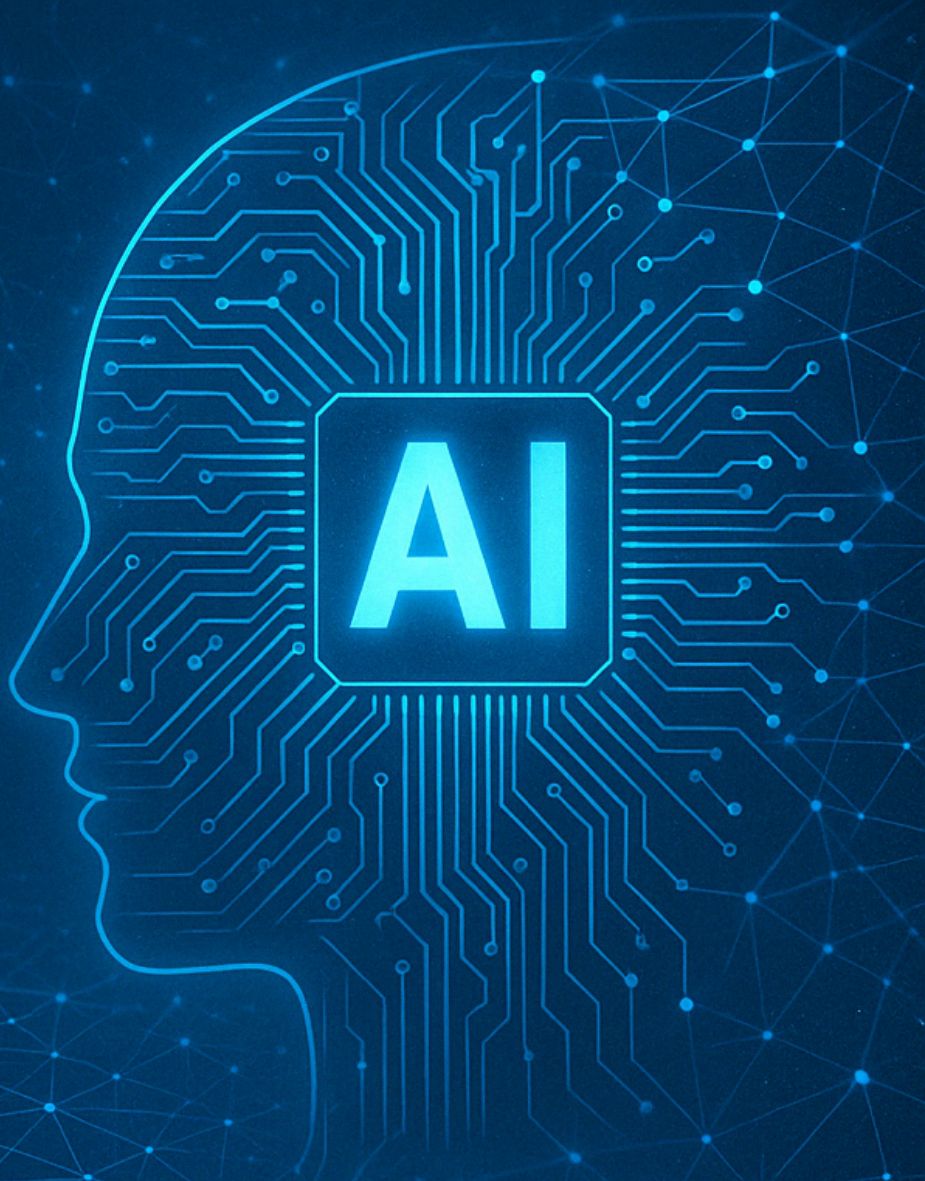
Network Anomaly Detection with AI



Compiled By Chitenga Marvellous

OVERVIEW

The solution applies machine learning to recognize unusual network behavior, such and backdoor activities. It uses modern data processing frameworks, real time streaming systems, and containerized infrastructure to build a full end to end scalable solution.

Below you'll find a full breakdown of:

- Project architecture and technologies used
- Source code (preprocessing, training, and real-time prediction)
- Pipeline flow using Kafka (Producer → Consumer → Model)
- Diagrams to illustrate system and tool stack
- Challenges faced and major debugging breakthroughs
- CRISP-DM flow and design thinking process

Note: This is not the final version of the project. Some stages (like deployment, dashboard, full automation) are still in progress. The full documentation will be released in future updates.

Introduction

This is a software tool designed to predict **network anomalies and threats** using artificial intelligence logic. It analyzes patterns in network traffic and flags suspicious or abnormal behavior. The goal is to provide a smart way to detect threats in real time without relying on manual inspection.

The system is particularly focused on detecting:

- Performance issues caused by unexpected or excessive network activity
- Unusual network traffic that may indicate intrusions or misuse
- Threats to network integrity, such as data manipulation or unauthorized access

- Irregular practices or loopholes in the system that could be exploited

It aims to recognize early signs of network based attacks, such as:

- **Fuzzers** (automated input generators that cause instability)
- **DDoS attacks** (distributed denial of service)
- **Exploits** targeting vulnerabilities
- **Malicious scanning or probing** behaviors

By identifying these issues early, the tool helps improve network **security**, **resilience**, and **operational efficiency**.

Problem Statement

Modern digital network infrastructures are increasingly complex and interconnected, making traditional security practices insufficient. Organizations today face a wide spectrum of network threats — from stealthy data exfiltration, and vulnerability exploits to high-volume DDoS attacks and intelligent fuzzers designed to probe weaknesses. These threats not only jeopardize data integrity but can paralyze operations entirely.

The challenge is two fold:

1. The scale and speed of modern attacks exceed human monitoring capacity.
2. Conventional rule-based intrusion detection systems often fail to catch novel or evolving threats.

This project addresses a growing need in sectors like IT firms, cloud service providers, and enterprises operating on-premises or hybrid network architectures, particularly those with dedicated cybersecurity teams or operating in compliance-heavy environments such as finance, healthcare, and logistics.

With the rise of automated threats, such as malicious bots, autonomous fuzzers, and AI driven exploit kits, traditional rule based detection mechanisms have become increasingly ineffective. These bots are no longer simple scripts they’re adaptive, evasive, and designed to mimic legitimate traffic patterns, making detection significantly harder.

As organizations scale digitally, they face a surge in attack surfaces and entry points. Without intelligent systems that learn and adapt, security teams are

often outpaced, reacting to threats after damage has occurred. This tool introduces a machine learning approach that learns network behavior patterns and identifies anomalies in real time, even if those anomalies don’t match known threat signatures.

By leveraging machine learning to model what "normal" traffic looks like, this tool can:

- Dynamically flag anomalous patterns that deviate from baseline behavior
- Identify early-stage indicators of compromise (IOCs)
- Detect previously unseen attack vectors with minimal false positives

For IT developers and cybersecurity engineers, this means:

- Faster incident response, thanks to early and intelligent alerts
- Reduced noise and fatigue from false alarms
- A foundation to build automated remediation pipelines
- A scalable, pluggable detection layer that integrates into CI/CD workflows or security dashboards

This tool isn’t just for analysis; it's a decision enabler. It equips teams with the insight needed to take proactive measures, mitigate risk, and harden their network against evolving cyber threats.

Solution Overview: How the Tool Solves the Problem

The Network Anomaly Detection Tool leverages machine learning to transform network security from a reactive, manual process into a predictive, intelligent system that operates autonomously, continuously learning and adapting. By doing so, it overcomes the limitations of traditional detection systems that rely solely on predefined rules and signature-based approaches. Here's how:

1. **Data Collection**: gathering and preprocessing network data to use for training and to ensure accurate anomaly detection.
2. **Model Training**: The algorithm learns to differentiate between **normal** and **unusual traffic** using unsupervised learning to detect new threats.
3. **Real-Time Detection**: Monitors live traffic to predict and flag suspicious activities like DDoS or bot behavior.

4. **Alerts & Dashboard**: Sends alerts and visualizes anomalies for quick response.
5. **Continuous Learning**: Continuously improves by learning from new data to detect emerging threats.

Approach: Project Phases

-The approach was based on the CRISP-DM approach:

1. Business Understanding:

   Identify the critical need for automated network anomaly detection to help organizations detect threats like DDoS attacks, botnets, and other security risks in real time. This tool will automate the detection process, enabling faster and more accurate responses, especially for organizations with dedicated cybersecurity teams.

2. Data Understanding:

   Using the UNSW dataset from Kaggle, which includes a variety of network traffic data, we gain insights into network behaviors and anomalies. This dataset, containing over 90,000 rows and 40 columns, provides rich data to understand normal network behavior and various types of cyber threats. We also had a separate testing dataset, ensuring model validation and accuracy.

3. Data Preparation:

   The raw data is cleaned and preprocessed in Kaggle to eliminate noise, handle missing values, and normalize features. The goal is to ensure the data is structured and ready for model training. This step is crucial to ensure that the machine learning model receives high-quality input for accurate anomaly detection.

