Bilel RAHMOUNI

# LAB 3 - Classification with Logistic Regression, KNN, and MLP on NSL-KDD Dataset

### Report for NSL-KDD Dataset Analysis

This report covers the process, results, and analysis of three machine learning models: **Logistic Regression**, **K-Nearest Neighbors (KNN)**, and **Multi-Layer Perceptron (MLP)** applied to the **NSL-KDD dataset**, which is a dataset used for network intrusion detection.

### 1. Objective

The main goal of this analysis was to detect anomalies (network intrusions) in the NSL-KDD dataset using three different machine learning models:

- **Logistic Regression**
- **K-Nearest Neighbors (KNN)**
- **Multi-Layer Perceptron (MLP)**

Each model was evaluated based on its accuracy, precision, recall, F1-score, and confusion matrix.

## 2. Data Preprocessing

### 2.1 Data Loading

The **NSL-KDD dataset** consists of both normal network traffic and attack types. The data was loaded into pandas DataFrames (`train_df` and `test_df`). The dataset contains both numerical and categorical features, which required different preprocessing steps:

- **Categorical features**: Protocol type (`tcp`, `udp`, `icmp`), service (`http`, `smtp`, etc.), and flag were non-numeric.

- **Numerical features**: These include fields like duration, source bytes, destination bytes, and many others.

| | duration | protocol_type | service | flag | src_bytes | dst_bytes | land | wrong_fragment | urgent | hot | ... | dst_host_same_srv_rate | dst_host_diff_srv_rate | dst_host_same_src_port_rate | dst_host_srv_diff_host_rate | dst_ho |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | tcp | private | REJ | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0.04 | 0.06 | 0.00 | 0.00 | |
| 1 | 0 | tcp | private | REJ | 0 | 0 | 0 | 0 | 0 | 0 | ... | 0.00 | 0.06 | 0.00 | 0.00 | |
| 2 | 2 | tcp | ftp_data | SF | 12983 | 0 | 0 | 0 | 0 | 0 | ... | 0.61 | 0.04 | 0.61 | 0.02 | |
| 3 | 0 | icmp | eco_i | SF | 20 | 0 | 0 | 0 | 0 | 0 | ... | 1.00 | 0.00 | 1.00 | 0.28 | |
| 4 | 1 | tcp | telnet | RSTO | 0 | 15 | 0 | 0 | 0 | 0 | ... | 0.31 | 0.17 | 0.03 | 0.02 | |

5 rows × 43 columns

### 2.2 Label Encoding

The target variable (`label`) was transformed into binary values:

- **0** for normal traffic.
- **1** for any kind of attack (anomaly).

```
# Check the unique labels in the training and testing sets
train_labels = train_df['label'].unique()
test_labels = test_df['label'].unique()

# Print the unique labels in both datasets to compare
print("Unique labels in training set:", train_labels)
print("Unique labels in test set:", test_labels)

# Check for any unseen labels in the test set
unseen_labels = set(test_labels) - set(train_labels)
print("Unseen labels in test set:", unseen_labels)
```

### 2.3 One-Hot Encoding

To handle the categorical features (`protocol_type`, `service`, and `flag`), **One-Hot Encoding** was applied. This converted each categorical value into a set of binary columns, making them usable by machine learning algorithms.

```
# One-Hot Encode the categorical columns: 'protocol_type', 'service', 'flag'
X_train = pd.get_dummies(X_train, columns=['protocol_type', 'service', 'flag'])
X_test = pd.get_dummies(X_test, columns=['protocol_type', 'service', 'flag'])

# Ensure both train and test sets have the same columns after encoding
X_train, X_test = X_train.align(X_test, join='left', axis=1, fill_value=0)

# Verify the shape after One-Hot Encoding
print(X_train.shape)
print(X_test.shape)
```

```
(125973, 123)
(22544, 123)
```

