Name: Sushant Tulasi
Class: D20B
Roll: 60

Aim: To Implement Inferencing with Bayesian Network in python

Naive Bayes Inference: Theory

What is Naive Bayes?

Naive Bayes is a probabilistic machine learning algorithm used for classification tasks. It's based on Bayes' Theorem with a key assumption: all features are **independent** of each other given the class. This "naive" independence assumption is why it's so efficient and easy to implement.

Bayes' Theorem

The foundation of Naive Bayes is **Bayes' Theorem**, which describes the probability of an event based on prior knowledge of conditions that might be related to the event. The formula is:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Where:

- $P(A|B)$ is the posterior probability: the probability of event A occurring given that event B has occurred.
- $P(B|A)$ is the likelihood: the probability of event B occurring given that event A has occurred.
- $P(A)$ is the prior probability: the initial probability of event A occurring.
- $P(B)$ is the marginal probability: the probability of event B occurring.

Naive Bayes for Classification

In the context of classification, we use Bayes' Theorem to find the probability of a class (C) given a set of features ($x_1, x_2, \ldots, x_n$):

$$P(C|x_1, x_2, \ldots, x_n) = \frac{P(x_1, x_2, \ldots, x_n|C) \cdot P(C)}{P(x_1, x_2, \ldots, x_n)}$$

Using the **"naive" independence assumption**, the likelihood term can be simplified:

$P(x_1, x_2, \ldots, x_n|C) = P(x_1|C) \cdot P(x_2|C) \cdot \ldots \cdot P(x_n|C)$

This simplifies the final formula for Naive Bayes to:

$$P(C|x_1, \ldots, x_n) \propto P(C) \cdot \prod_{i=1}^{n} P(x_i|C)$$

Since the denominator $P(x_1, \ldots, x_n)$ is constant for all classes, we can ignore it and focus on finding the class with the highest probability.

How Inference Works

Inference in Naive Bayes isn't just about outputting a single class label. It's about calculating the **probability distribution** over all possible classes for a given input. This is done by calculating the value of $P(C|x_1, \ldots, x_n)$ for each class and then normalizing them to sum to 1. This gives you the model's confidence for each possible outcome.

- **Example:** For a new data point, a Naive Bayes model might not just say "play tennis," but could provide the probabilities:
    - $P(\text{play tennis}|\text{weather, temp}) = 0.85$
    - $P(\text{don't play tennis}|\text{weather, temp}) = 0.15$

This probabilistic output is what makes it a powerful tool for inference, as it provides more insight than a simple classification label.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, classification_report, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Load the data from the CSV file
df = pd.read_csv('tennis_data.csv')

# Convert categorical data to numerical data using one-hot encoding
df = pd.get_dummies(df, columns=['weather', 'temperature'], drop_first=True)
```

```
# Separate features (X) and target (y)
X = df.drop('play_tennis', axis=1)
y = df['play_tennis']

# Convert target variable to numerical
y = y.apply(lambda x: 1 if x == 'yes' else 0)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)

# Initialize the Gaussian Naive Bayes classifier
model = GaussianNB()

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

# Make predictions on the test set
y_pred = model.predict(X_test)

# --- Generate and display performance metrics ---
# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Generate and print classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

