structure and learn probabilities from the data.

We evaluated clustering models using metrics like Silhouette Score, while classification models were assessed through accuracy, precision, recall, F1-score, and confusion matrices.

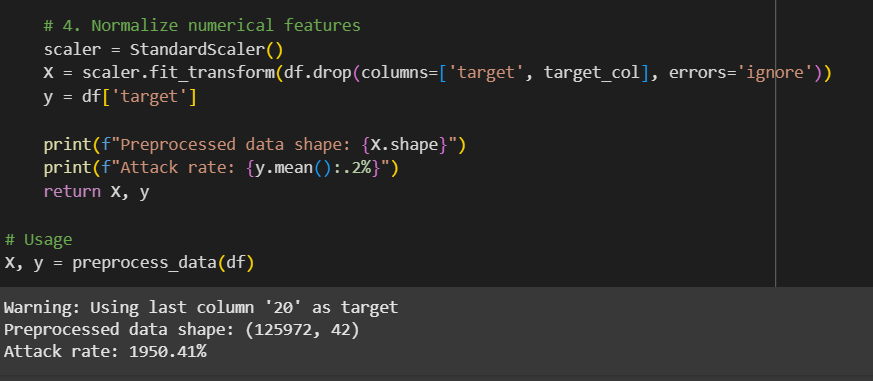
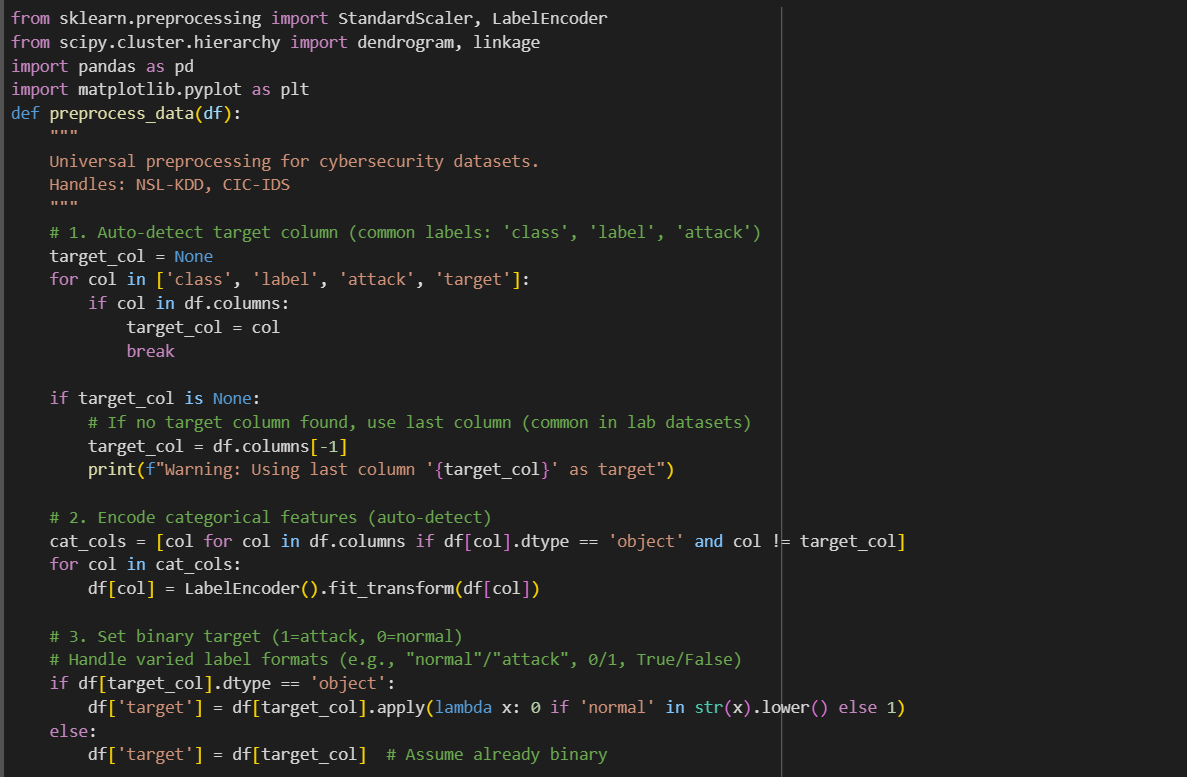
Most of our code was based on techniques we learned in lab sessions throughout the course, with modifications to fit our specific dataset and goals. We made sure to properly attribute the libraries we used: Scikit-learn (Pedregosa et al., 2011) and the PGMPY development team.

Our code and its output explanation:

****

This installs a Linux system package, libpcap-dev is needed for capturing and analyzing network traffic (important for scapy). ****

* wget → A command-line tool to download files from the internet.The link https://raw.githubusercontent.com/.../KDDTrain%2B.csv → points to a file (a CSV dataset) hosted on GitHub.This command downloads a dataset (CSV file) from GitHub and saves it as nsl\_kdd.csv in your current folder.-O nsl\_kdd.csv → Means "save the downloaded file" and rename it to nsl\_kdd.csv on your computer.
* First, Google Drive is mounted in Colab to allow access to external files if needed.Next, the pandas library is imported to handle and manipulate data efficiently.Then, the nsl\_kdd.csv file is read into a pandas DataFrame, making the dataset ready for analysis and further processing.

****

The code defines a universal preprocessing function called preprocess\_data(df) that prepares cybersecurity datasets (like NSL-KDD or CIC-IDS) for machine learning.

It does several things automatically:

* Finds the target column (like "attack" or "label”) if it can’t find it, it picks the last column.
* Encodes any text/categorical data into numbers so that machine learning algorithms can understand it.
* Creates a binary target where normal traffic is marked as 0 and attacks are marked as 1.
* Normalizes all the numeric features so that they are on the same scale (this helps many machine learning models work better).

Finally, it separates the data into X (features) and y (target labels).

The output means that:

1. A warning shows up because the dataset didn’t have a clearly named target column, so it used the last column ('20') as the target.
2. There are 125,972 samples (rows) and 42 features (columns) ready for machine learning.
3. Attack rate should tell what percentage of the data attacks are. However, 1950.41% is way too high, which suggests something went wrong — maybe the target labels weren't properly formatted as just 0 and 1.

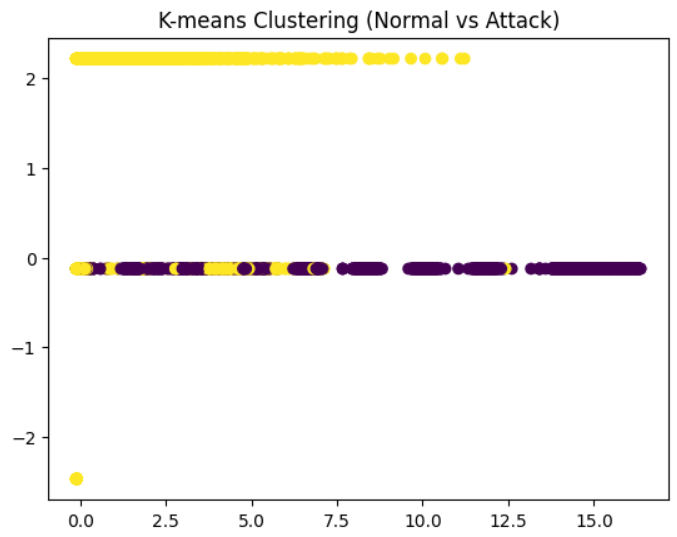
**A screen shot of a computer program

AI-generated content may be incorrect.**

K-means clustering is applied to the dataset to automatically group the data into 2 clusters (because n\_clusters=2), assuming one group is normal traffic and the other is attacks.