### 2.4 Feature Scaling

All numerical features were standardized using **StandardScaler**, ensuring that they were on a common scale with a mean of 0 and a standard deviation of 1.

```
          0         1         2         3         4         5         6   \
0 -0.110249 -0.007679 -0.004919 -0.014089 -0.089486 -0.007736 -0.095076
1 -0.110249 -0.007737 -0.004919 -0.014089 -0.089486 -0.007736 -0.095076
2 -0.110249 -0.007762 -0.004919 -0.014089 -0.089486 -0.007736 -0.095076
3 -0.110249 -0.007723 -0.002891 -0.014089 -0.089486 -0.007736 -0.095076
4 -0.110249 -0.007728 -0.004814 -0.014089 -0.089486 -0.007736 -0.095076

          7         8         9  ...       113      114       115       116  \
0 -0.027023 -0.809262 -0.011664  ... -0.312889 -0.11205 -0.028606 -0.139982
1 -0.027023 -0.809262 -0.011664  ... -0.312889 -0.11205 -0.028606 -0.139982
2 -0.027023 -0.809262 -0.011664  ... -0.312889 -0.11205 -0.028606 -0.139982
3 -0.027023  1.235694 -0.011664  ... -0.312889 -0.11205 -0.028606 -0.139982
4 -0.027023  1.235694 -0.011664  ... -0.312889 -0.11205 -0.028606 -0.139982

        117       118       119       120       121       122
0 -0.618438 -0.053906 -0.031767 -0.019726  0.825150 -0.046432
1 -0.618438 -0.053906 -0.031767 -0.019726  0.825150 -0.046432
2  1.616978 -0.053906 -0.031767 -0.019726 -1.211901 -0.046432
3 -0.618438 -0.053906 -0.031767 -0.019726  0.825150 -0.046432
4 -0.618438 -0.053906 -0.031767 -0.019726  0.825150 -0.046432

[5 rows x 123 columns]
```

## 3. Model Training and Evaluation

### 3.1 Logistic Regression

Logistic Regression is a simple yet effective algorithm for binary classification problems. The model was trained using the preprocessed dataset and evaluated on the test set.

```python
# Importing necessary libraries
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Initialize the Logistic Regression model
logreg_model = LogisticRegression(random_state=42, max_iter=1000)

# Train the model on the PCA-transformed training data
logreg_model.fit(X_train_pca, y_train)

# Model trained successfully
print("Logistic Regression model trained successfully!")
```

### 3.2 K-Nearest Neighbors (KNN)

KNN is a non-parametric model that classifies data points based on the majority class of their "k" nearest neighbors. We used k=5 neighbors for this analysis.

```
# Import necessary libraries
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Initialize the KNN model
knn_model = KNeighborsClassifier(n_neighbors=5)  # You can change 'n_neighbors' to try different values

# Train the KNN model on the PCA-transformed training data
knn_model.fit(X_train_pca, y_train)

# Model trained successfully
print("K-Nearest Neighbors model trained successfully!")
```

### 3.3 Multi-Layer Perceptron (MLP)

MLP is a neural network model that uses backpropagation to learn weights. We used one hidden layer with 100 neurons for the analysis

.

```
[ ] # Import necessary libraries
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Initialize the MLP model
mlp_model = MLPClassifier(hidden_layer_sizes=(100,), max_iter=300, random_state=42)

# Train the MLP model on the PCA-transformed training data
mlp_model.fit(X_train_pca, y_train)

# Model trained successfully
print("Multi-Layer Perceptron model trained successfully!")
```

## 4. Model Performance

The performance of each model was evaluated based on:

- **Accuracy**: The proportion of correct predictions.
- **Precision**: The proportion of predicted anomalies that were actually anomalies.
- **Recall**: The proportion of actual anomalies that were correctly identified.
- **F1-score**: The harmonic mean of precision and recall, giving a balance between the two.
- **Confusion Matrix**: A matrix showing the breakdown of true positives, false positives, true negatives, and false negatives.