# --- Plot the Confusion Matrix ---
fig, ax = plt.subplots(figsize=(8, 6))
ConfusionMatrixDisplay.from_estimator(model, X_test, y_test, ax=ax)
ax.set_title("Confusion Matrix")
plt.tight_layout()
plt.savefig('confusion_matrix.png')
print("Confusion matrix saved to confusion_matrix.png")
```

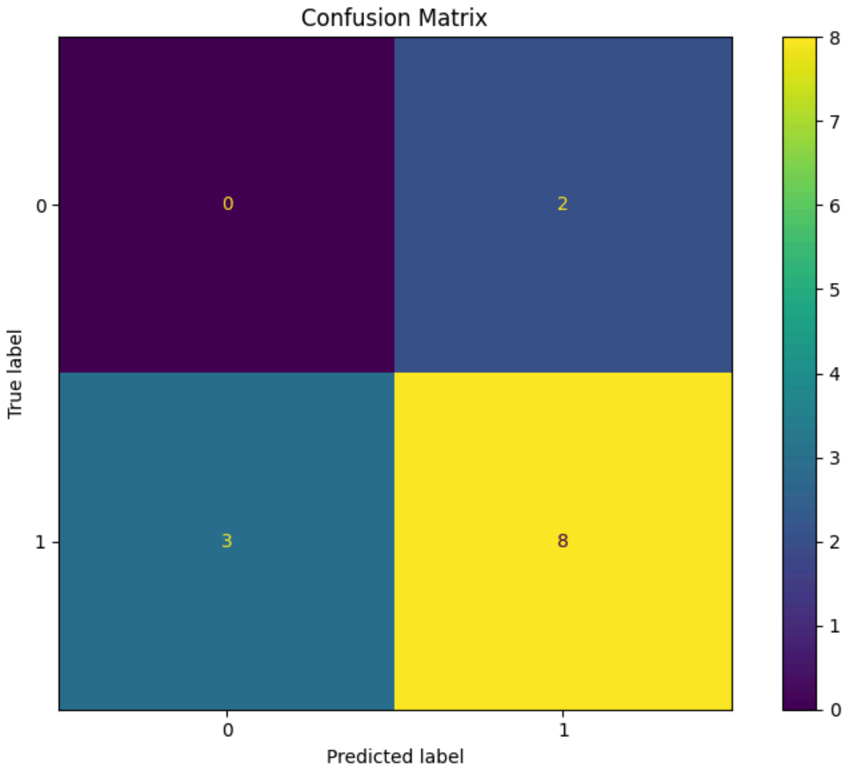
```
Accuracy: 0.62

Classification Report:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00         2
           1       0.80      0.73      0.76        11

    accuracy                           0.62        13
   macro avg       0.40      0.36      0.38        13
weighted avg       0.68      0.62      0.64        13

Confusion matrix saved to confusion_matrix.png
```



Bayesian Networks: The Theory

A **Bayesian network** is a probabilistic graphical model that represents a set of variables and their conditional dependencies using a **directed acyclic graph (DAG)**. It's a powerful tool for reasoning and inference under uncertainty.

Key Components

A Bayesian network consists of two main parts:

1. **Directed Acyclic Graph (DAG):** This graph represents the network's structure.
   - **Nodes:** Each node represents a random variable (e.g., `Burglary`, `Alarm`).
   - **Edges:** A directed edge from node A to node B signifies a **causal or influential relationship** where A is a parent of B. It means that the probability of B is conditionally dependent on the state of A. The graph must be acyclic, meaning there's no path that allows you to return to the same node.
2. **Conditional Probability Distributions (CPDs):** Each node has a conditional probability table (CPT) that quantifies the relationships defined by the graph.
   - For **root nodes** (nodes with no parents), the CPT is simply the **prior probability** of that variable (e.g., $P(\text{Burglary})$).
   - For **child nodes** (nodes with one or more parents), the CPT specifies the **conditional probability** of the node's state given all possible combinations of its parents' states (e.g., $P(\text{Alarm}|\text{Burglary}, \text{Earthquake})$).

The Joint Probability Distribution

The power of a Bayesian network lies in its ability to compactly represent the **full joint probability distribution** of all variables in the network. The joint probability of a specific state for all variables is calculated by taking the product of each node's conditional probability given its parents.

$$P(X_1, X_2, \dots, X_n) = \prod_{i=1}^{n} P(X_i|\text{Parents}(X_i))$$

This formula is a direct result of the conditional independence assumptions encoded in the network's structure. It allows for efficient calculation of probabilities, avoiding the need for a massive joint probability table.

Inference in Bayesian Networks

Inference is the process of calculating the posterior probability of a variable or a set of variables, given some evidence. This is the core task that Bayesian networks are designed for. You're essentially asking questions like:

- **Diagnostic Inference (Reasoning from effects to causes):** "What is the probability of a burglary given that the alarm is ringing?" ($P(\text{Burglary}|\text{Alarm})$)
- **Predictive Inference (Reasoning from causes to effects):** "What is the probability that John will call if there is a burglary?" ($P(\text{JohnCalls}|\text{Burglary})$)
- **Intercausal Inference (Reasoning about competing causes):** "Given that the alarm is ringing and there was no earthquake, what is the probability of a burglary?" ($P(\text{Burglary}|\text{Alarm}, \neg\text{Earthquake})$)

Algorithms like **Variable Elimination** or **Belief Propagation** are used to perform these complex probabilistic queries by efficiently manipulating the CPTs, allowing the model to "reason" about the state of the network.