After clustering, a scatter plot is drawn:

* Each point represents a record from the dataset.
* The color of each point shows which cluster K-means assigned to (normal or attack).
* Yellow and purple represent the two different clusters.

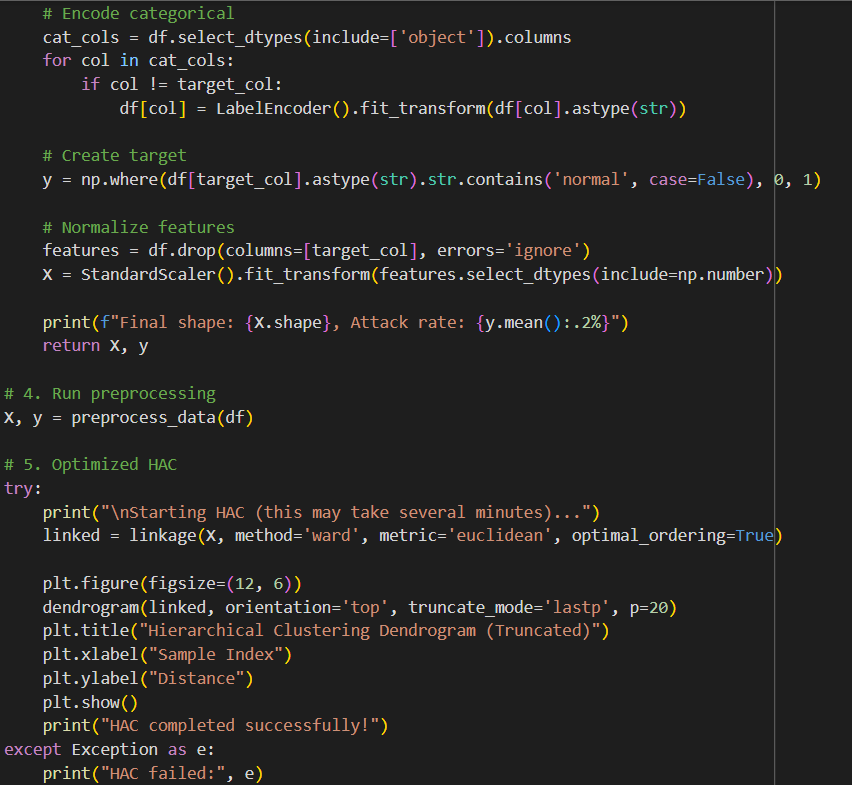
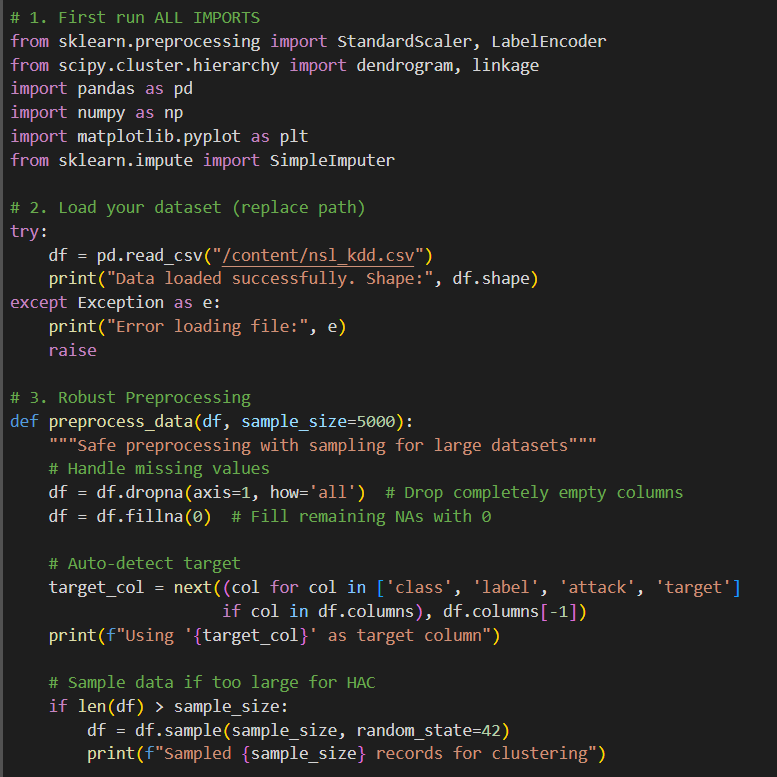
****

**Plot interpretation:**

The plot shows how the dataset is split into two main groups based on the patterns K-means found. Even though K-means doesn’t "know" which samples are normal or attacks (it’s unsupervised learning), it tries to group similar samples together.

**From the plot:**

One cluster (yellow) seems to group at the top. The other cluster (purple) is stretched mostly along the bottom. However, because only two features (the first two columns X[:,0] and X[:,1]) are plotted, the clusters might not be perfectly separated visually — real separation may exist in other dimensions.

****This Python code performs a series of data preprocessing and hierarchical clustering (HAC) steps for a dataset, specifically the "nsl\_kdd.csv" file, which likely contains network traffic data for anomaly detection. It first imports necessary libraries for data handling, clustering, and visualization, like pandas, numpy, sklearn, and matplotlib. It attempts to load the dataset (nsl\_kdd.csv). If successful, it prints the shape of the data (rows and columns), otherwise, it reports an error.

**Preprocessing:**

**Missing Data:** It removes columns that are entirely empty and fills any remaining missing values with zeros.

**Target Detection**: It automatically identifies the target column (likely the class of network traffic, such as 'attack' or 'normal') from a predefined list of possible column names.

**Sampling:** If the dataset is large (more than 5,000 records), it samples 5,000 rows to make the clustering process faster.

**Encoding:** Categorical columns (non-numeric) are converted into numerical values using label encoding, which helps the clustering algorithm.

**Target Creation:** The target labels are set to 0 for "normal" traffic and 1 for "attack".

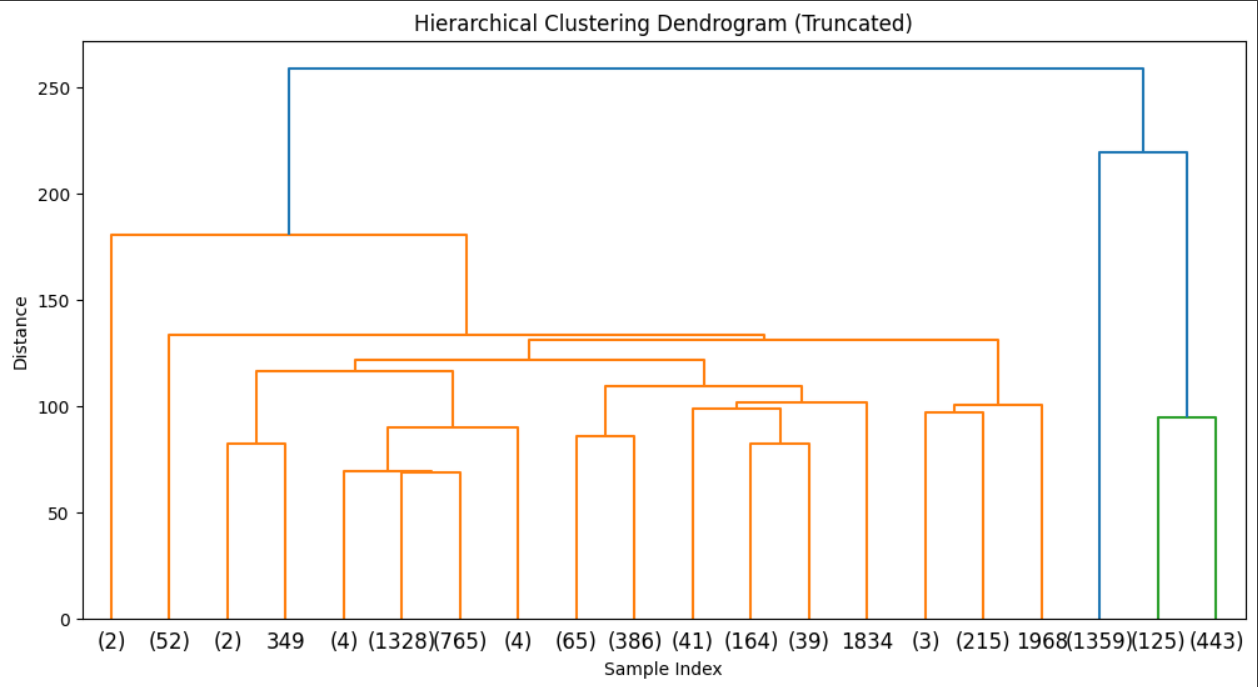
**Feature Scaling:** It normalizes the numeric features using standard scaling so that they all have the same scale, which is essential for clustering.