### Logistic Regression Results:

- **Accuracy**: 84.62%
- **Precision**: 93%
- **Recall**: 79%
- **F1-Score**: 85%

```
Accuracy: 84.62%
Confusion Matrix:
[[ 8909   802]
 [ 2666 10167]]
Classification Report:
              precision    recall  f1-score   support

           0       0.77      0.92      0.84      9711
           1       0.93      0.79      0.85     12833

    accuracy                           0.85     22544
   macro avg       0.85      0.85      0.85     22544
weighted avg       0.86      0.85      0.85     22544
```

### K-Nearest Neighbors (KNN) Results:

- **Accuracy**: 80.86%
- **Precision**: 96%
- **Recall**: 69%
- **F1-Score**: 80%

```
Accuracy: 80.86%
Confusion Matrix:
[[9335  376]
 [3938 8895]]
Classification Report:
              precision    recall  f1-score   support

           0       0.70      0.96      0.81      9711
           1       0.96      0.69      0.80     12833

    accuracy                           0.81     22544
   macro avg       0.83      0.83      0.81     22544
weighted avg       0.85      0.81      0.81     22544
```

### Multi-Layer Perceptron (MLP) Results:

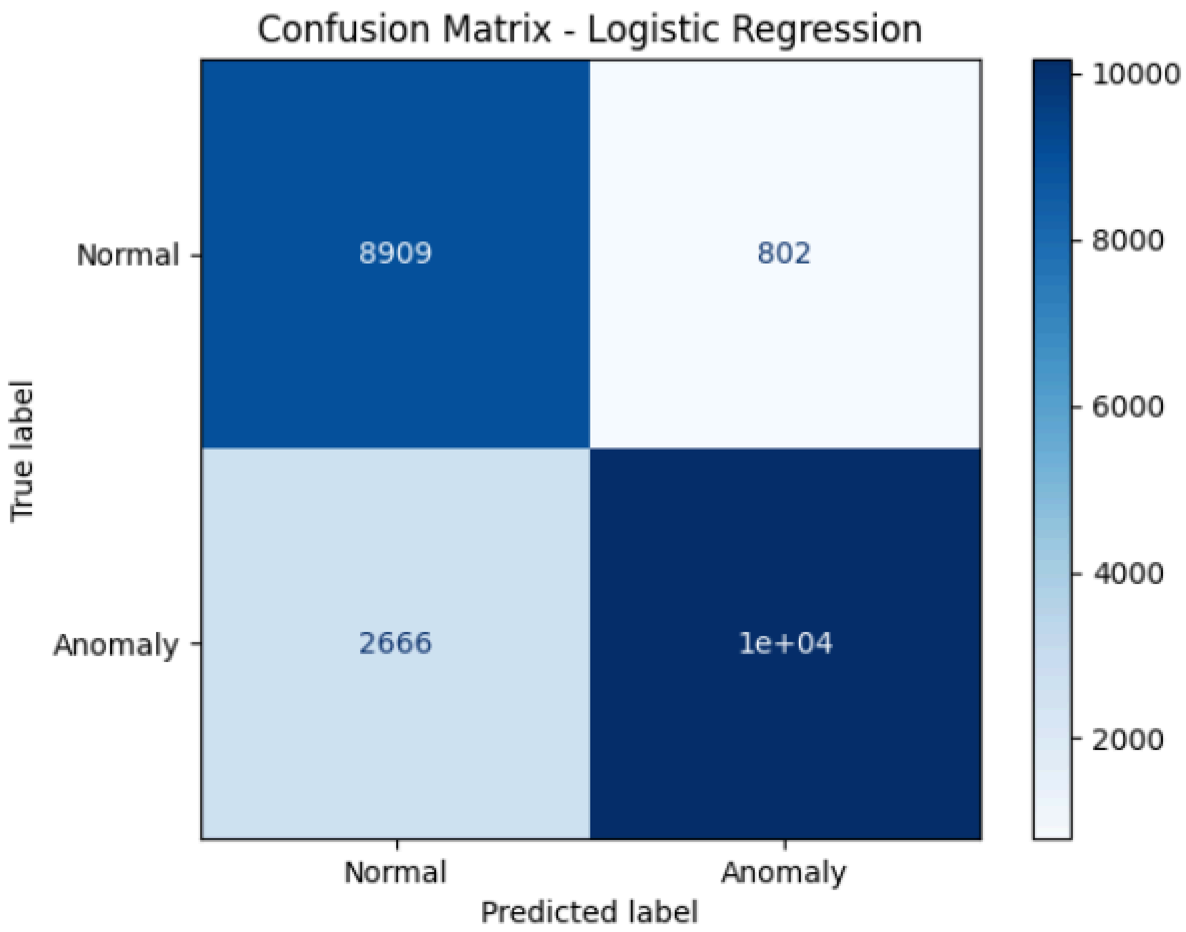
- **Accuracy**: 84.76%
- **Precision**: 93%
- **Recall**: 79%
- **F1-Score**: 86%

```
Accuracy: 84.76%
Confusion Matrix:
[[ 8971   740]
 [ 2696 10137]]
Classification Report:
              precision    recall  f1-score   support

           0       0.77      0.92      0.84      9711
           1       0.93      0.79      0.86     12833

    accuracy                           0.85     22544
   macro avg       0.85      0.86      0.85     22544
weighted avg       0.86      0.85      0.85     22544
```

