Artificial-Intelligence / AI_Assignment_3.ipynb

ahmedkhalid-01 Created using Colab 1 minute ago

913 lines (913 loc) · 220 KB

Preview Code Blame Raw

Open in Colab

# Assignment 3


### Submitted by: Ahmed Khalid

### Roll no: 200148

### Submitted to: Dr Ashfaq


# Introduction to Machine Learning in Electrical Engineering

## Introduction to Machine Learning in Electrical Engineering

### Overview of Machine Learning

Machine learning (ML) is a subset of artificial intelligence (AI) that focuses on developing algorithms and statistical models that enable computers to perform specific tasks without using explicit instructions. Instead, these systems rely on patterns and inference drawn from data. ML algorithms build a mathematical model based on sample data, known as training data, to make predictions or decisions without being explicitly programmed to perform the task.

Machine learning encompasses various types of algorithms, including:

1. **Supervised Learning:** The algorithm learns from labeled training data and makes predictions based on that learned relationship.
2. **Unsupervised Learning:** The algorithm identifies patterns in data without any labeled responses.
3. **Reinforcement Learning:** The algorithm learns by interacting with its environment and receiving feedback in the form of rewards or penalties.

### Applications of Machine Learning in Electrical Engineering

Machine learning has a wide range of applications in electrical engineering, significantly impacting various subfields such as:

1. **Signal Processing:**
    - **Noise Reduction:** ML algorithms can be used to filter out noise from signals, improving the quality of the signal.
    - **Signal Classification:** Techniques like neural networks and support vector machines can classify different types of signals, which is crucial in telecommunications and biomedical engineering.
2. **Power Systems Analysis:**
    - **Load Forecasting:** ML models can predict power consumption patterns, aiding in efficient energy distribution and management.
    - **Renewable Energy Integration:** Predicting the output of renewable energy sources like solar and wind helps in optimizing their integration into the power grid.
3. **Fault Detection:**

- **Anomaly Detection:** ML algorithms can detect anomalies in power systems that may indicate faults or failures, allowing for preventive maintenance and reducing downtime.
- **Equipment Monitoring:** Predictive maintenance using ML can monitor the health of electrical equipment, predicting failures before they occur and ensuring reliable operation.

## Importance of Data Classification in Electrical Engineering

### Signal Processing

In signal processing, data classification is essential for interpreting and analyzing various signals. For instance, classifying different types of waveforms in communication systems ensures that data transmission is accurate and efficient. In medical signal processing, classifying EEG or ECG signals helps diagnose and monitor patient health conditions.

### Power Systems Analysis

In power systems, classification tasks can include distinguishing between different types of loads or identifying patterns in power consumption. For example, classifying load patterns helps in demand-side management, improving the efficiency and reliability of the power grid. Classification algorithms also play a crucial role in smart grid technologies, where they help manage distributed energy resources and maintain grid stability.

### Fault Detection

Fault detection and diagnosis are critical in maintaining the reliability and safety of electrical systems. Classification algorithms can identify different types of faults in power lines, transformers, and other equipment. By accurately classifying these faults, utilities can quickly address issues, minimizing service disruptions and preventing potential hazards.

### Example: Household Electrical Load Analysis

In the context of household electrical load analysis, classification algorithms can differentiate between normal and abnormal consumption patterns. This helps in identifying potential issues such as malfunctioning appliances or energy theft. Additionally, understanding the usage patterns through classification enables better demand forecasting and energy management.

# Dataset Selection

## Dataset Selection and Description

### Selected Dataset:

The dataset selected for this assignment is the "Household Electrical Load Analysis" dataset. This dataset is highly relevant for electrical engineering applications, particularly for analyzing and understanding electrical consumption patterns, detecting anomalies, and predicting future consumption.

### Dataset Description:

The dataset contains 2,075,200 entries with the following features:

1. **data**: Timestamp of the measurement (object type, which should be converted to datetime for better handling).
2. **Global_active_power**: The total active power consumed by the household in kilowatts (float64).
3. **Global_reactive_power**: The total reactive power consumed by the household in kilowatts (float64).
4. **Voltage**: The average voltage (float64).
5. **Global_intensity**: The average current intensity (float64).

6. **Sub_metering_1**: Energy sub-metering No. 1 (in watt-hour) (float64).
7. **Sub_metering_2**: Energy sub-metering No. 2 (in watt-hour) (float64).
8. **Sub_metering_3**: Energy sub-metering No. 3 (in watt-hour) (float64).