4. Modeling:

   Random Forest, an ensemble learning method, is used for classification to detect network anomalies. The model is trained on the training dataset, learning to distinguish between normal and unusual traffic patterns based on the dataset's features. Random Forest’s ability to handle large, high-dimensional data makes it well-suited for identifying complex patterns in network traffic and detecting both known and novel threats.

5. Evaluation:

   The model is evaluated on the testing dataset to assess its ability to identify network anomalies while minimizing false positives. Key evaluation metrics such as precision, recall, and F1-score are used to measure performance, ensuring that the model can accurately detect abnormal traffic behaviors.

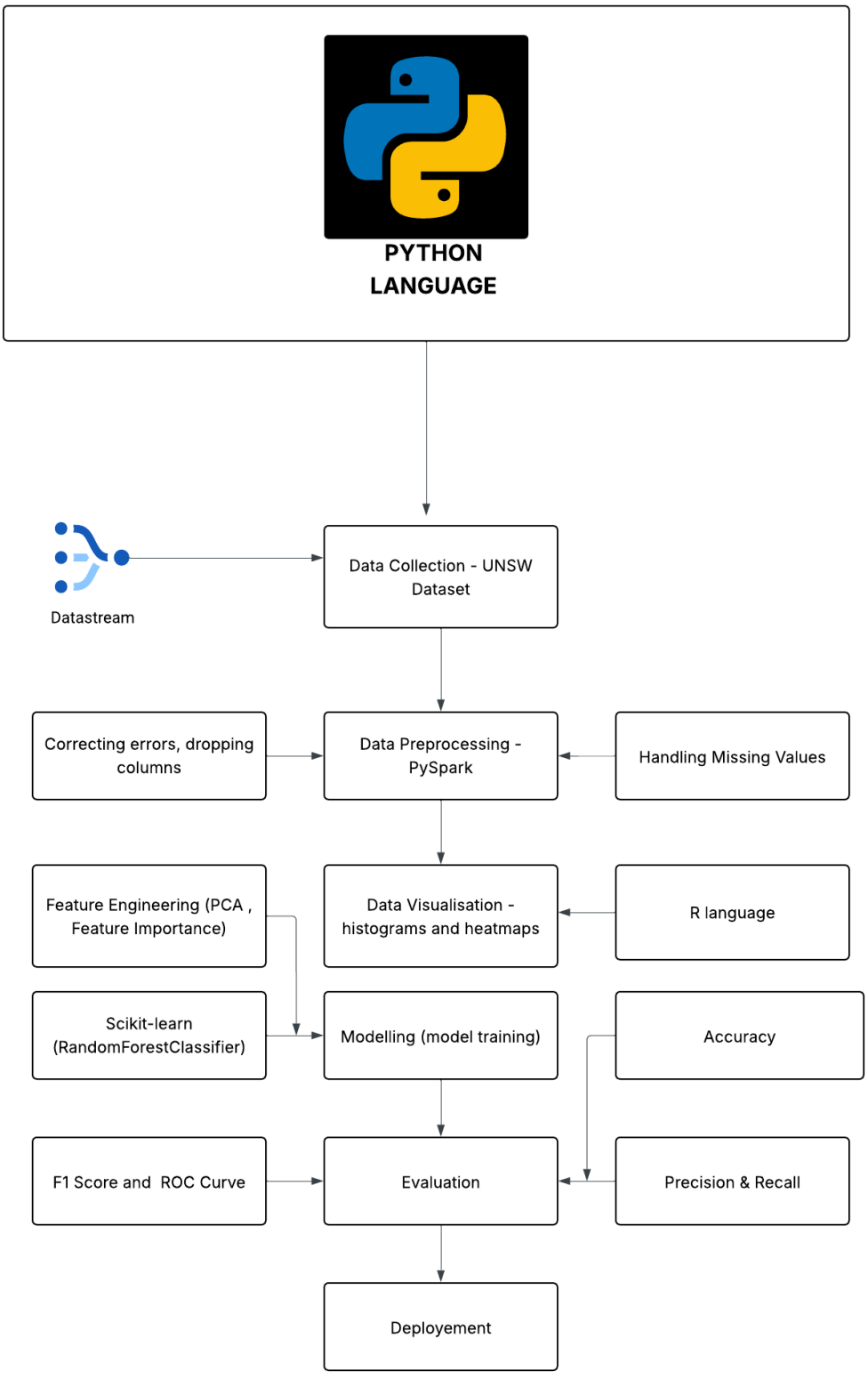
6. Deployment:

   Once the model is trained and evaluated, it is integrated into a real-time monitoring system. The model continuously analyzes live network traffic, detects anomalies, and triggers alerts for suspicious activity such as botnet behavior, DDoS attacks, or data exfiltration attempts.

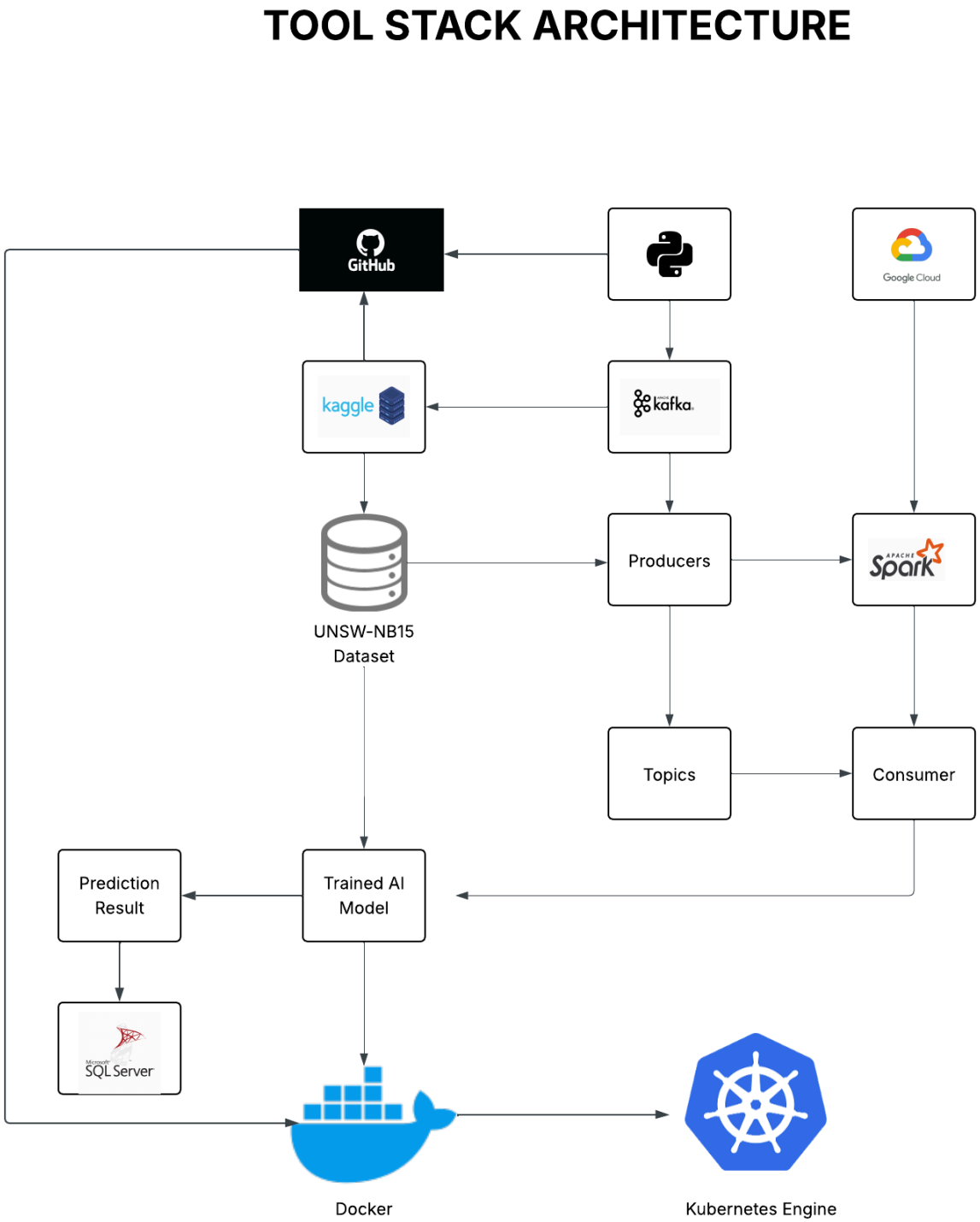
7. Monitoring & Maintenance:

   After deployment, the tool's performance is continuously monitored. As the network evolves and new threats emerge, the model is updated with fresh data to ensure it adapts to changing patterns in network traffic. This helps maintain the effectiveness of the anomaly detection tool over time.

PROJECT ARCHITECTURE



Tool Stack Architecture



Development Environment Setup & Early Stage Implementation

Platform Decision: Why Kaggle?