## 5. Confusion Matrix Analysis

The **confusion matrices** for each model show how well they classified normal and anomaly classes.

**Logistic Regression Confusion Matrix:**



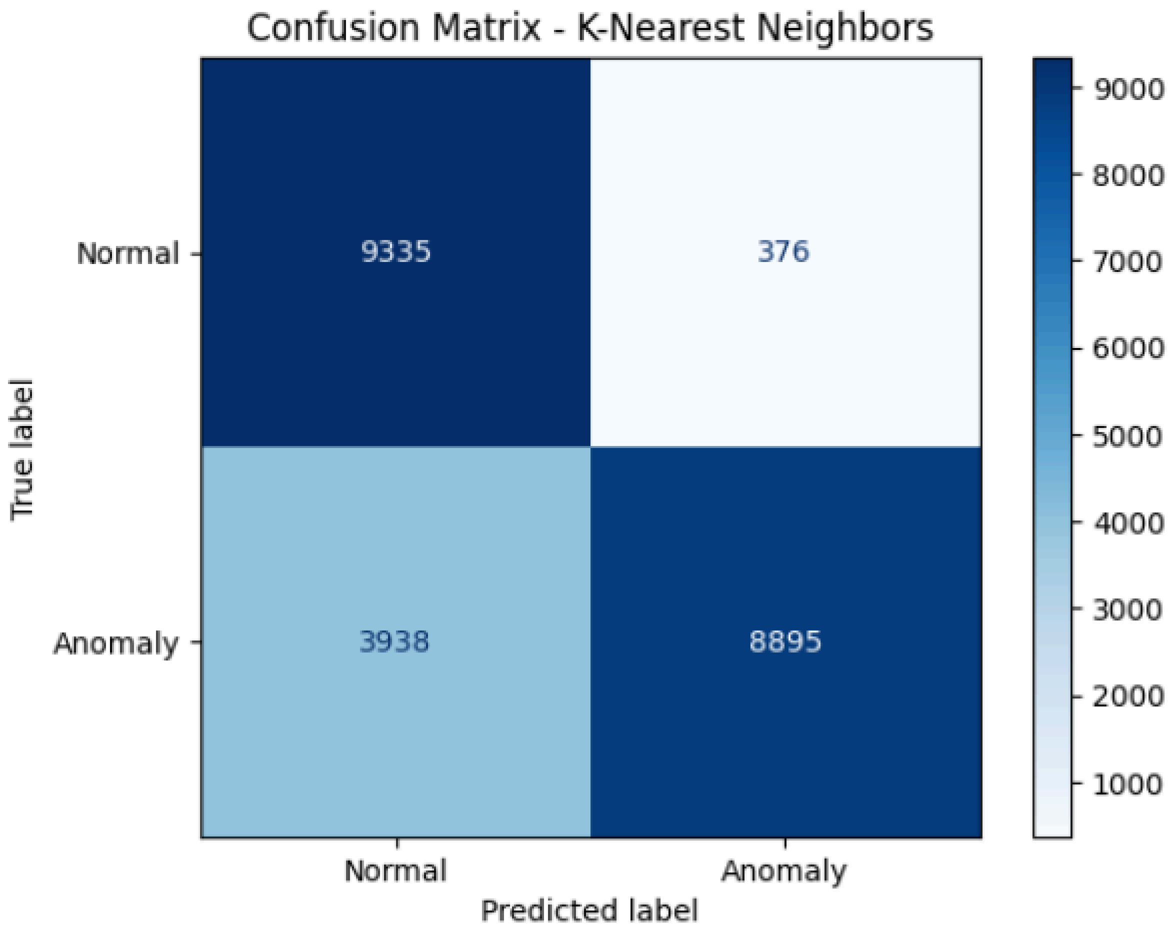
```
[[ 8909   802]
```

8909 True Negatives (normal classified as normal), 802 False Positives (normal misclassified as anomaly)

```
 [ 2666 10167]]
```

2666 False Negatives (anomaly misclassified as normal), 10167 True Positives (anomaly classified as anomaly)

**KNN Confusion Matrix:**




```
[[9335  376]
```

High true negatives, low false positives, but many anomalies are missed (high false negatives)