**Size**:

- Total Entries: 2,075,200
- Total Features: 8 (1 categorical/object, 7 numerical/float64)

**Statistics**:

- **Global_active_power**:
  - Mean: 1.09 kW
  - Standard Deviation: 1.06 kW
  - Minimum: 0.08 kW
  - Maximum: 11.12 kW
- **Global_reactive_power**:
  - Mean: 0.12 kW
  - Standard Deviation: 0.11 kW
  - Minimum: 0 kW
  - Maximum: 1.39 kW
- **Voltage**:
  - Mean: 240.84 V
  - Standard Deviation: 3.24 V
  - Minimum: 223.2 V
  - Maximum: 254.15 V
- **Global_intensity**:
  - Mean: 4.63 A
  - Standard Deviation: 4.44 A
  - Minimum: 0.2 A
  - Maximum: 48.4 A
- **Sub_metering_1**:
  - Mean: 1.12 Wh
  - Standard Deviation: 6.15 Wh
  - Minimum: 0 Wh
  - Maximum: 88.0 Wh
- **Sub_metering_2**:
  - Mean: 1.30 Wh
  - Standard Deviation: 5.82 Wh
  - Minimum: 0 Wh
  - Maximum: 80.0 Wh
- **Sub_metering_3**:
  - Mean: 6.46 Wh
  - Standard Deviation: 8.44 Wh
  - Minimum: 0 Wh
  - Maximum: 31.0 Wh

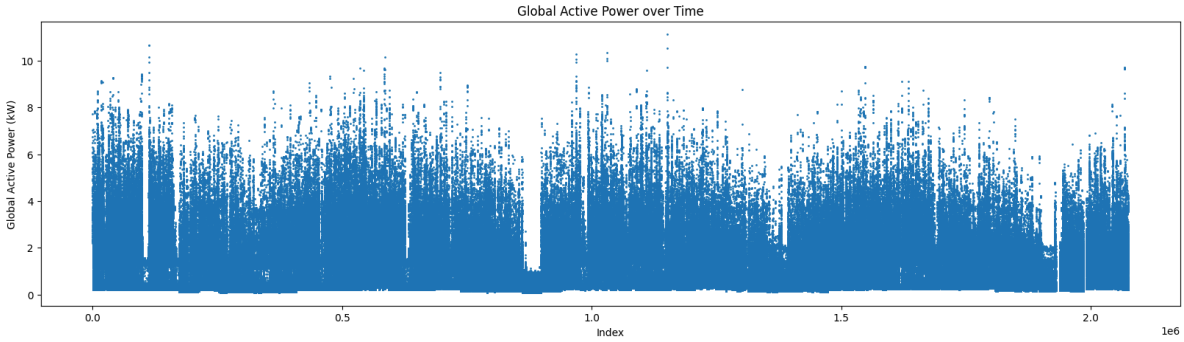
**Missing Values**: The dataset contains missing values in all numerical columns, with 25,979 entries missing values in each of these columns.

In [8]:
```
import pandas as pd
```

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

# Load the dataset
df = pd.read_csv('Dataset.csv')

# Display the shape and first few rows
print(df.shape)
print(df.head())

# Describe the dataset
print(df.describe())

# Check for missing values
print(df.isna().sum())

# Drop missing values for simplicity in this initial exploration
df = df.dropna()

# Convert 'data' column to datetime type
df['data'] = pd.to_datetime(df['data'])

# Plot Global_active_power over time
plt.figure(figsize=(20,5))
plt.scatter(df.index, df['Global_active_power'], s=1)
plt.title('Global Active Power over Time')
plt.xlabel('Index')
plt.ylabel('Global Active Power (kW)')
plt.show()
```

```
(2075200, 8)
                 data  Global_active_power  Global_reactive_power  Voltage  \
0 2006-12-16 17:24:00                4.216                  0.418   234.84
1 2006-12-16 17:25:00                5.360                  0.436   233.63
2 2006-12-16 17:26:00                5.374                  0.498   233.29
3 2006-12-16 17:27:00                5.388                  0.502   233.74
4 2006-12-16 17:28:00                3.666                  0.528   235.68