```
# Install pgmpy library
!pip install pgmpy

# Import necessary modules
from pgmpy.models import DiscreteBayesianNetwork as BayesianNetwork
from pgmpy.factors.discrete import TabularCPD
from pgmpy.inference import VariableElimination


# Define the structure of the Bayesian Network
alarm_model = BayesianNetwork([('Burglary', 'Alarm'),
                               ('Earthquake', 'Alarm'),
                               ('Alarm', 'JohnCalls'),
                               ('Alarm', 'MaryCalls')])


# CPT for Burglary (B)
cpd_burglary = TabularCPD(variable='Burglary', variable_card=2,
                          values=[[0.999], [0.001]]) # P(~B)=0.999, P(B)=0.001

# CPT for Earthquake (E)
cpd_earthquake = TabularCPD(variable='Earthquake', variable_card=2,
                            values=[[0.998], [0.002]]) # P(~E)=0.998, P(E)=0.002
```

```
# CPT for Alarm (A) given Burglary and Earthquake
cpd_alarm = TabularCPD(variable='Alarm', variable_card=2,
                       values=[[0.999, 0.71, 0.06, 0.05], # P(~A|~B,~E), P(~A|B,~E), P(~A|~B,E), P(~A|B,E)
                               [0.001, 0.29, 0.94, 0.95]], # P(A|~B,~E), P(A|B,~E), P(A|~B,E), P(A|B,E)
                       evidence=['Burglary', 'Earthquake'],
                       evidence_card=[2, 2])

# CPT for JohnCalls (J) given Alarm
cpd_johncalls = TabularCPD(variable='JohnCalls', variable_card=2,
                           values=[[0.95, 0.10], # P(~J|~A), P(~J|A)
                                   [0.05, 0.90]], # P(J|~A), P(J|A)
                           evidence=['Alarm'],
                           evidence_card=[2])

# CPT for MaryCalls (M) given Alarm
cpd_marycalls = TabularCPD(variable='MaryCalls', variable_card=2,
                           values=[[0.10, 0.70], # P(~M|~A), P(~M|A)
                                   [0.90, 0.30]], # P(M|~A), P(M|A)
                           evidence=['Alarm'],
                           evidence_card=[2])

# Add the CPTs to the model
alarm_model.add_cpds(cpd_burglary, cpd_earthquake, cpd_alarm, cpd_johncalls, cpd_marycalls)

# Verify the model
print("Is the model valid? ", alarm_model.check_model())
```

```
Is the model valid?  True
```

```
# Create an inference object
alarm_infer = VariableElimination(alarm_model)
```

```
# P(Burglary | JohnCalls=True, MaryCalls=True)
query1 = alarm_infer.query(variables=['Burglary'], evidence={'JohnCalls': 1, 'MaryCalls': 1})
print("Probability of a burglary if both John and Mary call:")
print(query1)
```

```
Probability of a burglary if both John and Mary call:
+-------------+-----------------+
| Burglary    |   phi(Burglary) |
+=============+=================+
| Burglary(0) |          0.9944 |
+-------------+-----------------+
| Burglary(1) |          0.0056 |
+-------------+-----------------+
```

```
# P(Alarm | Burglary=False, Earthquake=True)
query2 = alarm_infer.query(variables=['Alarm'], evidence={'Burglary': 0, 'Earthquake': 1})
print("\nProbability of the alarm ringing if there's an earthquake but no burglary:")
print(query2)
```

```

Probability of the alarm ringing if there's an earthquake but no burglary:
+----------+--------------+
| Alarm    |   phi(Alarm) |
+==========+==============+
| Alarm(0) |       0.7100 |
+----------+--------------+
| Alarm(1) |       0.2900 |
+----------+--------------+
```

```
# P(Earthquake | Alarm=True, JohnCalls=False)
query3 = alarm_infer.query(variables=['Earthquake'], evidence={'Alarm': 1, 'JohnCalls': 0})
print("\nProbability of an earthquake if the alarm is ringing and John does not call:")
print(query3)
```

```

Probability of an earthquake if the alarm is ringing and John does not call:
+---------------+-------------------+
| Earthquake    |   phi(Earthquake) |
+===============+===================+
| Earthquake(0) |            0.7690 |
+---------------+-------------------+
| Earthquake(1) |            0.2310 |
+---------------+-------------------+
```

Conclusion

This series of exercises has provided a hands-on exploration of two fundamental probabilistic models: Naive Bayes and Bayesian Networks.