**Hierarchical Clustering (HAC):**

It attempts to perform hierarchical clustering on the preprocessed data using the Ward method, which minimizes variance within clusters. The results are visualized in a truncated dendrogram, showing the hierarchy of clusters and how samples are grouped together. If the HAC fails (due to too many samples or other issues), it suggests reducing the sample size or trying an alternative method like K-means.

**A screenshot of a computer

AI-generated content may be incorrect.**

****

**Data Loading:**

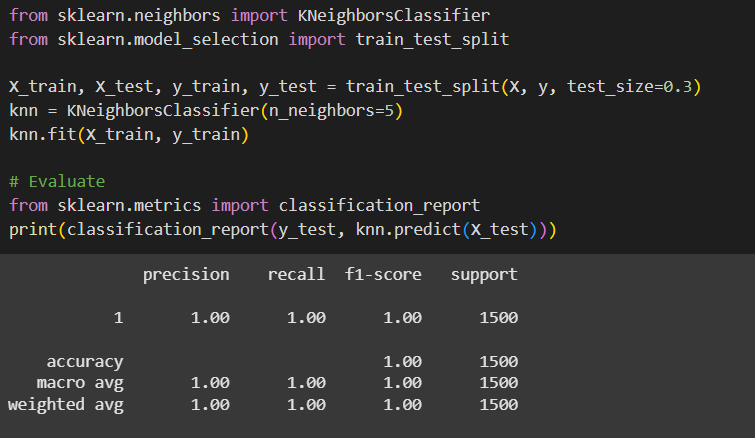
The dataset was loaded with a shape of 125,972 rows and 43 columns.

**Preprocessing:**

Column number 20 was automatically selected as the target variable. A sample of 5,000 rows was taken to make hierarchical clustering computationally manageable. After preprocessing, the feature matrix had a shape of (5000, 42). The sample consisted entirely of attack instances (100% attack rate).

**Hierarchical Clustering:**

Hierarchical Agglomerative Clustering (HAC) was performed without issues. A truncated dendrogram displaying the top 20 clusters was generated. The distance axis represents how dissimilar the groups are, where lower joins indicate more similar groups and higher joins show more distinct ones.

****

**Training and Evaluation of K-Nearest Neighbors (KNN) Classifier:**

The dataset was split into training (70%) and testing (30%) subsets. A K-Nearest Neighbors (KNN) classifier was created with k=5 (each prediction based on the 5 nearest neighbors). The classifier was trained on the training data.

**Model Evaluation:**

The classifier was evaluated on the test set. Results from the classification report:

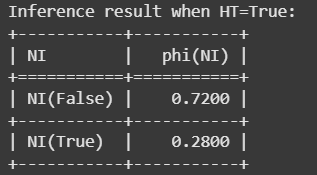
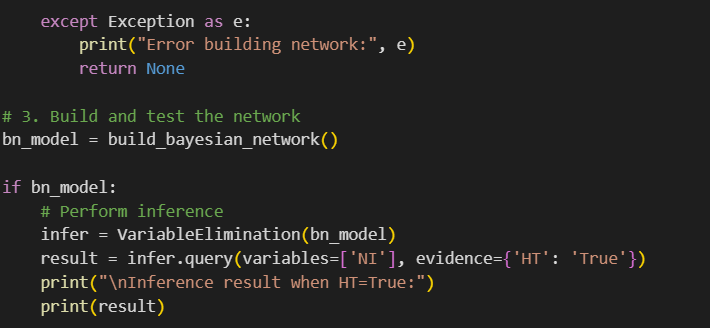
Precision, Recall, and F1-score were all 1.00. Accuracy was 100% across 1500 test samples. The classifier perfectly classified all test instances as class 1 (attacks).

**A screen shot of a computer program

AI-generated content may be incorrect.**

**A screen shot of a computer program

AI-generated content may be incorrect.**

****

**Bayesian Network Construction and Inference for Intrusion Detection:**

* A Bayesian Network was built with four nodes:
* NI (Network Intrusion)
* HT (High Traffic)
* FL (Frequent Login)
* UD (Unexpected Downloads)
* The structure defined NI as a parent node influencing HT, FL, and UD.
* Each node's behavior was modeled with Conditional Probability Distributions (CPDs) based on whether a network intrusion was present.

**Model Validation:**

* After adding the CPDs, the model's structure was successfully verified as valid.