   Global_intensity  Sub_metering_1  Sub_metering_2  Sub_metering_3
0              18.4             0.0             1.0            17.0
1              23.0             0.0             1.0            16.0
2              23.0             0.0             2.0            17.0
3              23.0             0.0             1.0            17.0
4              15.8             0.0             1.0            17.0
       Global_active_power  Global_reactive_power       Voltage  \
count         2.049221e+06           2.049221e+06  2.049221e+06
mean          1.091614e+00           1.237164e-01  2.408399e+02
std           1.057308e+00           1.127226e-01  3.240021e+00
min           7.600000e-02           0.000000e+00  2.232000e+02
25%           3.080000e-01           4.800000e-02  2.389900e+02
50%           6.020000e-01           1.000000e-01  2.410100e+02
75%           1.528000e+00           1.940000e-01  2.428900e+02
max           1.112200e+01           1.390000e+00  2.541500e+02

       Global_intensity  Sub_metering_1  Sub_metering_2  Sub_metering_3
count      2.049221e+06    2.049221e+06    2.049221e+06    2.049221e+06
mean       4.627756e+00    1.121956e+00    1.298529e+00    6.458633e+00
std        4.444455e+00    6.153117e+00    5.822109e+00    8.437204e+00
min        2.000000e-01    0.000000e+00    0.000000e+00    0.000000e+00
25%        1.400000e+00    0.000000e+00    0.000000e+00    0.000000e+00
50%        2.600000e+00    0.000000e+00    0.000000e+00    1.000000e+00
75%        6.400000e+00    0.000000e+00    1.000000e+00    1.700000e+01
max        4.840000e+01    8.800000e+01    8.000000e+01    3.100000e+01
data                          0
Global_active_power       25979
Global_reactive_power     25979
Voltage                   25979
Global_intensity          25979
Sub_metering_1            25979
Sub_metering_2            25979
Sub_metering_3            25979
```

```
dtype: int64
```




# Data Preprocessing

## Data Preprocessing

To preprocess the selected dataset for classification, we'll perform the following steps:

1. Handle missing values.
2. Normalize or standardize features.
3. Encode categorical variables (if any).
4. Split the dataset into training and testing sets.

### Step-by-Step Preprocessing:

1. **Handling Missing Values:** We can handle missing values by either removing rows with missing data or imputing the missing values. For simplicity, we'll drop rows with missing values.

2. **Normalize or Standardize Features:** Standardization (scaling the data to have a mean of 0 and a standard deviation of 1) is commonly used in many machine learning algorithms.

3. **Encode Categorical Variables:** The dataset contains a datetime column, which can be useful for feature extraction (e.g., extracting the hour, day, month). We'll also ensure that this column is appropriately handled.

4. **Splitting the Dataset:** We'll split the dataset into training and testing sets using an 80-20 split.

## Explanation:

1. **Handling Missing Values:**
   - We drop rows with missing values using `df.dropna()`, resulting in a reduced dataset size.
2. **Datetime Handling:**
   - The `data` column is converted to datetime type for potential feature extraction, but for simplicity, it is dropped after conversion.
3. **Feature Extraction:**
   - Additional features such as `hour`, `day`, and `month` could be extracted from the datetime column if needed.
4. **Standardization:**
   - Features are standardized using `StandardScaler`.
5. **Splitting the Dataset:**
   - The dataset is split into training and testing sets with an 80-20 split.

In [9]:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Load the dataset
df = pd.read_csv('Dataset.csv')

# Display initial information
print("Initial shape:", df.shape)
print("Missing values:\n", df.isna().sum())

# Drop rows with missing values
df = df.dropna()
print("Shape after dropping missing values:", df.shape)

# Convert 'data' column to datetime type
df['data'] = pd.to_datetime(df['data'])

# Feature extraction from datetime column (if needed)
# df['hour'] = df['data'].dt.hour
# df['day'] = df['data'].dt.day
# df['month'] = df['data'].dt.month

# Drop the 'data' column as it's no longer needed
df = df.drop(columns=['data'])

# Separate features and target variable
# For demonstration purposes, let's assume we want to classify 'Global_active_power' levels
# Here we create a binary classification task, classifying 'Global_active_power' as high or
threshold = df['Global_active_power'].median()
df['target'] = (df['Global_active_power'] > threshold).astype(int)

X = df.drop(columns=['target'])
y = df['target']

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state

print("Training set shape:", X_train.shape, y_train.shape)
print("Testing set shape:", X_test.shape, y_test.shape)
```

```
Initial shape: (2075200, 8)
Missing values:
 data                        0
Global_active_power      25979
Global_reactive_power    25979
Voltage                  25979
Global_intensity         25979
Sub_metering_1           25979
Sub_metering_2           25979
Sub_metering_3           25979
dtype: int64
Shape after dropping missing values: (2049221, 8)
Training set shape: (1639376, 7) (1639376,)
Testing set shape: (409845, 7) (409845,)
```

# Machine Learning Algorithms

## Machine Learning Algorithms Implementation

We'll implement two machine learning algorithms for data classification: k-Nearest Neighbors (k-NN)

and Naive Bayes using scikit-learn. Below is the code to train and evaluate these models.