Due to limited compute power on local machines we migrated development to **Kaggle**, a robust cloud-based platform offering:

- Free access to **GPUs, TPUs**, and s**ample CPU resources to train machine learning model**
- A pre-configured environment with **popular Python libraries like Pytorch and Tensorflow which require large storage capacities ton install and run on local pc**
- Seamless integration with datasets and notebooks
- Support for collaborative, shareable, and reproducible workflows

This decision significantly **accelerated prototyping**, minimized setup effort, and enabled smoother execution of resource-intensive tasks like model training.

Step-by-Step Kaggle Workflow

1. Kaggle Setup

- **Account Creation**: Signed up at kaggle.com
- **Notebook Usage**: Leveraged **Kaggle Notebooks** (Jupyter-style) to run Python code online

2. Dataset Acquisition

- Downloaded the **UNSW-NB15** dataset directly from Kaggle
- Dataset Details:
  - **Training Set**: ~90,000 rows, 40 features
  - **Testing Set**: Used later for evaluation with over 45k rows

3. Uploading Dataset

- Uploaded datasets manually using the "**Add Data**" feature within the Kaggle notebook interface.

4. Libraries Used

- No manual installation required; Kaggle includes:
  - `pandas`, `numpy` for data wrangling
  - `matplotlib`, `seaborn` for visualization
  - `sklearn` for machine learning
  - `joblib` for saving/loading models
  - `pyspark` (added via pip if needed)

Data Preprocessing & Visualization (OOP Approach)

To enforce **clean structure and reusability**, the preprocessing phase was fully implemented in **Object-Oriented Python approach**, combining **PySpark for scalable data cleaning** and **Pandas for rich visualizations**.

Class-Based Python Structure

Installing & Importing Libraries

```
pip install pyspark
```

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when, isnn, count, mean, stddev
from pyspark.ml.feature import StringIndexer
```

DataCleaning Class (PySpark Based)

```
class DataCleaning:
    def __init__(self, dataset_path):
        self.spark = SparkSession.builder.appName("DataCleaning").getOrCre
ate()
        self.df = self.spark.read.parquet(dataset_path, header=True, inferSche
ma=True)
```

- Initializes a **Spark session**
- Loads the dataset in **Parquet format** because dataset was a parquet file and not csv

---

Handling Missing Values

```
def handle_missing_values(self):
    missing_values = self.df.select([
        count(when(col(c).isNull() | (col(c).isNotNull() if c not in self.df.column
s else isnan(c)), c)).alias(c)
        for c in self.df.columns])
    missing_values.show()

    numeric_cols = [c for c, dtype in self.df.dtypes if dtype in ['int', 'double']]
    for col_name in numeric_cols:
        mean_value = self.df.select(mean(col_name)).collect()[0][0]
        self.df = self.df.na.fill({col_name: mean_value})
```

- Displays null/NaN counts
- Fills missing numeric values using **column mean**

Clear explanation of the block of code above:

Handling Missing Values - Code Breakdown

```
missing_values = self.df.select([
    count(when(col(c).isNull() | (col(c).isNotNull() if c not in self.df.columns e
lse isnan(c)), c)).alias(c)
    for c in self.df.columns])
missing_values.show()
```

What It Does:

- This line **checks every column** in the DataFrame for missing values.
- It uses `when(...).alias(c)` to **count how many nulls or NaNs** exist in each column.
- `isNull()` checks for `null` entries.
- `isnan()` checks for `NaN` (Not a Number) — important for **numeric types**.
- The output is a summary table showing the **missing value count** per column.

```
numeric_cols = [c for c, dtype in self.df.dtypes if dtype in ['int', 'double']]
```

What It Does:

- It filters and creates a list of columns with numeric data types (`int` or `double`).
- These are the only columns for which filling with the mean makes sense.

```
for col_name in numeric_cols:
    mean_value = self.df.select(mean(col_name)).collect()[0][0]
    self.df = self.df.na.fill({col_name: mean_value})
```

What It Does:

- **Loops through each numeric column**.
- Calculates the **mean** of the column using `mean(col_name)`.
- Uses `.na.fill()` to **replace all missing values in that column** with the mean.
- This is a common **data imputation technique** to avoid biasing the dataset.

Result:

- All missing values in numeric columns are **filled with the column's mean**, ensuring the dataset is **numerically complete and clean**.
- The logic avoids affecting categorical or string columns which would need different strategies (e.g., mode, 'Unknown').

Removing Duplicates

```
def handle_duplicates(self):
    self.df = self.df.dropDuplicates()
```

- drops duplicates to avoid biased training

Removing Outliers

```
def handle_outliers(self):
    numeric_cols = [c for c, dtype in self.df.dtypes if dtype in ['int', 'double']]
    for col_name in numeric_cols:
        stats = self.df.select(mean(col_name), stddev(col_name)).first()
        mean_value = stats[0]
        stddev_value = stats[1]
        self.df = self.df.filter((col(col_name) > (mean_value - 3 * stddev_value))
&
```

```
                    (col(col_name) < (mean_value + 3 * stddev_value)))
```

- Applies **Z-score filtering** to remove extreme values
- This is done by filtering extreme values away form the standard deviation

This is a **common statistical method** for detecting outliers using the **3 sigma rule**.

- The idea is that in a **normal distribution**:
  - 99.7% of the data falls within **±3 standard deviations** from the mean.

So:

- Values outside `mean ± 3 * stddev` are **extreme outliers**.
- We filter them out to **reduce noise** and improve model performance.

Encoding Categorical Features

```
def encode_categorical_values(self):
    string_cols = [c for c, dtype in self.df.dtypes if dtype == 'string']
    for col_name in string_cols:
        indexer = StringIndexer(inputCol=col_name, outputCol=col_name + "_i
ndex")
        self.df = indexer.fit(self.df).transform(self.df)
```

- Converts string categories into numeric indices required for ML models

Why??:

- Because ML models dont understand text or string values but numerical.
- Full explanation of what encoding is below:

What is Encoding?

Machine learning models can't understand text-based values. So we **encode categorical (string) features** by converting them into **numerical formats** that models can work with.

Example:

- `"TCP"` → `0`
- `"UDP"` → `1`
- `"ICMP"` → `2`

Why We Did It in This Project

Our dataset contains several categorical columns like:

- `protocol`, `state`, `service`, etc.

To prepare them for modeling, we needed to:

- Convert all string values to numbers
- Avoid model errors (since models only accept numeric inputs)
- Preserve important features for training

How We Did It in PySpark

```
string_cols = [c for c, dtype in self.df.dtypes if dtype == 'string']
for col_name in string_cols:
    indexer = StringIndexer(inputCol=col_name, outputCol=col_name + "_ind
ex")
    self.df = indexer.fit(self.df).transform(self.df)
```

- `StringIndexer` converts each string column into a **numerical label** column.
- It adds a new column with the suffix `_index` (e.g., `protocol_index`).
- This is a required first step before you can do further encoding like one-hot.

Connecting to One-Hot Encoding

Some ML algorithms (like logistic regression, deep learning) perform better with **One-Hot Encoded** features instead of label-encoded values.

One-Hot Encoding converts numerical labels into **binary vectors**, like this:

TCP	0	`[1, 0, 0]`
Protocol	protocol_index	protocol_vec
UDP	1	`[0, 1, 0]`
ICMP	2	`[0, 0, 1]`

Important Rule in PySpark:

> In PySpark, you cannot apply OneHotEncoder directly to string columns.
>
> You **must** first apply `StringIndexer`, which we did in encoding categorical values because `OneHotEncoder` expects **numeric indices**.

Correct Order in PySpark:

1. `StringIndexer` (encoding categoricalvalues) → turns strings into labels
2. `OneHotEncoder` → turns labels into binary vectors

How It's Different in Scikit-learn

In **Scikit-learn**, you can use `OneHotEncoder` **directly on string columns** without needing to label encode first.

Example in sklearn:

```
from sklearn.preprocessing import OneHotEncoder
import pandas as pd

df = pd.DataFrame({'protocol': ['TCP', 'UDP', 'ICMP']})
encoder = OneHotEncoder()
encoded = encoder.fit_transform(df[['protocol']])
```

In Scikit-learn:

- No need for `StringIndexer`

- `OneHotEncoder` can handle strings directly
- Works seamlessly with `pandas`

Why Is It Different in PySpark vs Scikit-learn?

The difference comes down to how **each framework handles data internally**:

🔷 PySpark

- PySpark is built on top of **Apache Spark**, which is a **distributed computing framework**.
- It processes data in **distributed clusters**, not in-memory like Pandas.
- To maintain consistency across **massive datasets split across nodes**, Spark uses a **strict schema system**.
- This means functions like `OneHotEncoder` in Spark **require all inputs to be numeric and typed correctly** before transformations.

So you need to use `StringIndexer` first to convert text to numeric labels before applying `OneHotEncoder`.

🔷 Scikit-learn

- Scikit-learn works with **small to medium in-memory datasets**, using **NumPy arrays** and **Pandas DataFrames**.
- It's more flexible because everything is already in memory, so the encoders can **dynamically infer and transform string values directly**.
- `OneHotEncoder` in Scikit-learn is designed to work **directly with string labels** and automatically handles them internally.

You **don't need to encode first** — you can just plug in your string column directly.

Summary of the Difference

Feature	PySpark	Scikit-learn
Environment	Distributed (Big Data)	In-memory (Small/Medium data)
Schema Strictness	High – needs numeric inputs	Low – handles strings directly

One-Hot Input Type	Requires numeric (after StringIndexer)	Can use strings directly
Workflow	`StringIndexer` → `OneHotEncoder`	Just `OneHotEncoder`

Note :

> The need for StringIndexer before OneHotEncoder in PySpark comes from its strict schema system and distributed processing design. Unlike Scikit-learn, which handles small in-memory data and can dynamically process strings, PySpark requires all data to be explicitly typed and structured before transformations like encoding can occur.

Summary

> In this project, we encoded categorical features using StringIndexer in PySpark to convert string columns into numeric labels. This step was essential for model compatibility and further transformations.
>
> Unlike Scikit-learn, where `OneHotEncoder` can be applied directly to string values, PySpark **requires** label encoding first before one-hot encoding. This two-step process ensures data is clean, consistent, and ready for use in machine learning models.

Return Cleaned Data

```
def return_cleaned_data(self):
    return self.df
```

- Makes the cleaned DataFrame available for the next stage

DataPreprocessing Class (Pandas-Based Visuals)

```
class DataPreprocessing:
    def __init__(self, cleaned_df):
        self.df = cleaned_df.toPandas()
```

- Converts Spark DataFrame to **Pandas because we cant use spark directly with the functions we wanted to use** for visualization

Visualize Dataset

```
def visualize_data(self):
    print(self.df.head())
    print(self.df.describe())

    self.df.hist(figsize=(10, 10), bins=20)
    plt.tight_layout()
    plt.show()
```

- Prints first few rows & descriptive statistics
- Plots **histograms** for distribution analysis

Example Visualizations & Goals

- **Correlation Heatmaps**: To detect feature interdependence

- **Boxplots**: For visualizing outliers
- **Histogram Distributions**: To examine normality

Outcome of Preprocessing

- **Missing values handled**, **duplicates removed**, and **outliers eliminated**
- **Categorical features encoded**
- Dataset ready for feature selection and model training

Method Calls & Output: Data Cleaning and Visualization Flow

Calling the Class with Dataset

```
data_cleaning = DataCleaning('/kaggle/input/unsw-nb15-testing-set-parquet-4-54-mb/UNSW_NB15_testing-set.parquet')
```

This creates an object of the `DataCleaning` class and loads the **UNSW NB15 dataset** from Kaggle (in Parquet format) using a **Spark session**. The data is now available as `self.df`.

Handle Missing Values

```
print(data_cleaning.handle_missing_values())
```

This method:

- Scans all columns in the dataset
- Counts null or NaN values
- For **numeric columns**, it replaces missing values with the **mean** of that column

Result printed:

A table showing **all columns had 0 missing values**, meaning no actual imputation was needed, but the logic was successfully applied.

Handle Duplicates

```
print(data_cleaning.handle_duplicates())
```

This method removes **duplicate rows** from the dataset using `dropDuplicates()` in Spark.

Handle Outliers

```
print(data_cleaning.handle_outliers())
```

This method uses **Z-score logic**:

- Calculates the **mean and standard deviation** of numeric columns
- Removes rows where values are beyond `mean ± 3 * stddev`

Result:

- Outliers removed for every numeric column

Encode Categorical Values

```
print(data_cleaning.encode_categorical_values())
```

This method:

- Finds all **string columns**
- Uses `StringIndexer` to convert them into numeric values
- Adds new columns like `proto_index`, `state_index`, etc.
- **Result:**

- Encoding done successfully

Return the Cleaned Data

```
cleaned_df = data_cleaning.return_cleaned_data()
print(cleaned_df)
```

This method returns the **cleaned and transformed DataFrame**.

Sample Output:

- Columns like `proto_index`, `state_index`, `attack_cat_index` are now present
- Data is ready for use in visualization or modeling

Convert to Pandas & Visualize

```
data_preprocessing = DataPreprocessing(cleaned_df)
data_preprocessing.visualize_data()
```

Since we cant use PySpark on pandas functions we need to convert it pandas function

This:

- Converts the Spark DataFrame into a **Pandas DataFrame**
- Visualizes the data using:
  - `.head()` – shows first few rows
  - `.describe()` – stats for each column (mean, std, min, max, etc.)
  - `hist()` – histograms of distributions for numeric columns

Result Highlights:

- Data is well-distributed
- Columns like `sload`, `rate`, `sbytes`, `dbytes` have **wide variance**
- Some columns are heavily right-skewed (common in network traffic)

Final Checks

```
rows = cleaned_df.count()
columns = len(cleaned_df.columns)
print(cleaned_df.columns)
```

This confirms:

- **Row count** = 95251
- **Column count** = `40`
- Cleaned DataFrame includes both original and newly created indexed columns:

  ```
  ['dur', 'proto', 'service', 'state', ..., 'proto_index', 'service_index', 'state_index', 'attack_cat_index']
  ```

Summary

> After calling all data cleaning methods (handle_missing_values(), handle_duplicates(), handle_outliers(), and encode_categorical_values()), the dataset was fully cleaned and preprocessed. Missing values were handled, duplicates removed, extreme outliers filtered, and categorical string columns were converted into numeric format using String Indexer. The final Data Frame contained 40 columns and 95251 rows, ready for training and further processing

Model Training Evaluation with Scikit-learn

After cleaning and encoding the dataset using PySpark and Pandas, the project moves to the **model training phase**, where we apply a **Random Forest Classifier** using Scikit learn. This is done inside a class-based Python structure for modularity.

While anomaly detection is typically suited for unsupervised learning algorithms like clustering, I opted for a supervised learning approach because the dataset I used was
labeled,each record clearly indicated whether it represented normal or malicious activity. This allowed me to use supervised models like **Random Forest**, which perform well when class labels are known, enabling more accurate training and evaluation

---

Objective:

- Build a supervised machine learning model to **predict network attack categories**
- Use **Random Forest**, a powerful ensemble model that handles both numerical and categorical features well
- Evaluate the model's performance and extract feature importance and PCA

Libraries Used:

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
from sklearn.preprocessing import OneHotEncoder
```

- `train_test_split`: For dividing data into training/testing sets
- `RandomForestClassifier`: The main model used for prediction
- `classification_report`: Shows accuracy, precision, recall, F1-score

- `OneHotEncoder`: Transforms categorical features into binary vectors

Class: `TrainModel`

`__init__()` – Load Data

```
def __init__(self, training_df):
    self.df = training_df
```

- Accepts the preprocessed **Pandas DataFrame** and stores it for use inside the class.

`one_hot_encoding()` Handle Categorical Features

```
string_cols = self.df.select_dtypes(include=['object']).columns.drop('attack_cat')
encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')
```

- Identifies all **categorical (string)** columns except the target (`attack_cat`)
- Initializes the OneHotEncoder (non-sparse for readability, `handle_unknown='ignore'` to prevent crashing on unseen categories)

What is One-Hot Encoding?

One Hot Encoding is a method used to convert **categorical (string) data** into **numeric format** so that machine learning models can understand and use them.

Instead of replacing strings with numbers like `TCP → 0`, `UDP → 1`, One-Hot Encoding creates **separate columns** for each category:

- Example:
  - Original column: `protocol = ['TCP', 'UDP', 'ICMP']`

- After One-Hot:

```
protocol_TCP  protocol_UDP  protocol_ICMP
    1           0           0
    0           1           0
    0           0           1
```

This method avoids assigning any **ordinal meaning** to the values (which would be misleading).

How We Used It in This Project:

- First, we selected all columns with data type `object` which means string-based categorical columns:

```
string_cols = self.df.select_dtypes(include=['object']).columns.drop('attack_cat')
```

- We dropped the target column `attack_cat` from this list because it's our **label**, not a feature to encode.
- Then we created the encoder:

```
encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')
```

  - `sparse=False` → returns a normal DataFrame, not a compressed matrix
  - `handle_unknown='ignore'` → skips any unseen category instead of throwing errors

---

Why This Matters:

> Machine learning models like Random Forest need numerical input only. By using One Hot Encoding, we keep the meaning of the categories while converting them into a form the model can process.