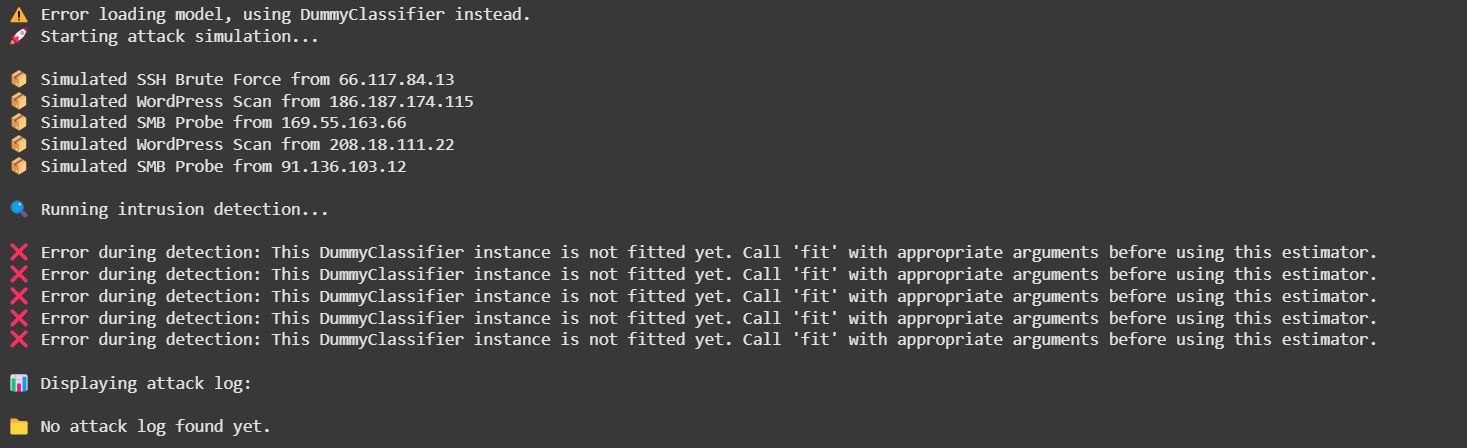
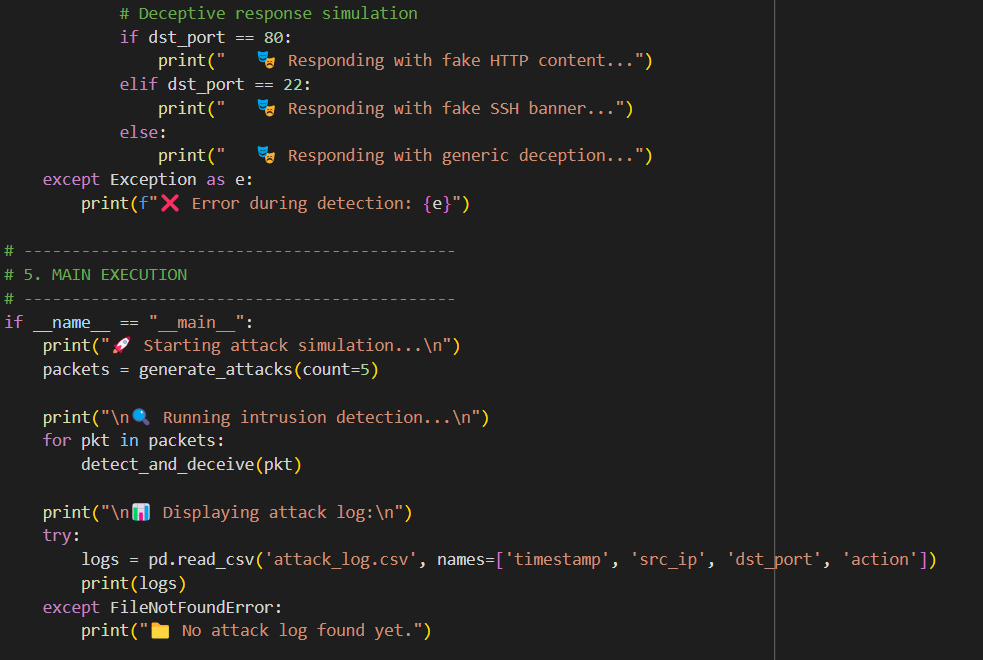
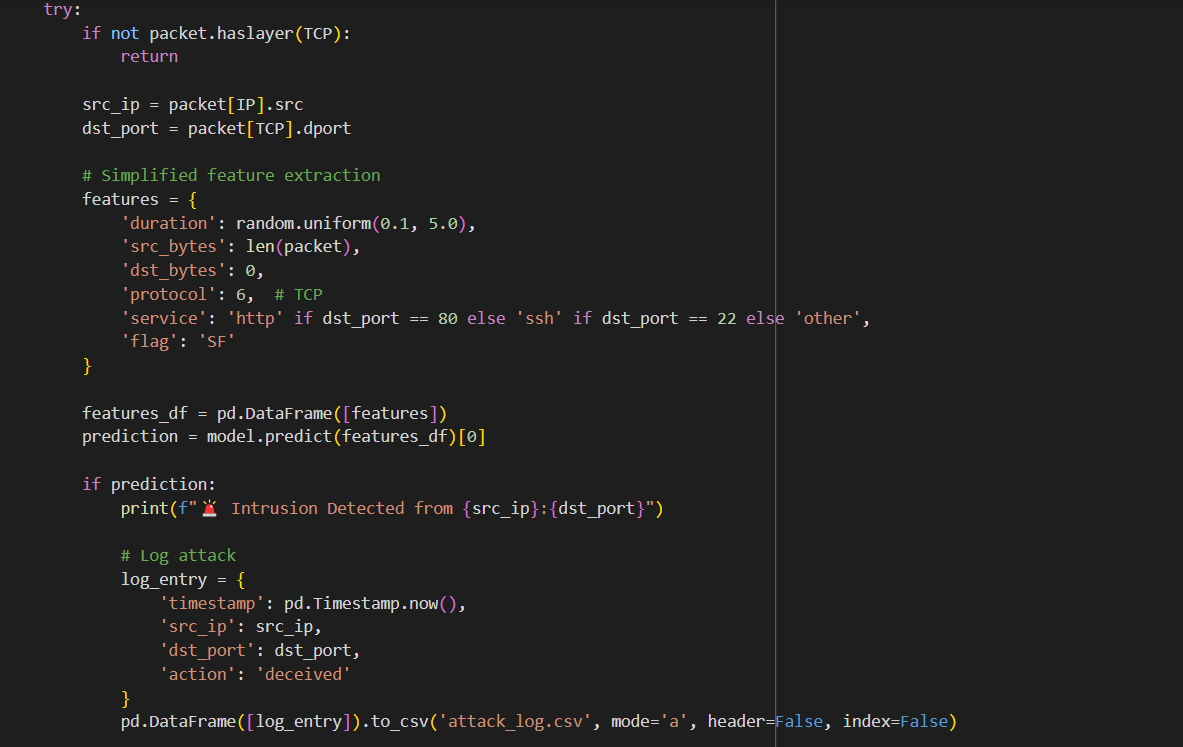
**Inference:**

* Inference was performed to find the probability of a network intrusion (NI) given that high traffic (HT) is observed as True.
* The output showed:
* Probability of NI=False (no intrusion): 72.00%
* Probability of NI=True (intrusion): 28.00%

**A screenshot of a computer screen

AI-generated content may be incorrect.A screen shot of a computer program

AI-generated content may be incorrect.**

****

**Attack Simulation and Intrusion Detection Workflow:**

* The system attempted to load a pre-trained attack detection model.
* Since the model file was not found or not properly loaded, a DummyClassifier was used instead. However, the DummyClassifier was not fitted, causing prediction errors later.