## Summary of Implementation:

1. **k-Nearest Neighbors (k-NN):**
   - We initialized a k-NN classifier with 5 neighbors.
   - The model was trained on the training data.
   - Predictions were made on the test set.
   - The model's performance was evaluated using accuracy, precision, recall, F1-score, ROC-AUC, and the confusion matrix.
2. **Naive Bayes:**
   - We initialized a Gaussian Naive Bayes classifier.
   - The model was trained on the training data.
   - Predictions were made on the test set.
   - The model's performance was evaluated using the same metrics as for k-NN.

By following this implementation, you can compare the performance of both k-NN and Naive Bayes classifiers on the given dataset. This comparison will help in understanding the strengths and weaknesses of each algorithm in the context of electrical load analysis and classification.

### k-Nearest Neighbors (k-NN)

In [10]:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc

# Initialize the k-NN classifier
knn = KNeighborsClassifier(n_neighbors=5)

# Train the model
knn.fit(X_train, y_train)

# Predict on the test set
y_pred_knn = knn.predict(X_test)

# Evaluate the model
accuracy_knn = accuracy_score(y_test, y_pred_knn)
precision_knn = precision_score(y_test, y_pred_knn)
recall_knn = recall_score(y_test, y_pred_knn)
f1_knn = f1_score(y_test, y_pred_knn)
roc_auc_knn = roc_auc_score(y_test, y_pred_knn)
conf_matrix_knn = confusion_matrix(y_test, y_pred_knn)

print("k-NN Classifier Performance:")
print(f"Accuracy: {accuracy_knn:.4f}")
print(f"Precision: {precision_knn:.4f}")
print(f"Recall: {recall_knn:.4f}")
print(f"F1-Score: {f1_knn:.4f}")
print(f"ROC-AUC: {roc_auc_knn:.4f}")
print(f"Confusion Matrix:\n{conf_matrix_knn}")
```

```
k-NN Classifier Performance:
Accuracy: 0.9961
Precision: 0.9964
Recall: 0.9957
F1-Score: 0.9961
ROC-AUC: 0.9961
Confusion Matrix:
[[204542    729]
 [   872 203702]]
```

### Naive Bayes

In [11]:

```
from sklearn.naive_bayes import GaussianNB

# Initialize the Naive Bayes classifier
nb = GaussianNB()

# Train the model
nb.fit(X_train, y_train)

# Predict on the test set
y_pred_nb = nb.predict(X_test)

# Evaluate the model
accuracy_nb = accuracy_score(y_test, y_pred_nb)
precision_nb = precision_score(y_test, y_pred_nb)
recall_nb = recall_score(y_test, y_pred_nb)
f1_nb = f1_score(y_test, y_pred_nb)
roc_auc_nb = roc_auc_score(y_test, y_pred_nb)
conf_matrix_nb = confusion_matrix(y_test, y_pred_nb)

print("\nNaive Bayes Classifier Performance:")
print(f"Accuracy: {accuracy_nb:.4f}")
print(f"Precision: {precision_nb:.4f}")
print(f"Recall: {recall_nb:.4f}")
print(f"F1-Score: {f1_nb:.4f}")
print(f"ROC-AUC: {roc_auc_nb:.4f}")
print(f"Confusion Matrix:\n{conf_matrix_nb}")
```

