



ಕೆ.ವಿ.ಜಿ. ತಾಂತ್ರಿಕ ಮಹಾವಿದ್ಯಾಲಯ, ಸುಳ್ಳ ದ.ಕೆ. 574 327

K.V.G COLLEGE OF ENGINEERING, SULLIA, D.K. – 574 327
(AFFILIATED TO VISVESVARAYA TECHNOLOGICAL UNIVERSITY, BELAGAVI)



DEPARTMENT COMPUTER SCIENCE AND ENGINEERING(AI&ML)



www.kvgengg.com

PRACTICAL COMPONENT OF MACHINE LEARNING II

COURSE CODE: BAI702

VII SEMESTER



Course Material Prepared by

Course Material Approved by

LIST OF EXPERIMENTS

Sl. No.	Experiment	Page No.	Marks Awarded	Staff Signature
1	Read a dataset from the user and i. Use the Find-S algorithm to find the most specific hypothesis that is consistent with the positive examples. ii. What is the final hypothesis after processing all the positive examples? Using the same dataset, apply the Candidate Elimination algorithm. Determine the final version space after processing all examples (both positive and negative). What are the most specific and most general hypotheses in the version space?	4-6		
2	Read a dataset and use an example-based method (such as RIPPER or CN2) to generate a set of classification rules . Apply the FOIL algorithm (First-Order Inductive Learner) to learn first-order rules for predicting.	7-10		
3	Read a supervised dataset and use bagging and boosting technique to classify the dataset. Indicate the performance of the model.	11-12		
4	Read an unsupervised dataset and group the dataset based on similarity based on k-means clustering .	13-14		
5	Read a dataset and perform unsupervised learning using SOM algorithm.	15-16		
6	Write a function to generate uniform random numbers in the interval [0, 1]. Use this function to generate 10 random samples and evaluate $f(x)$ for each sample. What are the sampled function values? Using the samples generated in the previous step, estimate the integral I using the Monte Carlo method.	17-18		
7	Read a dataset and indicate the likelihood of an event occurring using Bayesian Networks.	19-20		
8	Refer to the dataset in question 7 and indicate inferences based on the sequence of steps .	21-24		
Average Marks Out of :				

Marks Distribution	Max. Marks	Marks Awarded
Average Marks Scaled Up		
Lab Test Marks		
Total Marks in the Practical Component of the Course		
Signature of the Staff with date		

Experiment 1: Read a dataset from the user and i. Use the Find-S algorithm to find the most specific hypothesis that is consistent with the positive examples. ii. What is the final hypothesis after processing all the positive examples? Using the same dataset, apply the Candidate Elimination algorithm. Determine the final version space after processing all examples (both positive and negative). What are the most specific and most general hypotheses in the version space?

Code Implementation:

```

import pandas as pd

# Load dataset

df = pd.read_csv("exp1_data.csv")

X = df.iloc[:, :-1].values # Features

y = df.iloc[:, -1].values # Target

# ----- FIND-S Algorithm ---- #

def find_s(X, y):

    hypothesis = X[0].copy()

    for i in range(len(y)):

        if y[i] == "Yes":

            for j in range(len(hypothesis)):

                if X[i][j] != hypothesis[j]:

                    hypothesis[j] = "?"

    return hypothesis

final_hypothesis = find_s(X, y)

print("== FIND-S RESULT ==")

print("Most Specific Hypothesis:", final_hypothesis)

# ----- CANDIDATE ELIMINATION ----- #

num_attributes = X.shape[1]

S = X[0].copy() # Most specific hypothesis

G = [['?' for _ in range(num_attributes)]] # Most general hypothesis

```

```
instance = X[i]
output = y[i]
if output == 'Yes':
    for j in range(num_attributes):
        if S[j] != instance[j]:
            S[j] = '?'
    G = [g for g in G if all(g[k] == '?' or g[k] == instance[k] for k in range(num_attributes))]
elif output == 'No':
    new_G = []
    for g in G:
        for j in range(num_attributes):
            if g[j] == '?':
                if S[j] != '?':
                    new_hypothesis = g.copy()
                    new_hypothesis[j] = S[j]
                    if new_hypothesis not in new_G:
                        new_G.append(new_hypothesis)
    G = new_G
print("\n==== CANDIDATE ELIMINATION RESULT ====")
print("Final Specific Hypothesis (S):", S)
print("Final General Hypotheses (G):")
for g in G:
    print(g)
```

Output:

==== FIND-S RESULT ====

Most Specific Hypothesis: ['Sunny' 'Warm' '?' 'Strong' '?' '?']