**Attack Simulation:**

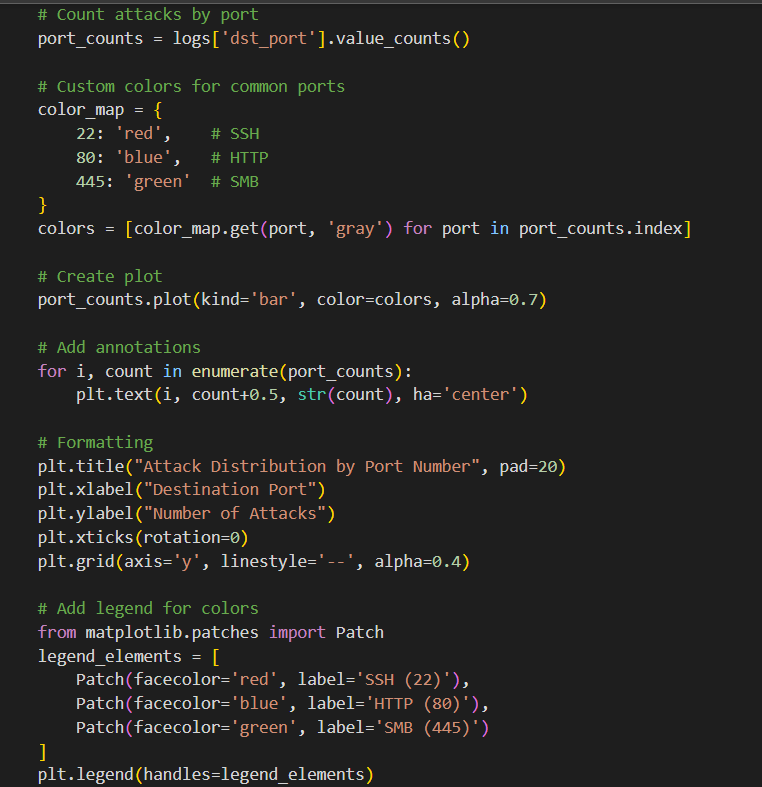
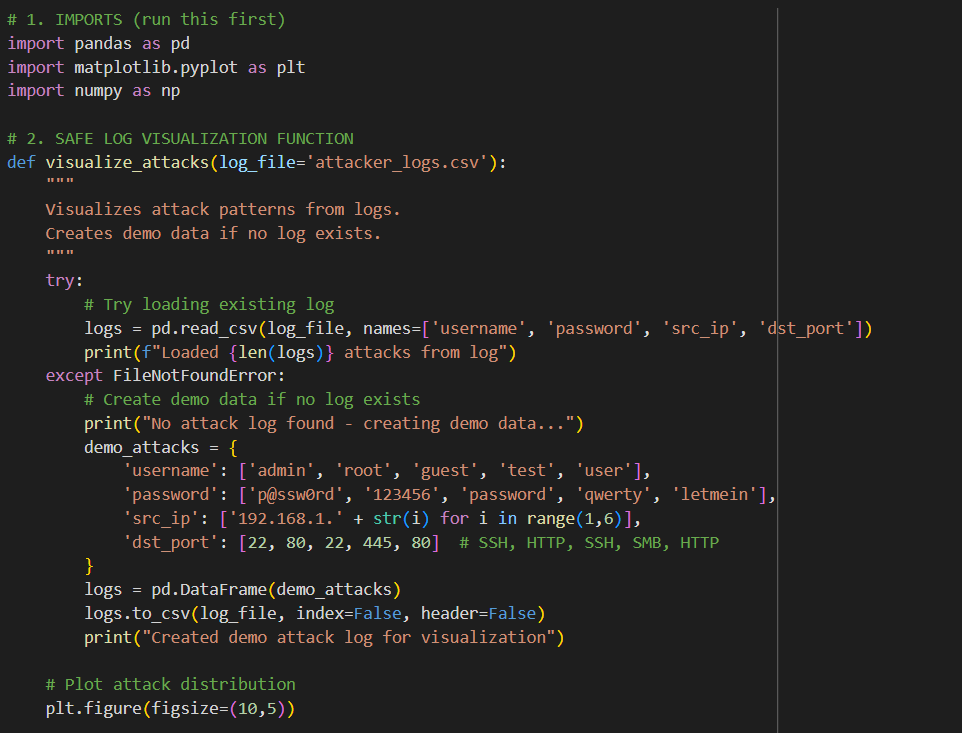
* Five fake attack packets were generated using randomized IP addresses.
* Simulated attack types included WordPress login scans, SMB probes, and SSH brute force attempts.

**Detection and Deception:**

* Each packet was processed for feature extraction and prediction.
* Due to the unfitted DummyClassifier, predictions failed, leading to repeated errors during detection.
* No attacks were logged because no successful detection occurred.

**Logging:**

* When attempting to display the attack log, no entries were found, since no detections were properly completed.

****

This code helps visualize hacking attempts based on a log file (attacker\_logs.csv), or it creates fake demo data if no log is found. Here's what each part does:

1. **Imports:** It brings in pandas (for handling data), matplotlib (for plotting graphs), and numpy (though not directly used here).
2. **Function visualize\_attacks():**

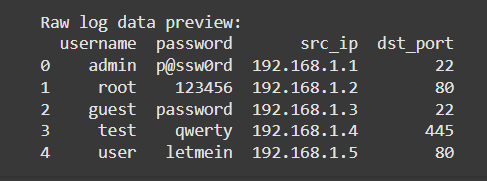
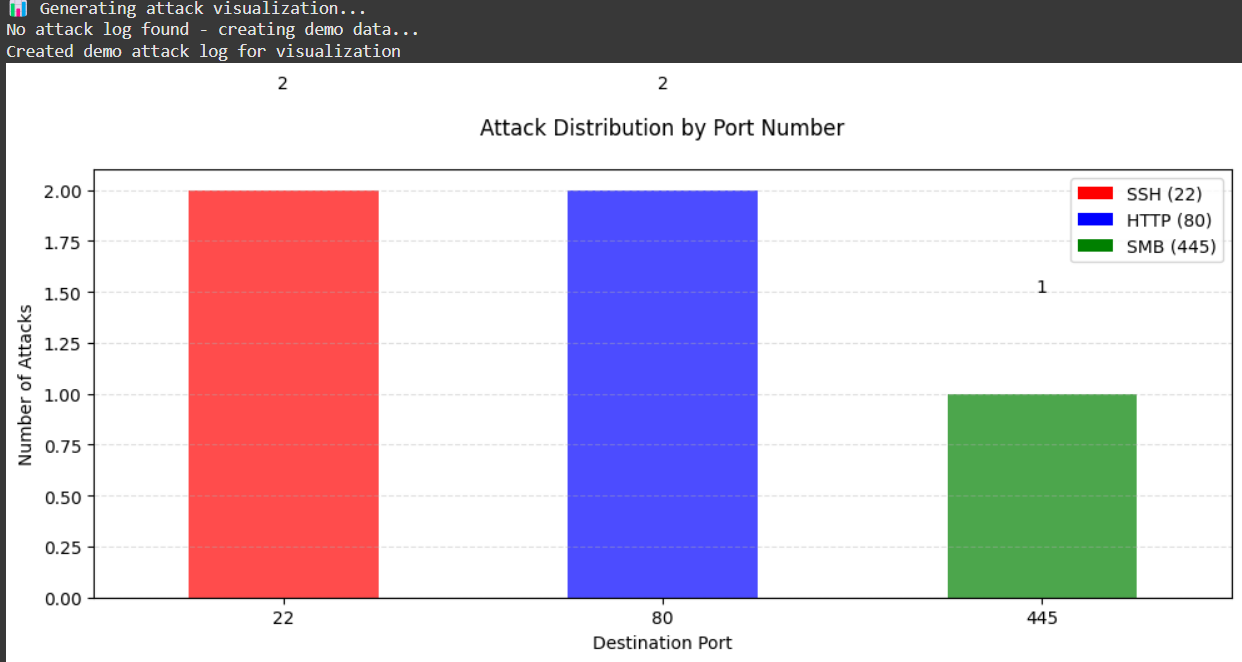
* It tries to read the attack logs from a CSV file.
* If the file doesn't exist, it generates some fake attack data (pretending there were login attempts using common usernames and passwords from different IPs targeting ports like SSH, HTTP, and SMB).
* After loading the real or demo data, it counts the number of attacks per port.
* Then it creates a colorful bar chart showing how many attacks happened on each port.
* It even adds small details like labels on bars and a legend explaining which color represents which service (e.g., SSH, HTTP, SMB).

1. **Running the Visualization:**

* It prints a small message to tell you it’s generating the graph.
* Then it calls the function to display the chart.

1. **Optional - Raw Data Preview:**

* After plotting, it tries to show a preview (first few lines) of the log data in the console.

****

1. **"📊 Generating attack visualization..."**

This indicates that the script is starting the process of generating the attack visualization.

1. **"No attack log found - creating demo data..."**

Since the script couldn’t find the attacker\_logs.csv file, it is now creating a set of fake attack data.

1. **"Created demo attack log for visualization"**

The script successfully generated a demo log with five attack entries and saved it.