This approach is especially helpful when using **Scikit-learn**, which supports one-hot encoding **directly** (unlike PySpark, which requires `StringIndexer` first).

```
transformed_data = encoder.fit_transform(self.df[string_cols])
encoded_cols = encoder.get_feature_names_out(string_cols)
encoded_df = pd.DataFrame(transformed_data, columns=encoded_cols)
```

- Transforms those columns into binary (0/1) features
- Gets readable names for new columns (e.g., `proto_tcp`, `state_FIN`, etc.)

```
self.df = pd.concat([self.df.drop(string_cols, axis=1), encoded_df], axis=1)
```

- Replaces original string columns with their one-hot encoded versions

---

🔷 `train_model()` Train, Predict, and Evaluate

```
X = self.df.drop('attack_cat', axis=1)
y = self.df['attack_cat']
```

- Features (`X`) = all columns except the label
- Target (`y`) = `attack_cat` (e.g., `Normal`, `Fuzzers`, `DoS`)

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

- Splits the data: 80% training, 20% testing , random state

```
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
```

- Trains the model using **100 trees**

What is `random_state`?

`random_state` is just a number (like a **seed**) that controls how **randomness behaves** in your code — specifically when:

- Splitting your data
- Training your model (e.g., how trees are randomly built)

Using the same `random_state` ensures you get the **same results every time** you run the code — this is called **reproducibility**

```
y_pred = model.predict(X_test)
print(classification_report(y_test, y_pred))
```

- Makes predictions
- Prints a detailed classification report (precision, recall, F1-score)

Feature Importance

```
feature_importances = pd.DataFrame(
    model.feature_importances_,
    index=X.columns,
    columns=['importance']
).sort_values('importance', ascending=False)
print(feature_importances)
```

- This shows which features contributed most to the model's decisions.
- Helps identify which network metrics (e.g., `sbytes`, `dload`, `proto_tcp`) are most predictive of attacks.

Summary

> The TrainModel class encapsulates the entire training pipeline using Scikit-learn. Categorical features were one-hot encoded using OneHotEncoder, where we passed it after encoding categorical values using PySpark where StringIndexer is required first. After preprocessing, we trained a RandomForestClassifier on the dataset and evaluated its performance using classification_report. We also extracted feature_importances_ to identify the most relevant input variables for network threat detection.

Next Stage

- After successfully training the model, the next step was to test its performance by making predictions with real network data, either from a live simulation or a server, to observe how the model performs in real-time.

- I initially thought this phase would be simpler than the earlier stages, but it turned out to be quite the opposite. I encountered several challenges that were beyond my control.
- As I worked through possible solutions, I continuously faced new obstacles, which was frustrating at times. However, I persisted and eventually found a final solution.

Challenges Faced in Implementing Real Time Streaming for Anomaly Detection

1. Initial Attempt with Kafka and Scapy on Kaggle

- **Kafka** is a distributed event streaming platform used for building real time data pipelines and streaming applications. It helps in managing high-throughput, fault-tolerant streams of data, which is essential for **real-time anomaly detection**.
- It was supposed to work together with scapy
- Scapy is a popular Python library for network packet manipulation
- it c**reates**, **send**, **capture**, and **analyze** network packets.
- **How it would have worked**:
    - Kafka would allow continuous streaming of network traffic data, enabling your model to make predictions on live packets.
    - I wanted it to work in conjunction with the scapy library
    - I would have used **Kafka Producers** to push data from the network, and **Kafka Consumers** to process and feed the data into your model for prediction.
- **The Problem**:
    - **Kaggle** doesn’t support running Kafka directly. The platform is more geared towards batch processing and model training in isolated environments, rather than streaming data in real-time.
    - The **lack of access to Kafka on Kaggle** prevented me from completing this approach. Since Kafka requires persistent infrastructure for message handling, running it on Kaggle wasn't feasible.

2. Switch to VS Code: Dependency Issues

- **VS Code** seemed like a good alternative for local development since it supports Python and external libraries well. However, running the model on my **local machine** presented the issue of limited storage space.
- **The Problem**:
    - To run my model and related processes, I needed several dependencies (e.g., Kafka, Scapy, machine learning libraries). My **local machine didn't have enough space** to install these dependencies without affecting performance.
    - **VS Code's local environment** couldn't handle the load due to space constraints, leading to a **dead end** in trying to implement real time streaming on my machine.

3. Migration to Cloud Platform (Google Cloud)

- **Google Cloud** provides scalable and cost effective solutions, especially suitable for heavy computational needs like machine learning and real time data streaming.
- **How it worked**:
    - By migrating to the **Google Cloud Platform**, I gained access to a more powerful environment with **greater storage and processing power**, which allowed me to run Kafka, install the necessary dependencies, and build out the real-time model prediction pipeline.
    - Google Cloud offered a **free tier** that was suitable for my needs, providing the resources required
    - The rest of implementation of my project I did it on google cloud
- **The Problem with Azure**:
    - While **Azure** offers free credits, the available resources (e.g., 1 GB on a virtual machine) were far too small to handle the scale of my model and the **real time data processing**. This made it an unsuitable choice compared to Google Cloud, which offered more flexibility and scalability.

Successful Transition to Cloud

- By migrating to **Google Cloud**, I was able to overcome the resource limitations and continue building the necessary **real-time data pipeline** for anomaly detection.
- With access to scalable cloud infrastructure, I was able to:
  - Stream real time data
  - Train my model on live traffic
  - Make **accurate predictions** in real time

Transition to Google Cloud

After migrating to the **cloud**, I was faced with the task of setting up my environment on a **Linux-based virtual machine** . I managed to navigate through by learning bash scripting concepts that i was not familiar with on the go and adapting to the new environment.

Transferring the Datasets

The first hurdle was **moving the datasets** (both the **training** and **testing datasets**) from Kaggle to the cloud system. Here's how I did it:

1. **Downloaded the dataset** directly from **Kaggle** to my local machine.
2. Then, using **SCP** (Secure Copy Protocol), I **moved the dataset files** from my local machine to the virtual machine:

```
scp /path/to/local/dataset user@remote_host:/path/to/remote/direc
tory
```

3. After the datasets were successfully transferred, I simply **copied the code** I had written in Kaggle's environment and **pasted** it into the virtual machine, ensuring the model logic and preprocessing steps remained intact.

Installing PySpark Dependencies

To work with **PySpark**, the next challenge was ensuring that the necessary **Java dependencies** were installed, as PySpark relies on **Java** to function correctly. Here's what I did:

Install Java dependencies: PySpark requires Java to be installed on your system. To do this, I used the following commands on the VM to install Java:

```
sudo apt update
sudo apt install openjdk-8-jdk
```

Verify Java Installation: After installing Java, I verified the installation with:

```
java -version
```

Set Environment Variables: PySpark also requires **JAVA_HOME** to be set. I did this by adding the following to my `.bashrc` file:

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export PATH=$JAVA_HOME/bin:$PATH
```

Then, I applied the changes by running:

```
source ~/.bashrc
```

Install PySpark: After the Java dependencies were set up, I installed the necessary Python libraries using:

```
pip install pyspark
```

Installing Kafka for Real-Time Data Streaming

Next came the task of setting up **Kafka**, the tool I planned to use for real-time data streaming. Since Kafka is essential for building a **real-time anomaly detection pipeline**, it was crucial to install it correctly. Here's how I set it up:

1. **Install Kafka Dependencies**:
   - **Download Kafka** from the official site:

     ```
     wget https://archive.apache.org/dist/kafka/2.8.0/kafka_2.13-2.8.0.tgz
     ```

   - **Extract** the tar file:

     ```
     tar -xvzf kafka_2.13-2.8.0.tgz
     cd kafka_2.13-2.8.0
     ```

2. **Start Zookeeper and Kafka Server**: Kafka depends on Zookeeper to manage its clusters, so I had to start it before running Kafka:
   - **Start Zookeeper**:

     ```
     bin/zookeeper-server-start.sh config/zookeeper.properties
     ```

What is Zookeeper?

Zookeeper is an open-source distributed coordination service that helps manage and coordinate distributed systems. In simpler terms, it provides a way for multiple **distributed applications** or services to

coordinate, **synchronize**, and **communicate** with each other in a reliable and fault-tolerant way.

Why Do We Need Zookeeper In this Project?

In distributed systems like **Apache Kafka**, **Zookeeper** is crucial because it performs the following tasks:

1. **Managing Kafka Brokers**:
    - Kafka consists of multiple **brokers** (servers). Zookeeper keeps track of these brokers and their status. It helps manage the **leadership election** among kafka brokers in case of failure.
2. **Maintaining Metadata**:
    - Zookeeper keeps track of the metadata about Kafka topics, partitions, and which brokers are handling each partition. This helps Kafka understand where to send or consume data.
3. **Distributed Coordination**:
    - In a **clustered environment**, Zookeeper is used for **distributed coordination**. It ensures that all nodes in the system work together as a single, reliable unit, preventing conflicts and data inconsistency.
4. **Fault Tolerance**:
    - If a Kafka broker goes down, Zookeeper helps with **failover** and **replication**, making sure the system continues to run smoothly by redirecting traffic to available brokers.

How Zookeeper Works

1. **Leader Election**:
    - Zookeeper helps in **choosing a leader** among Kafka brokers. The leader is responsible for managing partitions of data, while the others act as followers. If a leader fails, Zookeeper facilitates the election of a new one.
2. **Configuration Management**:
    - Zookeeper stores configuration data (metadata) and makes sure all brokers in the Kafka cluster are aware of changes, ensuring

consistency across the system.

3. **Distributed Synchronization**:
    - Zookeeper enables **synchronization** between different Kafka nodes (brokers). This allows Kafka to manage partitions and data replication seamlessly across multiple servers.

Starting Zookeeper in Kafka

To run **Zookeeper** for Kafka, you use the command:

```
bash
Copy
bin/zookeeper-server-start.sh config/zookeeper.properties
```

Explanation:

- `bin/zookeeper-server-start.sh`: This is the script to start Zookeeper on your Kafka setup.
- `config/zookeeper.properties`: This is the configuration file that defines how Zookeeper should run (e.g., ports, data directories, server settings).

What Happens When You Run It:

- The script **starts Zookeeper** using the settings specified in the `zookeeper.properties` configuration file.
- Zookeeper begins listening for requests from Kafka brokers and manages the coordination of these brokers in your Kafka cluster.

In Summary

Zookeeper is essential for **distributed systems like Kafka**, as it provides a reliable mechanism for **synchronization**, **fault tolerance**, and **configuration management**. It ensures that Kafka brokers can **coordinate**, manage their metadata, and handle failures smoothly.

- **Start Kafka Server**:

```
bin/kafka-server-start.sh config/server.properties
```

1. **Verify Kafka is Running**: After starting the servers, I ensured that Kafka was running by creating a **Kafka topic** and sending messages:
   - Create a topic:

     ```
     bin/kafka-topics.sh --create --topic my_topic --bootstrap-serve
     r localhost:9092 --partitions 1 --replication-factor 1
     ```

   - Produce messages to the topic:

     ```
     bin/kafka-console-producer.sh --topic my_topic --bootstrap-se
     rver localhost:9092
     ```

   - Consume messages from the topic:

     ```
     bin/kafka-console-consumer.sh --topic my_topic --from-beginn
     ing --bootstrap-server localhost:9092
     ```

Step 4: Challenges and Final Thoughts

Although I faced numerous obstacles along the way, including learning Linux basics and setting up various dependencies, I persevered. The most significant challenge was the **complexity of setting up Kafka**, as it required careful installation and configuration to work with PySpark. Each step, from transferring datasets to configuring the environment, took time, but it was all part of the learning process.

Once everything was installed and configured correctly on the virtual machine, I successfully implemented **Kafka** to handle real-time data

streaming, which was essential for my **anomaly detection model**.

Summary of the Setup Process:

- **Datasets** were moved from Kaggle to the cloud using SCP.
- Installed **Java dependencies** for PySpark, configured environment variables, and installed **PySpark** itself.
- Set up **Kafka** on the cloud VM for real-time streaming, allowing me to work with live network data.

- **Kafka Broker**:
  - A **broker** is a **Kafka server** that stores and serves data. Kafka runs in a **distributed environment**, and each broker is responsible for handling a part of the data. A **Kafka cluster** consists of multiple brokers working together.
  - Brokers manage the **storage** and **distribution** of messages across the cluster, ensure **fault tolerance** through replication, and manage **partitions**.

Kafka Producer and Consumer

1. **Kafka Producer**:
   - A **Producer** is an application or system that **produces or sends** messages to Kafka topics. In the context of our project:
     - A **Producer** could be your **network simulator it can be** real-time data coming from a **networked server**.
     - The Producer sends messages (like **network traffic data**) to Kafka topics.

     Example:
     - A **DoS attack simulation** can generate traffic (packets) and send them to Kafka.
     - A Producer sends messages to Kafka topics in real-time.

```
from kafka import KafkaProducer

producer = KafkaProducer(bootstrap_servers='localhost:9092')
producer.send('topic_name', b'Some data')
```

2. **Kafka Consumer**:
   - A **Consumer** is an application that **reads or consumes** messages from Kafka topics.
   - In our project, the **model** is the **consumer**. It receives packets from Kafka, processes them, and makes predictions on whether they are **normal** or **anomalous** (e.g., DoS, Fuzzers).

   Example:

   - A **Consumer** reads network traffic data from Kafka topics and sends it to the machine learning model for prediction.

```
from kafka import KafkaConsumer

consumer = KafkaConsumer('topic_name', bootstrap_servers='loca
lhost:9092')
for message in consumer:
    process_packet(message.value)  # Send packet data for predicti
on
```

How Kafka, Producers, Consumers, and Brokers Link Together

In a **Kafka-powered real-time system**, **Producers**, **Consumers**, and **Brokers** work together to create a **fully functional data pipeline**.

1. **Kafka Producers** send data (e.g., network traffic) into **Kafka Topics** (Kafka acts as a **message bus**). The data is stored in **partitions** across the **Kafka Brokers** in a distributed manner.

2. **Kafka Brokers** manage the data in the Kafka Topics and distribute it across different **partitions** to ensure data is balanced, fault-tolerant, and scalable. They store the messages until **Consumers** are ready to retrieve them.
3. **Kafka Consumers** retrieve data from Kafka Topics and process it. In the case of our project:
    - The Consumer reads the **network traffic data** from Kafka.
    - It sends the data to a **Machine Learning model** (our **anomaly detection model**) for classification.
    - The model predicts whether the data is **normal** or part of an **attack** (DoS, Fuzzers, etc.).

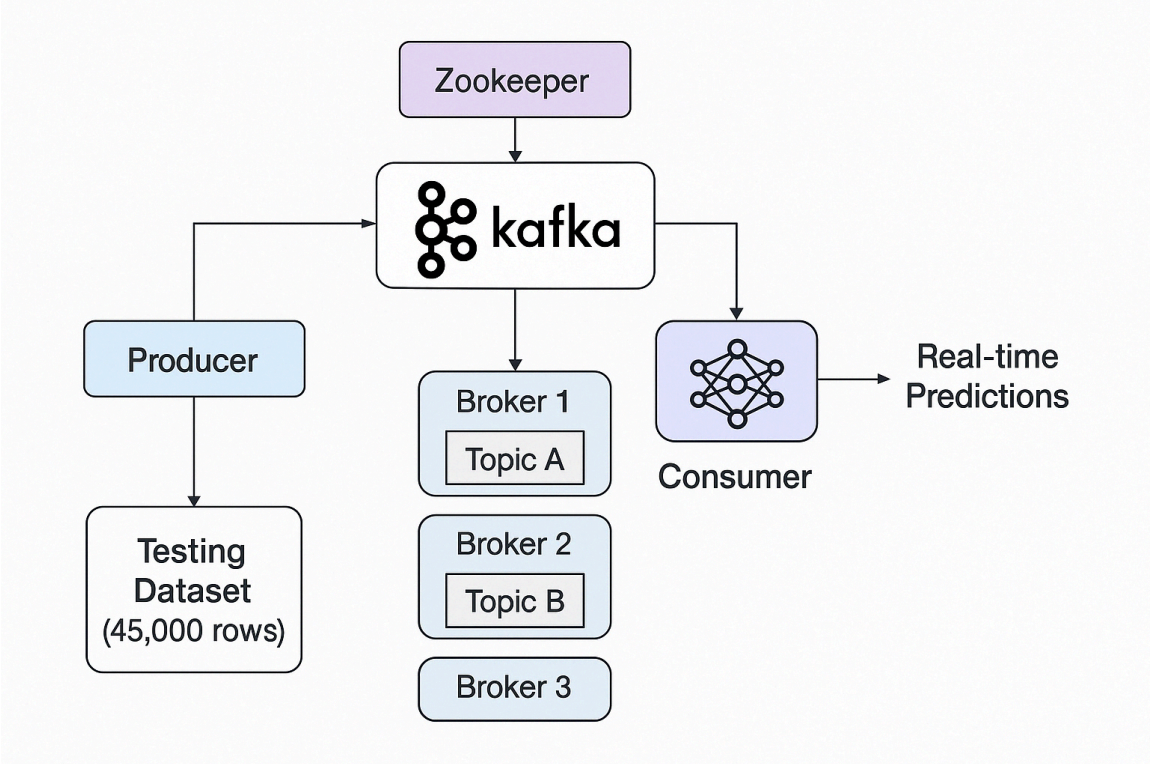
In the Context of Our Project:

- The **Producer** simulates or collects **network traffic** and sends it to Kafka topics.
- The **Broker** (Kafka server) manages and distributes the messages across the cluster.
- The **Consumer** receives the data from Kafka and passes it to the **machine learning model** for **real-time predictions**.

Conclusion:

- **Kafka** is the backbone of the real-time data pipeline, allowing for efficient, scalable, and fault-tolerant handling of network traffic data.
- **Producers** push data (network traffic) into Kafka.
- **Kafka Brokers** manage and store the data.
- **Consumers** retrieve the data, process it, and feed it into the anomaly detection system.

Kafka Architecture



Producer Code Logic:

The producer is responsible for simulating network traffic and streaming the testing dataset through **Kafka**. The purpose is to test the anomaly detection model with simulated **real-time data**.

Producer Code Breakdown:

1. **Spark Session Setup:**