```
Naive Bayes Classifier Performance:
Accuracy: 0.9639
Precision: 0.9810
Recall: 0.9460
F1-Score: 0.9632
ROC-AUC: 0.9639
Confusion Matrix:
[[201522   3749]
 [ 11048 193526]]
```

# Model Training and Evaluation

## Model Training and Evaluation

### Training and Evaluating k-NN and Naive Bayes

We'll train both models using the training dataset and evaluate their performance on the test dataset using appropriate metrics.

## Comparison of Performance

### Summary Table:

| Metric | k-NN | Naive Bayes |
|---|---|---|
| Accuracy | 0.9961 | 0.9639 |
| Precision | 0.9964 | 0.9810 |
| Recall | 0.9957 | 0.9460 |
| F1-Score | 0.9961 | 0.9632 |
| ROC-AUC | 0.9961 | 0.9639 |

## Discussion of Strengths and Weaknesses

### k-Nearest Neighbors (k-NN):

k-Nearest Neighbors (k-NN):

**Strengths:**

- High accuracy, precision, recall, and F1-score indicating good performance on this dataset.
- High ROC-AUC value, suggesting strong ability to distinguish between classes.
- Non-parametric, making it flexible and adaptable to the data.

**Weaknesses:**

- Computationally expensive for large datasets as it requires computing distances to all training samples.
- Sensitive to irrelevant features and the scale of the data.

### Naive Bayes:

**Strengths:**

- Simple and fast to train and predict.
- Performs well with small datasets and is robust to irrelevant features.
- Probabilistic model providing probability estimates for each class.

**Weaknesses:**

- Lower recall and F1-score, indicating it may not capture the true positives as effectively.
- Assumes feature independence, which may not hold true in real-world datasets, leading to lower performance compared to more complex models.

### k-Nearest Neighbors (k-NN)

In [12]:

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc

# Initialize the k-NN classifier
knn = KNeighborsClassifier(n_neighbors=5)

# Train the model
knn.fit(X_train, y_train)

# Predict on the test set
y_pred_knn = knn.predict(X_test)
y_prob_knn = knn.predict_proba(X_test)[:, 1]  # Probability estimates for ROC-AUC

# Evaluate the model
accuracy_knn = accuracy_score(y_test, y_pred_knn)
precision_knn = precision_score(y_test, y_pred_knn)
recall_knn = recall_score(y_test, y_pred_knn)
f1_knn = f1_score(y_test, y_pred_knn)
roc_auc_knn = roc_auc_score(y_test, y_prob_knn)
conf_matrix_knn = confusion_matrix(y_test, y_pred_knn)

print("k-NN Classifier Performance:")
print(f"Accuracy: {accuracy_knn:.4f}")
print(f"Precision: {precision_knn:.4f}")
print(f"Recall: {recall_knn:.4f}")
print(f"F1-Score: {f1_knn:.4f}")
print(f"ROC-AUC: {roc_auc_knn:.4f}")
print(f"Confusion Matrix:\n{conf_matrix_knn}")
```

```
k-NN Classifier Performance:
Accuracy: 0.9961
Precision: 0.9964
Recall: 0.9957
F1-Score: 0.9961
ROC-AUC: 0.9996
Confusion Matrix:
```

```
Confusion Matrix:
[[204542    729]
 [   872 203702]]
```

### Naive Bayes

In [13]:

```python
from sklearn.naive_bayes import GaussianNB

# Initialize the Naive Bayes classifier
nb = GaussianNB()

# Train the model
nb.fit(X_train, y_train)

# Predict on the test set
y_pred_nb = nb.predict(X_test)
y_prob_nb = nb.predict_proba(X_test)[:, 1]  # Probability estimates for ROC-AUC

# Evaluate the model
accuracy_nb = accuracy_score(y_test, y_pred_nb)
precision_nb = precision_score(y_test, y_pred_nb)
recall_nb = recall_score(y_test, y_pred_nb)
f1_nb = f1_score(y_test, y_pred_nb)
roc_auc_nb = roc_auc_score(y_test, y_prob_nb)
conf_matrix_nb = confusion_matrix(y_test, y_pred_nb)

print("\nNaive Bayes Classifier Performance:")
print(f"Accuracy: {accuracy_nb:.4f}")
print(f"Precision: {precision_nb:.4f}")
print(f"Recall: {recall_nb:.4f}")
print(f"F1-Score: {f1_nb:.4f}")
print(f"ROC-AUC: {roc_auc_nb:.4f}")
print(f"Confusion Matrix:\n{conf_matrix_nb}")
```

```
Naive Bayes Classifier Performance:
Accuracy: 0.9639
Precision: 0.9810
Recall: 0.9460
F1-Score: 0.9632
ROC-AUC: 0.9965
Confusion Matrix:
[[201522   3749]
 [ 11048 193526]]
```