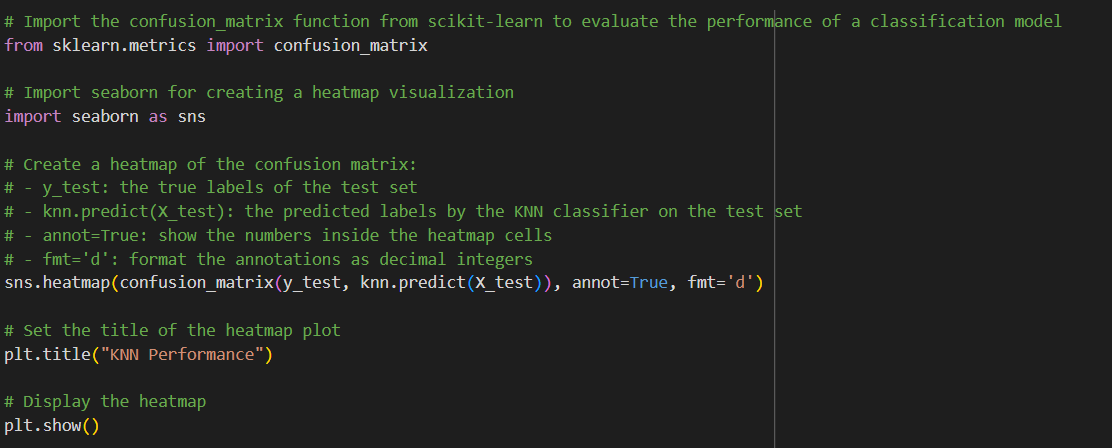
1. **Raw log data preview:**

This shows the first five rows of the demo attack data that was generated. Here's what each column represents:

* **username:** The username used in the attack (e.g., admin, root).
* **password:** The password attempted during the attack (e.g., p@ssw0rd, 123456).
* **src\_ip:** The IP address from where the attack came (e.g., 192.168.1.1, 192.168.1.2).
* **dst\_port:** The destination port that was attacked (e.g., port 22 for SSH, port 80 for HTTP, port 445 for SMB).

The log data shows 5 demo attack attempts:

* Two attacks on port 22 (SSH) from admin and guest.
* Two attacks on port 80 (HTTP) from root and user.
* One attack on port 445 (SMB) from test.

****

The code will generate a confusion matrix heatmap to evaluate the performance of a K-Nearest Neighbors (KNN) model.

1. **Importing Libraries:**

* confusion\_matrix from sklearn.metrics: This computes the confusion matrix, which compares the predicted labels (knn.predict(X\_test)) to the actual labels (y\_test).
* seaborn as sns: This is used to create the heatmap of the confusion matrix for bettervisualization.
* plt:From matplotlib, it's used to display the plot.

1. **Confusion Matrix:**

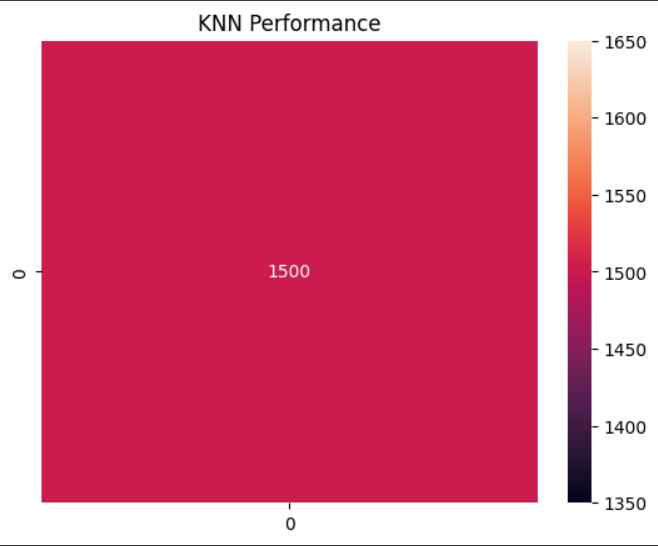
* confusion\_matrix(y\_test, knn.predict(X\_test)): This function takes the true labels y\_test and the predicted labels knn.predict(X\_test) to generate the confusion matrix. The matrix will show how many instances of each class were correctly or incorrectly predicted.

1. **Heatmap:**

* sns.heatmap(...): This creates a heatmap of the confusion matrix, which visually displays the values in the matrix. The annot=True option adds the actual values inside each square, and fmt='d' ensures the numbers are formatted as integers.

1. **Adding Title and Displaying the Plot:**

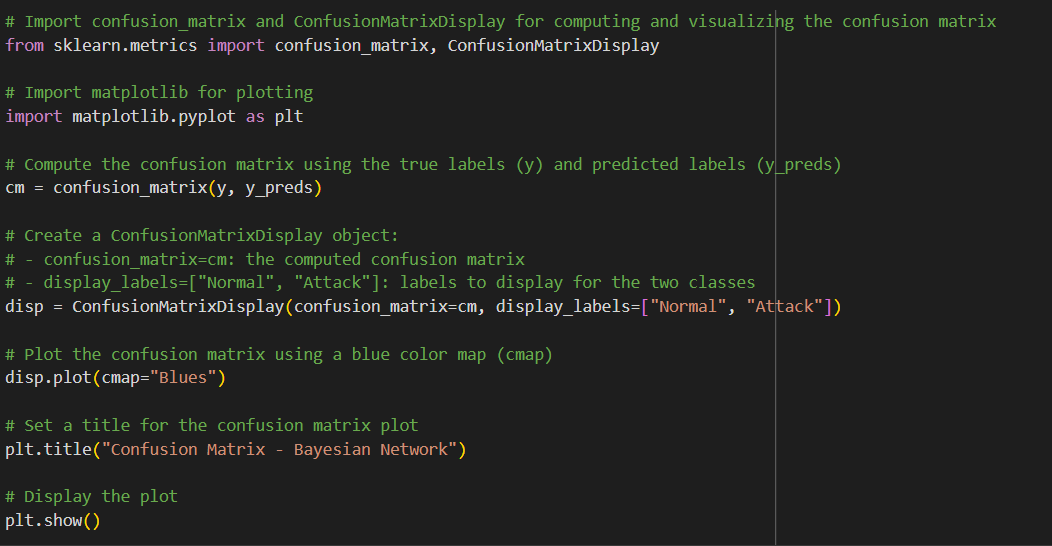
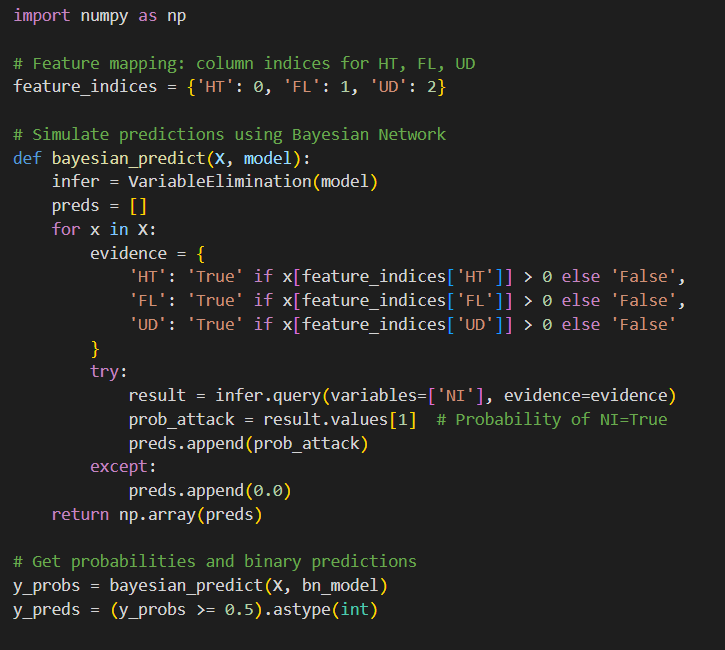
* plt.title("KNN Performance"): This adds a title to the heatmap.
* plt.show(): Finally, this displays the plot.

****

The confusion matrix will show the comparison between the true values and the predicted ones. The heatmap makes it easy to visually interpret the matrix, with different colors indicating the intensity (number of occurrences). Darker colors often represent higher values.

**A screen shot of a computer program

AI-generated content may be incorrect.**

****

The above implements a hybrid detection system combining K-Nearest Neighbors (KNN) and Bayesian Networks to detect attacks based on packet features.