```
spark = SparkSession.builder.appName("ProducerApp").getOrCreat
e()
```

- **Spark** is initialized here because the dataset you're working with is large and stored in **partitioned CSV files**. Spark allows you to handle large datasets efficiently in a distributed fashion.
- **SparkSession** is the entry point for **PySpark**, enablingit to interact with the data using Spark’s functionalities.

2. **Dataset Loading**:

```
directory_path = 'testing_dataset path'
all_files = [os.path.join(directory_path, f) for f in os.listdir(directory_
path) if f.startswith('part-')]
df_list = [pd.read_csv(file) for file in all_files]
df = pd.concat(df_list, ignore_index=True)
```

- loade the **partitioned CSV files** from the directory.
- **Pandas** is used here to read CSV files and **concatenate** all the partitions into a single DataFrame for further processing.

3. **One-Hot Encoding:**

```
python
Copy
with open('onehot_encoder.pk1', 'rb') as f:
    encoder = pickle.load(f)
```

- The **One-Hot Encoder** is loaded from the `pickle` file (`onehot_encoder.pk1`), which was pre-trained earlier.
- We used the pretrained one hot encoded function from the training model script because it essentially had the same structure since we streamed using a testing dataet

```
string_cols = df.select_dtypes(include=['object']).columns.drop('att
ack_cat')
encoded_data = encoder.transform(df[string_cols])
encoded_cols = encoder.get_feature_names_out(string_cols)
encoded_df = pd.DataFrame(encoded_data, columns=encoded_col
s)
df = pd.concat([df.drop(string_cols, axis=1), encoded_df], axis=1)
```

- Remember we still need to encode categorical string values first before one hot encoding

- **String columns** (categorical features) like `proto`, `service`, `state` are selected.
- These categorical columns are **one-hot encoded** using the loaded encoder.
- The encoded features are appended to the DataFrame, and the original categorical columns are dropped.
- We dropped the original columns because they are still string values which wont be used during model prediction

4. **Kafka Producer Setup**:
    - The producer is set up to **send data to Kafka topics**. Here, we'd typically use:

```
producer = KafkaProducer(bootstrap_servers='localhost:9092', val
ue_serializer=lambda v: json.dumps(v).encode('utf-8'))
```

    - **Producer sends the data** to Kafka topics (e.g., `topic1`, `topic2`) in real-time as if it were actual network traffic.

```
producer.send('test-data', value=df.to_dict())
```

    - Each message (network packet or row in your DataFrame) is **sent to Kafka topics** for consumers to consume.

    The **producer** works by **sending partitioned data** as **streaming data** to Kafka, which mimics how network traffic would flow in a real-world scenario.

Consumer Code Logic:

The consumer listens to Kafka topics, retrieves the data, and sends it to the model for **real-time anomaly detection**.

Consumer Code Breakdown:

1. **Model Loading**:

```
p
with open('network_model.pk3', 'rb') as m:
    model = pickle.load(m)
```

- Loads the **trained machine learning model** from the `pickle` file (`network_model.pk3`). This is the model we trained earlier for anomaly detection.
- rb is read binary since model uses binary logic

2. **Kafka Consumer Setup**:

```
consumer = KafkaConsumer(
    'test-data',
    bootstrap_servers='localhost:9092',
    value_deserializer=lambda m: json.loads(m.decode('utf-8'))
)
```

- The **consumer** listens to the Kafka topic (`test-data`) where the **producer** is sending the data.
- It receives the messages in **JSON format**, deserializes them, and prepares them for processing.

3. **Feature Order for Prediction**:

```
feature_order = [ ... ]
```

- A **list of feature names** that matches the order expected by the trained model.
- This is important because the model was trained on this **specific feature order**, and the consumer needs to ensure that it sends data to the model in the same structure.

- If number of features or feature order is different the prediction phase will give an error
- Thats why i used the already pretrained one hot encoder to avoid this
- There were 155 encoded fetaures intotal

4. **Processing Incoming Data**:

```
for message in consumer:
    data = message.value
    if all(feature in data for feature in feature_order):
        features = [data[feature] for feature in feature_order]
        prediction = model.predict([features])
        print(f"AI Prediction Result: {prediction}")
```

- **Data is received** from Kafka.
- The consumer checks if all the required features (from `feature_order`) are available in the incoming message.
- If all features are present, it **extracts the features** and passes them to the model for prediction.
- The **prediction result** (e.g., DoS, Normal, etc.) is printed.

5. **Handling Missing Features**:

```
if not all(feature in data for feature in feature_order):
    print("Missing features in the message")
```

- If any required features are missing in the incoming data, the consumer logs a message saying **“Missing features”**.

Producer & Consumer Interaction

- **Producer**: Generates and sends network traffic data (from the testing dataset) to Kafka topics. It **simulates real-time streaming** by sending the data as if it were coming from a network.

- **Consumer**: Listens to the Kafka topics and processes incoming data. It uses a trained machine learning model to **predict anomalies** (like DoS, Normal traffic, etc.) from the streaming data.

How Kafka and Streaming Work Together in Your System

1. The **Producer** (using Spark for large dataset handling and one-hot encoding) sends partitioned **testing data** to Kafka in real-time, simulating how network traffic would flow.
2. The **Consumer** listens to Kafka topics and uses the **trained model** to make predictions on the incoming data.
3. The Kafka infrastructure ensures that both the **Producer** and **Consumer** can **scale independently**, and **real-time data** can be processed with **minimal delays**.

DEEP EXPLANATION OF PRODUCER AND CONSUMER INTERGRATION

In the **Producer script**, the goal was to simulate **real network traffic** in the absence of access to a real network server. Here's how we achieved that:

- **Dataset Handling**:
  The dataset used in the project is the **testing dataset** — a large dataset stored in **partitioned CSV files**. Since the data is too big to handle as a single file, we used **Spark** to efficiently read and process the data in parallel. We loaded these files, concatenated them into a **single DataFrame**, and prepared it for streaming.
- **One-Hot Encoding**:
  As part of preprocessing, we used a pre-trained **One-Hot Encoder** (stored as a **pickle file**) to encode the **categorical features** in the dataset, like `proto`, `service`, and `state`. The **encoded data** was then appended to the DataFrame, creating a **fully prepared dataset** ready for streaming.
- **Kafka Integration**:
  Since I didn't have access to a real network server, I used **Kafka** to simulate real-time data streaming. I sent the **testing dataset**

through Kafka, just like real network traffic. This setup mimicked **network packets** being streamed through Kafka topics, where they would be consumed by the **consumer** for **real-time anomaly detection**.

In essence, **Kafka** allowed us to send pre-existing data in **real-time**, simulating how network traffic would typically flow through a live system, and giving us a platform to test the model.

The **Consumer script** listens to Kafka topics and processes the incoming simulated network traffic in real-time. Here's how it works:

- **Model Loading**:
  The trained
  machine learning model (stored in a **pickle file**) is loaded into the consumer. This model is responsible for making predictions on incoming network traffic and detecting anomalies, such as DoS attacks, Fuzzers, or other network threats.
- **Kafka Consumer Setup**:
  The
  consumer subscribes to the same Kafka topic (`'test-data'`) to listen for the streaming data. Once data is received, it is **deserialized** and processed into the correct format for the model to consume.
- **Feature Handling**:
  The consumer is designed to
  expect a specific set of features, which were determined during training. These features are **extracted** from each incoming message (which represents a network packet), and they are passed to the model for **prediction**. If any features are missing, the system logs an error, ensuring the model receives the correct input.
- **Prediction**:
  Finally, the model predicts the
  network traffic type (e.g., DoS attack, normal traffic) and outputs the result. This step allows for **real-time anomaly detection** based on the simulated network data.

How the Producer and Consumer Work Together

Here's how the two scripts tie into the full system:

1. **Producer** simulates real-time network traffic by streaming the **testing dataset** (using Kafka) to Kafka topics. This dataset, though not coming from a real network, mimics real-world network data.
2. **Consumer** listens to these Kafka topics, extracts the data, and passes it to the trained **machine learning model** for **real-time anomaly detection**. The model makes predictions based on this simulated network traffic, identifying potential threats or anomalies.

Kafka's Role in This Setup

- **Kafka** serves as the **message broker** that enables real-time communication between the **Producer** (which simulates network traffic) and the **Consumer** (which processes this traffic).
- Even though we don't have access to a live network, **Kafka** allowed us to simulate network traffic efficiently and test the model with real-time data flows.
- This Kafka-based architecture ensures scalability, allowing for the **continuous flow of data** from the producer to the consumer for **real-time predictions**.

Why Kafka Was Key to This Solution

- **Real-Time Data Handling**: Kafka made it possible to simulate a **real-time stream of data**, which is crucial for testing how the anomaly detection system would function in a live network environment.
- **Scalability and Flexibility**: Kafka allows you to scale the system by adding more producers and consumers, meaning the system can handle **large volumes of data** just like it would in production.
- **Asynchronous Data Flow**: Kafka enables a decoupled system where producers and consumers can operate independently, which is ideal for real-time systems

Changes In the main script

- Initially I had trained the model, preprocessed the data and cleaned it

- But there were many challenges and logic errors i faced and had to makes some changes in the main script
- I called it the nadt2.py script
- Changes are explained and listed as below:

Dropping the `proto-index` Feature

While working on the dataset, I noticed that the `proto-index` feature, which was present in the **training dataset**, was **non-existent in the testing dataset**. This discrepancy between the datasets caused an issue because the model was expecting the same features for both training and testing, but the absence of the `proto-index` feature in the testing data led to errors.

Solution:

To resolve this, I decided to **drop the** `proto-index` **feature** from the training dataset to ensure that both the training and testing datasets had the **same set of features**. This change was essential for maintaining consistency between the two datasets, allowing the model to function properly without feature mismatches.

Replacing Print Statements with Method Calls

In the initial stages, I was using **print statements** to invoke methods in the classes. However, I faced issues where the methods weren't being properly executed. The problem stemmed from the fact that **print statements** don’t invoke methods in the way that calling a method directly does. While print statements show output, they don't necessarily execute the method's underlying logic. Initially I was using them because I wanted to visualize the output of every execution

Issue:

The **print statements** I was using made it look like the methods were being executed, but they weren’t performing the necessary operations.

Solution:

I removed all **print statements** and ensured that methods were properly **called** to execute their logic without printing intermediate results

unnecessarily. This allowed the methods to work as intended, ensuring that the data preprocessing and model training were done properly.

Reusing the One-Hot Encoder

Since both the **training** and **testing datasets** were similar (though not identical), I needed to ensure that the encoding process was consistent across both datasets. To achieve this, I used **Pickle** to **load** the **One-Hot Encoder** that had been trained on the training dataset. This way, I could use the same encoder to transform both datasets.

Why this was necessary:

If different encoders were used for the training and testing datasets, the features would be encoded differently, leading to inconsistencies and errors. By reusing the same encoder, I ensured that the **categorical features** were encoded in the same way for both datasets.

Saving the Model with Pickle for Future Use

Once the model was trained, I **saved it using Pickle library in python** to make it reusable. This step was crucial for the **consumer** script, which needed to load the trained model to make predictions on new data. Saving the model allowed me to store it in a file (`network_model.pk3`), which could then be loaded and used by the consumer.

Why I saved the model:

Saving the model ensures that we don't need to retrain it every time. The **Pickle file** holds the trained model's parameters and structure, making it possible to **load and use the model** for real-time predictions in the **consumer script** without retraining.

---

5. Addressing the Feature Mismatch Error

One of the most persistent issues I faced was a **feature mismatch error** during the real time testing phase. The error message indicated that the features in the **testing dataset** didn't match those in the **training dataset**. This was due to:

1. The **proto-index feature** being dropped in the training dataset (as it was missing in the testing data).

2. Inconsistent **feature encoding** between training and testing datasets, particularly with categorical variables.

How I fixed it:

- I ensured that **all categorical features were consistently encoded** using the same **One-Hot Encoder**.
- I **dropped any non-matching features**, such as `proto-index`, to guarantee that both datasets had the same feature set.

Once these issues were resolved, the model was able to process the testing data and generate predictions without errors.

Changes to One-Hot Encoding Handling

I faced an issue for over 50 hours and this is one one the reasons why I faced this error. I pointed out that there was an important change in the way the encoded columns were handled, which directly impacted the **feature mismatch error**. I used countless debugging techniques for countless hours especially printing statements after each block of code to navigate the error. That's actually when I found out about the proto-index issue which was also causing this Here's the breakdown of this:

Original Code:

```
self.df = pd.concat([self.df.drop(string_cols, axis=1), encoded_df], axis
=1)
```

In the **original code**, I dropped the **original categorical columns** (`string_cols`) and then concatenated the **encoded columns** (`encoded_df`) with the remaining dataset. This approach was good but caused problems when it came to handling the **target column** (`attack_cat`) or maintaining column order properly.

Updated Code:

```
self.df = pd.concat([encoded_df, self.df['attack_cat']], axis=1)
```

In the **updated code**, instead of dropping the categorical columns and directly concatenating the encoded ones, you took a more controlled approach:

- **Added the target column** (`attack_cat`) directly into the **encoded DataFrame**, ensuring that the model would still have the correct target feature when making predictions.

- **Feature Mismatch**: In the original code, after encoding, the categorical features were dropped from the original DataFrame. This **dropped the target column** (`attack_cat`) in some cases, which could lead to an issue where the feature space is different between the training and testing datasets, causing the **"different features" error**.
- **Consistent Data**: By keeping the `attack_cat` column intact and adding it explicitly to the encoded data, the **training data** now has the correct structure with both the encoded features and the target column, ensuring that the model works as intended.
- **Fix for the Error**: This change solved the error you were facing with **different features** during testing, as the structure of the DataFrame was now consistent.

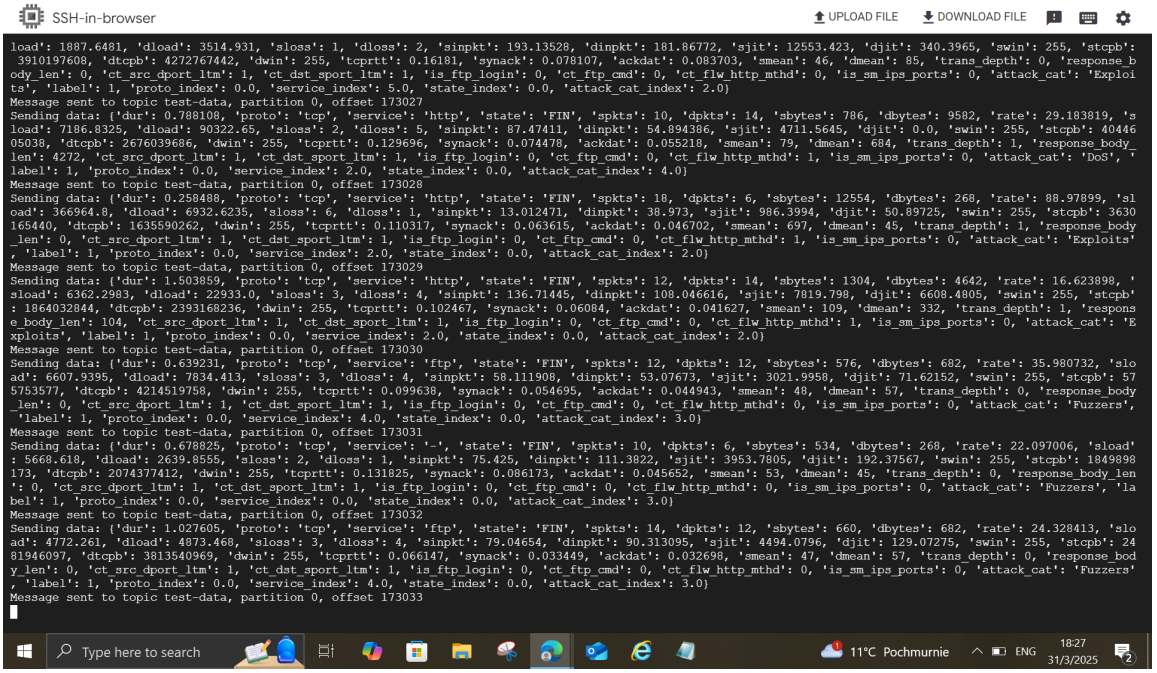
Summary of Key Changes

- **Dropped the** `proto-index` **feature** from the training dataset to ensure consistency with the testing dataset.
- **Replaced print statements** with direct **method calls** to ensure proper method execution.
- **Reused the One-Hot Encoder** for both training and testing datasets to maintain consistent feature encoding.
- **Saved the trained model** using Pickle for future use in the consumer script.

- **Fixed the feature mismatch error** by ensuring both training and testing datasets had the same set of feature

OUTPUTS

Kafka messenger output:



- Kafka messenger output with features being sent directly to the topics

AI Algorithm Prediction Output:

- This is the output produced after running the consumer script with the trained model
- Received messages from the topics that was sent by the producer
- AI Prediction result on the last line
- There is a warning for feature names it was because I used raw arrays when making the predictions which doesn't affect the performance of the model

This document captures the core foundation of the detection software, from concept to real time prediction using Kafka and AI logic.

While this version focuses on the system architecture, preprocessing, and model behavior, further development and documentation are in progress.

Upcoming phases will cover full-scale deployment using Docker and Kubernetes, database integration for log storage, and the development of an interactive dashboard for visualizing network anomalies in real time.

For more updates , connect with me on my linkedln:

https://www.linkedin.com/in/marvellous-chitenga