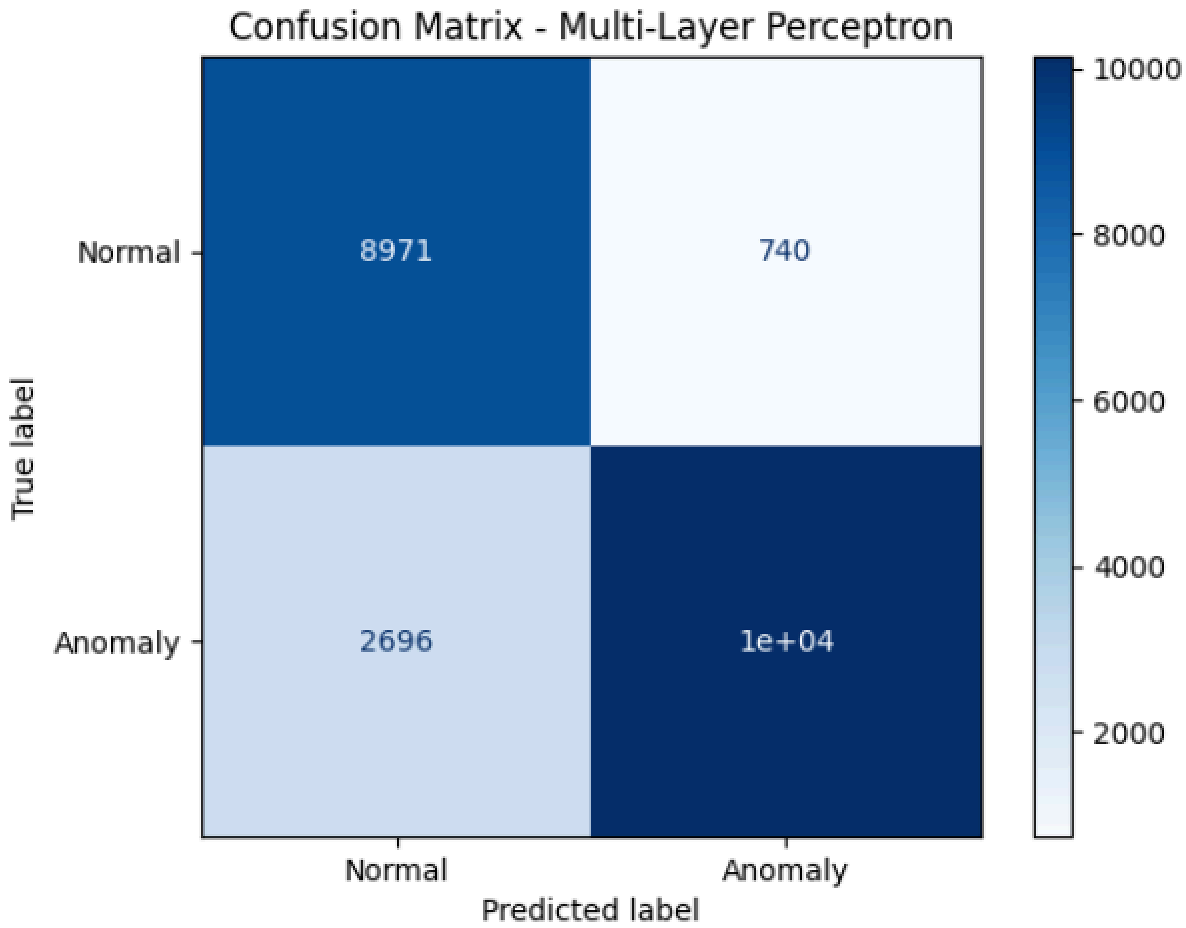
```
 [3938 8895]]
```

**MLP Confusion Matrix:**

```
[[ 8971   740]
```

Balanced performance between normal and anomaly classifications

```
 [ 2696 10137]]
```




## 6. Comparison of Models

| Model | Accuracy | Precision | Recall | F1-Score |
|---|---|---|---|---|
| Logistic Regression | 84.62% | 93% | 79% | 85% |
| K-Nearest Neighbors | 80.86% | 96% | 69% | 80% |
| Multi-Layer Perceptron | 84.76% | 93% | 79% | 86% |

```python
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Logistic Regression Performance
print("=== Logistic Regression ===")
print(f"Accuracy: {accuracy_score(y_test, y_pred):.2f}")
print(f"Precision: {precision_score(y_test, y_pred):.2f}")
print(f"Recall: {recall_score(y_test, y_pred):.2f}")
print(f"F1-Score: {f1_score(y_test, y_pred):.2f}")

# K-Nearest Neighbors (KNN) Performance
print("\n=== K-Nearest Neighbors (KNN) ===")
print(f"Accuracy: {accuracy_score(y_test, y_pred_knn):.2f}")
print(f"Precision: {precision_score(y_test, y_pred_knn):.2f}")
print(f"Recall: {recall_score(y_test, y_pred_knn):.2f}")
print(f"F1-Score: {f1_score(y_test, y_pred_knn):.2f}")

# Multi-Layer Perceptron (MLP) Performance
print("\n=== Multi-Layer Perceptron (MLP) ===")
print(f"Accuracy: {accuracy_score(y_test, y_pred_mlp):.2f}")
print(f"Precision: {precision_score(y_test, y_pred_mlp):.2f}")
print(f"Recall: {recall_score(y_test, y_pred_mlp):.2f}")
print(f"F1-Score: {f1_score(y_test, y_pred_mlp):.2f}")
```

- **Best Overall Model**: **Multi-Layer Perceptron (MLP)** slightly outperformed Logistic Regression in terms of accuracy and F1-score, making it the best model for this task.
- **Best Precision**: **KNN** had the highest precision (96%), meaning it rarely misclassified normal instances as anomalies, but its lower recall (69%) indicates that many anomalies were missed.
- **Best Recall**: **Logistic Regression** and **MLP** both had better recall than KNN, meaning they detected more anomalies.

## 7. Conclusion and Recommendations

- **MLP** and **Logistic Regression** are the most balanced models in terms of accuracy, precision, recall, and F1-score.
- **KNN** had a very high precision but missed many anomalies (lower recall), which might not be ideal for intrusion detection where detecting anomalies is critical.
- **MLP** achieved the highest F1-score and a slightly better accuracy, making it the most effective model for this analysis.