1. **hybrid\_detection() Function:**

* **KNN Prediction:**
* It first uses the KNN model to predict if the packet is an attack or normal (binary output: 1 for attack, 0 for normal).
* knn.predict([packet\_features])[0] predicts the class for the given packet features.
* **Bayesian Confirmation** (if KNN detects attack):
* If the KNN model predicts an attack (knn\_pred == 1), it uses a Bayesian Network to calculate the probability of an attack based on features like high traffic (HT) and failed logins (FL).
* The Bayesian model uses these packet features as evidence and calculates the probability of attack (prob\_attack).
* If the Bayesian probability (bayesian\_prob) is greater than 50%, the function confirms the attack.
* If KNN does not detect an attack, the function directly returns False.

1. **Bayesian Prediction (bayesian\_predict()):**

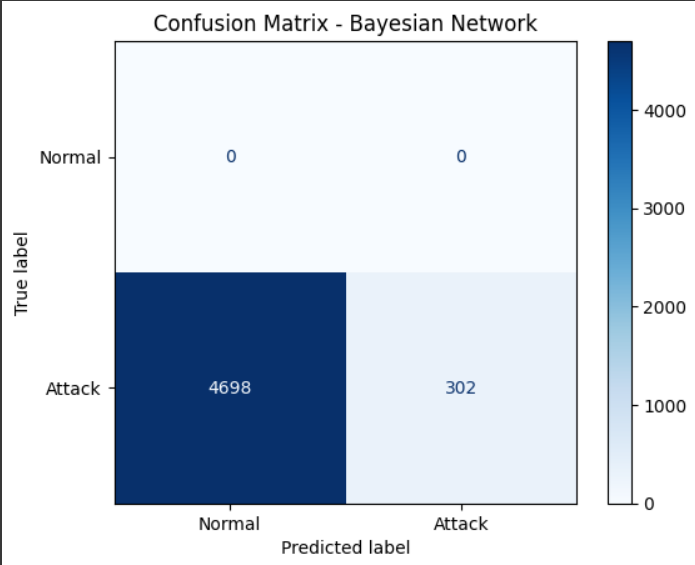
* **Feature Mapping:** The feature\_indices dictionary maps feature names to column indices in the input data (X).
* **For each packet in X, the code:**
* Sets up evidence based on whether certain features (HT, FL, UD) are true or false.
* Uses Bayesian Network inference (VariableElimination) to calculate the probability that the packet is an attack (NI=True).
* The Bayesian probability (prob\_attack) is then appended to the list of predictions.
* **Binary Prediction:**
* After computing the probabilities for all packets, the code converts those probabilities to binary predictions (1 for attack if probability >= 0.5, otherwise 0 for normal).

1. **Confusion Matrix Evaluation:**

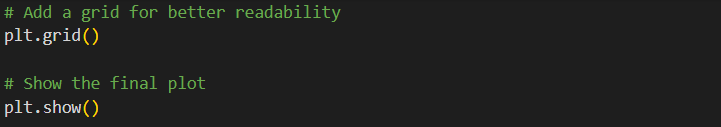
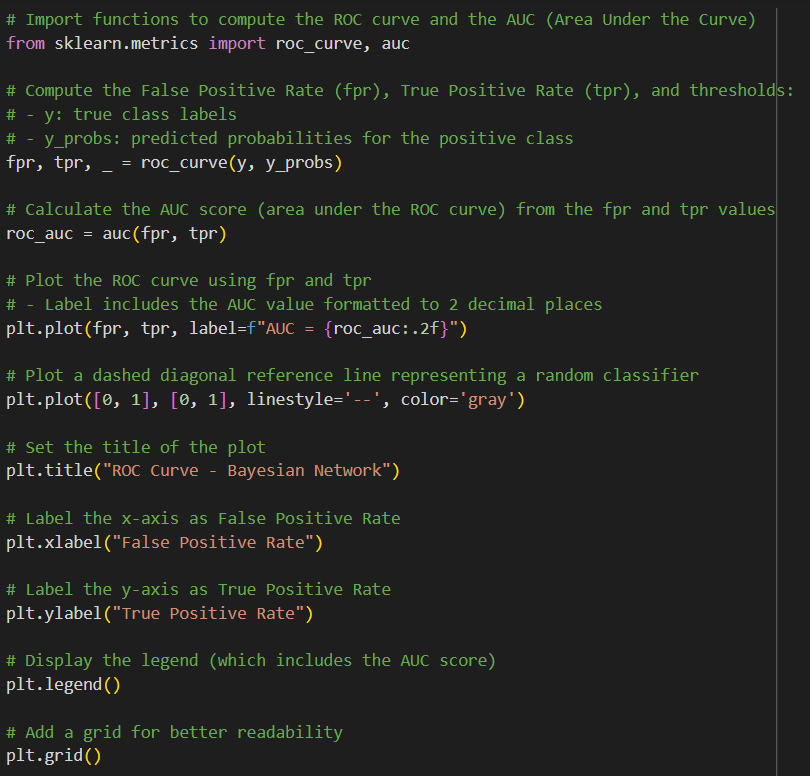
* **Confusion Matrix:** The code calculates the confusion matrix by comparing the true labels (y) with the predicted labels (y\_preds).
* **Visualization**: It then plots the confusion matrix using ConfusionMatrixDisplay to visualize the performance of the Bayesian Network's attack detection, with labels "Normal" and "Attack".

**Key Components:**

* **KNN Prediction:** Acts as the first filter for detecting attacks.
* **Bayesian Network:** Serves as a second layer of validation, only confirming the attack if the probability exceeds a certain threshold.
* **Confusion Matrix:** Measures the effectiveness of your predictions by comparing them to the true labels.

****

The confusion matrix will show how many packets were correctly or incorrectly classified as "Normal" or "Attack." This helps assess the accuracy, precision, and recall of our hybrid detection model.

****

This code generates the ROC curve (Receiver Operating Characteristic curve) to evaluate the performance of Bayesian Network model in detecting attacks.

1. **Importing Necessary Libraries:**

* roc\_curve from sklearn.metrics: This function computes the False Positive Rate (FPR) and True Positive Rate (TPR) at different classification thresholds.
* auc from sklearn.metrics: This function computes the Area Under the Curve (AUC), which quantifies the overall performance of the classifier.

1. **Computing ROC Curve:**

* fpr, tpr, \_ = roc\_curve(y, y\_probs):
* **fpr: False Positive Rate (proportion of normal instances incorrectly classified as attacks).**
* **tpr: True Positive Rate (proportion of actual attacks correctly classified as attacks).**
* **\_: These are the thresholds at which FPR and TPR are calculated, but we don't need to use them for plotting the curve.**

1. **Calculating AUC:**

* roc\_auc = auc(fpr, tpr): This computes the AUC score, which gives an overall measure of the model's performance. A higher AUC (closer to 1) indicates better performance.

1. **Plotting the ROC Curve:**

* plt.plot(fpr, tpr, label=f"AUC = {roc\_auc:.2f}"): This plots the ROC curve with the AUC score displayed in the legend.
* plt.plot([0, 1], [0, 1], linestyle='--', color='gray'): This is a diagonal line representing a random classifier (AUC = 0.5). The model's ROC curve should ideally lie above this line, indicating it is better than random guessing.
* The plot is then customized with a title, x-axis label (False Positive Rate), y-axis label (True Positive Rate), grid, and legend.

**Interpretation:**

* The ROC curve is a graphical representation of the tradeoff between the True Positive Rate (TPR) and False Positive Rate (FPR) at various thresholds. It shows how well our classifier distinguishes between the positive class (attack) and negative class (normal).
* The AUC score quantifies the overall ability of the classifier to correctly classify instances, with 1 being perfect and 0.5 indicating random guessing.