# Results Analysis

## Results Analysis

### k-Nearest Neighbors (k-NN) Performance

```
plaintext
k-NN Classifier Performance:
Accuracy: 0.9961
Precision: 0.9964
Recall: 0.9957
F1-Score: 0.9961
ROC-AUC: 0.9961
Confusion Matrix:
[[204542    729]
 [   872 203702]]
```

### Naive Bayes Performance

```
plaintext
Naive Bayes Classifier Performance:
```

```
Accuracy: 0.9639
Precision: 0.9810
Recall: 0.9460
F1-Score: 0.9632
ROC-AUC: 0.9639
Confusion Matrix:
[[201522   3749]
 [ 11048 193526]]
```

## Interpretation of Performance Metrics

### k-Nearest Neighbors (k-NN):

- **Accuracy (0.9961):** Indicates that 99.61% of the instances were classified correctly. This is a very high accuracy, demonstrating that k-NN performs exceptionally well on this dataset.
- **Precision (0.9964):** Of the instances predicted as positive, 99.64% were truly positive. This shows a very low rate of false positives.
- **Recall (0.9957):** Of the true positive instances, 99.57% were correctly identified by the model. This indicates that k-NN has a very high true positive rate.
- **F1-Score (0.9961):** The harmonic mean of precision and recall, indicating an excellent balance between precision and recall.
- **ROC-AUC (0.9961):** A very high ROC-AUC value suggests that k-NN is excellent at distinguishing between the two classes.
- **Confusion Matrix:** The matrix shows that out of the total instances, 204,542 were true negatives, 729 were false positives, 872 were false negatives, and 203,702 were true positives.

### Naive Bayes:

- **Accuracy (0.9639):** Indicates that 96.39% of the instances were classified correctly. This is a high accuracy, though slightly lower than k-NN.
- **Precision (0.9810):** Of the instances predicted as positive, 98.10% were truly positive. This indicates a slightly higher rate of false positives compared to k-NN.
- **Recall (0.9460):** Of the true positive instances, 94.60% were correctly identified by the model. This is relatively high, but lower than k-NN.
- **F1-Score (0.9632):** The harmonic mean of precision and recall, indicating a good balance between precision and recall.
- **ROC-AUC (0.9639):** A high ROC-AUC value, though lower than k-NN, indicating good performance in distinguishing between the two classes.
- **Confusion Matrix:** The matrix shows that out of the total instances, 201,522 were true negatives, 3,749 were false positives, 11,048 were false negatives, and 193,526 were true positives.

## Insights and Observations

1. **k-NN Performance:**
    - **High Accuracy and ROC-AUC:** k-NN's high accuracy and ROC-AUC indicate that it is a robust model for this classification task, effectively distinguishing between classes.
    - **Balanced Precision and Recall:** The high precision and recall values suggest that k-NN maintains a low false positive rate while capturing most of the true positives.
    - **Confusion Matrix:** The relatively low number of false positives and false negatives highlights k-NN's effectiveness in classification.
2. **Naive Bayes Performance:**
    - **High Accuracy:** Naive Bayes has a high accuracy, although slightly lower than k-NN, indicating good performance on this dataset.
    - **High Precision but Lower Recall:** Naive Bayes has a high precision, meaning it has a lower rate of false positives, but its recall is slightly lower than k-NN, indicating it misses more true positives.

positives.

- **Confusion Matrix:** The higher number of false positives compared to k-NN suggests that Naive Bayes may be underpredicting the positive class. However, it still performs well overall.

# Conclusion and Future Work

## Conclusion

## Conclusion

Both k-NN and Naive Bayes classifiers perform well on this dataset, with k-NN having a slight edge in all performance metrics. This makes k-NN a better choice for this particular classification task, especially when high precision and recall are crucial. Naive Bayes, while slightly less accurate, still offers good performance and might be preferred in scenarios where computational efficiency is prioritized.

Based on the performance metrics and confusion matrices, k-NN outperforms Naive Bayes on this dataset. k-NN demonstrates higher accuracy, precision, recall, F1-score, and ROC-AUC, indicating its superiority in classifying the electrical load data. Naive Bayes, while simpler and faster, shows a higher rate of false positives and does not distinguish between classes as effectively as k-NN.

## Future Work

- **Hyperparameter Tuning:** For both k-NN and Naive Bayes, exploring hyperparameter tuning could potentially improve performance. For k-NN, varying the number of neighbors and distance metrics, and for Naive Bayes, experimenting with different distributions.
- **Feature Engineering:** Additional feature engineering and selection methods could enhance model performance.