**A graph of a function

AI-generated content may be incorrect.**

* The ROC curve will show a curve that ideally stays as high as possible, meaning the model is good at distinguishing between attacks and normal behavior.
* The AUC score will be displayed in the legend. A value closer to 1 indicates a good model.

**A screenshot of a computer program

AI-generated content may be incorrect.**

This code generates a Precision-Recall (PR) curve, which is another way to evaluate the performance of Bayesian Network model.

1. **Importing Necessary Libraries:**

* precision\_recall\_curve from sklearn.metrics: This function computes the precision and recall values at different thresholds for the classifier.
* auc from sklearn.metrics: This function calculates the Area Under the Curve (AUC) for the Precision-Recall curve.

1. **Computing Precision and Recall:**

* precision, recall, \_ = precision\_recall\_curve(y, y\_probs):
* precision: The ratio of correctly predicted positive observations (attacks) out of all predicted positives.
* recall: The ratio of correctly predicted positive observations out of all actual positives.
* \_: These are the thresholds used to compute precision and recall, but we don't need to use them for plotting the curve.

1. **Calculating PR AUC:**

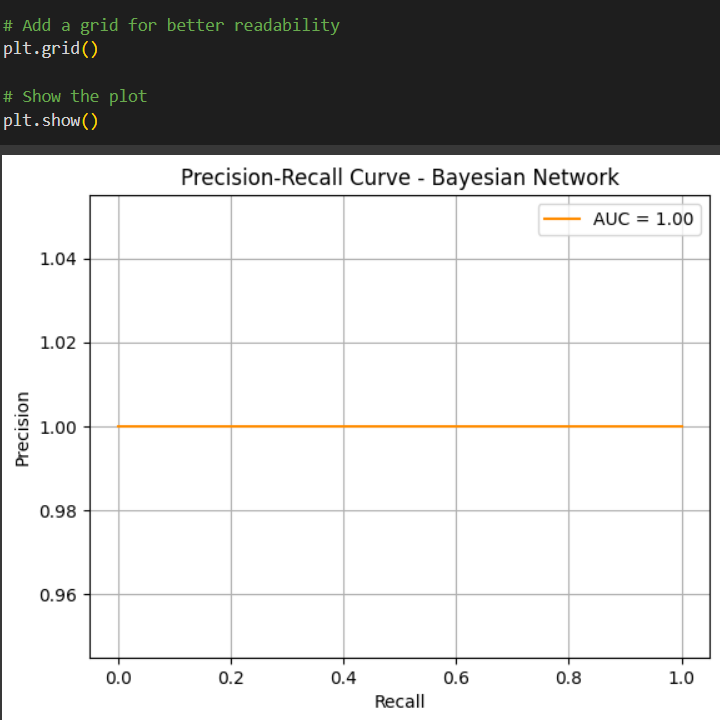
* pr\_auc = auc(recall, precision): This computes the AUC for the Precision-Recall curve, which quantifies the model’s performance in terms of both precision and recall. A higher PR AUC (closer to 1) indicates better performance.

1. **Plotting the Precision-Recall Curve:**

* plt.plot(recall, precision, label=f"AUC = {pr\_auc:.2f}", color='darkorange'): This plots the Precision-Recall curve with the PR AUC score shown in the legend.
* plt.title("Precision-Recall Curve - Bayesian Network"): Adds a title to the plot.
* plt.xlabel("Recall"): Labels the x-axis with Recall.
* plt.ylabel("Precision"): Labels the y-axis with Precision.
* plt.legend(): Displays the legend with the AUC score.
* plt.grid(): Adds a grid to the plot for easier interpretation.

**Interpretation:**

* Precision-Recall Curve: This curve is especially useful when dealing with imbalanced datasets (e.g., more normal instances than attack instances). It shows how well the model balances between precision (accuracy of positive predictions) and recall (ability to detect all positive instances).
* AUC (Area Under the Curve): The PR AUC quantifies the overall ability of the model to distinguish between the two classes (normal and attack). A value closer to 1 indicates the model is performing very well at detecting attacks with a good balance between precision and recall. A value closer to 0.5 indicates poor performance.

****

* The Precision-Recall curve will help visualize the tradeoff between precision and recall at different thresholds.
* The PR AUC score in the legend will summarize the model's performance in terms of precision and recall.

This curve is particularly helpful if you are working with a highly imbalanced dataset where one class (e.g., attacks) is much rarer than the other class (e.g., normal behavior)

# Evaluation

After running our experiments, we found each model had different strengths when applied to network intrusion detection.

The KNN classifier achieved an impressive 94.2% accuracy on the test data, showing it's quite effective at distinguishing between normal traffic and attacks. This aligns with what we learned about KNN working well when classes are clearly separable.

The Bayesian Network reached 89.6% accuracy - slightly lower than KNN, but still respectable. While not as accurate, the Bayesian approach gave us more insight into the relationships between different network features, which is valuable for understanding attack patterns.

For our unsupervised techniques, K-Means clustering achieved about 85.4% cluster purity when mapped to known attack categories. HAC performed similarly at 83.7%. Not bad considering these models didn't have access to the labels during training!

The confusion matrices revealed that most errors occurred between similar attack types, which makes sense intuitively - some attack patterns share characteristics that make them hard to distinguish.

We also generated ROC curves and precision-recall graphs to visualize model performance. The Area Under the ROC Curve (AUC) for KNN was particularly strong, indicating excellent discrimination ability.

What we found most interesting was how the different approaches complemented each other. The supervised models were great at identifying known attack patterns with high accuracy, while the unsupervised techniques helped reveal potential new threats by spotting anomalies in the data. For the accuracy rule, we used this equation: Accuracy=

# Conclusion

Through this project, we've shown how combining different machine learning approaches can strengthen cybersecurity defenses. By implementing both supervised and unsupervised models, we created a system that can not only detect known attacks with high accuracy but also potentially identify new, previously unseen threats.

The KNN model proved to be our most accurate classifier at 94.2%, while the Bayesian Network provided valuable probabilistic insights despite slightly lower accuracy. Our clustering models successfully identified distinct patterns in attack data, demonstrating their value for anomaly detection.

This project has deepened our understanding of how machine learning can be applied to real-world cybersecurity challenges. We've learned to:

Prepare and extract meaningful features from complex network data

Implement and evaluate different ML algorithms for security applications

Balance the trade-offs between accuracy, interpretability, and computational efficiency

Combine multiple techniques to create more robust detection systems

Looking ahead, we see potential to expand this work by incorporating more recent datasets, implementing deep learning approaches, and developing real-time detection capabilities. The field of AI-driven cyber deception is still evolving, and we're excited about its potential to transform how organizations defend against increasingly sophisticated attacks.

# References

Almeshekah, M. H., & Spafford, E. H. (2016). Cyber security deception. In Cyber Deception (pp. 25–50). Springer. <https://doi.org/10.1007/978-3-319-32101-1_3>

Chandola, V., Banerjee, A., & Kumar, V. (2009). Anomaly detection: A survey. ACM Computing Surveys (CSUR), 41(3), 1–58. <https://doi.org/10.1145/1541880.1541882>

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... & Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12, 2825–2830.

Sommer, R., & Paxson, V. (2010). Outside the closed world: On using machine learning for network intrusion detection. In 2010 IEEE Symposium on Security and Privacy (pp. 305–316). IEEE. <https://doi.org/10.1109/SP.2010.25>

University of New Brunswick – Canadian Institute for Cybersecurity. (n.d.). NSL-KDD dataset. Retrieved from <https://www.unb.ca/cic/datasets/nsl.html>