==== CANDIDATE ELIMINATION RESULT ====

Final Specific Hypothesis (S): ['Sunny' 'Warm' '?' 'Strong' '?' '?']

Final General Hypotheses (G):

['Sunny', '?', '?', '?', '?', '?']

['?', 'Warm', '?', '?', '?', '?']

['?', '?', '?', 'Strong', '?', '?']

Experiment 2: Read a dataset and use an example-based method (such as RIPPER or CN2) to generate a set of classification rules . Apply the FOIL algorithm (First-Order Inductive Learner) to learn first-order rules for predicting.

Code Implementation:

```
import pandas as pd

# Load dataset

df = pd.read_csv("exp2&3_data.csv") # CSV should have a "Fruit" column as target

# Split features and target

X = df.iloc[:, :-1].values # Features

y = df.iloc[:, -1].values # Target

attributes = list(df.columns[:-1])

target_class = "Apple" # You can change this as needed

def foil(X, y, attributes, target_class):

    rules = []

    used_indices = set()

    for i in range(len(X)):

        if y[i] != target_class or i in used_indices:

            continue

            rule = []

            for j in range(len(attributes)):

                attr_val = X[i][j]

                rule.append((j, attr_val))

    # Try all subsets of conditions to form a good rule

    for k in range(len(rule), 0, -1):

        from itertools import combinations

        forconds in combinations(rule, k):
```

```

# Check if this subset covers only positive examples

matches_target = []
matches_others = []
for m in range(len(X)):
    if all(X[m][j] == val for j, val inconds):
        if y[m] == target_class:
            matches_target.append(m)
        else:
            matches_others.append(m)
    if matches_target and not matches_others:
        # Valid rule found
        rule_str = " AND ".join(f'{attributes[j]} = {val}' for j, val inconds)
        rules.append(f'IF {rule_str} THEN {target_class}')
        used_indices.update(matches_target)
        break
    else:
        continue
return rules

def cn2_like(X, y, attributes):
    rules = []
    seen = set()
    for i in range(len(X)):
        rule = []
        for j in range(len(attributes)):
            rule.append(f'{attributes[j]} = {X[i][j]}')
        rule_str = f'IF {" AND ".join(rule)} THEN {y[i]}'
        if rule_str not in seen:
            seen.add(rule_str)
            rules.append(rule_str)
    return rules

```

```
seen.add(rule_str)

rules.append(rule_str)

return rules

foil_rules = foil(X, y, attributes, target_class)

cn2_rules = cn2_like(X, y, attributes)

print("\n==== CN2-like Example-based Rules ===")

for r in cn2_rules:

    print(r)

print("\n==== FOIL-style Rules to predict:", target_class, "====")

for r in foil_rules:

    print(r)
```

Output:

==== CN2-like Example-based Rules ===

IF Color = Red AND Shape = Small AND Size = Oblong THEN Apple
IF Color = Yellow AND Shape = Large AND Size = Oblong THEN Banana
IF Color = Green AND Shape = Medium AND Size = Round THEN Apple
IF Color = Red AND Shape = Medium AND Size = Oblong THEN Cherry
IF Color = Yellow AND Shape = Medium AND Size = Round THEN Banana
IF Color = Green AND Shape = Small AND Size = Oblong THEN Watermelon
IF Color = Red AND Shape = Large AND Size = Oblong THEN Cherry
IF Color = Yellow AND Shape = Large AND Size = Round THEN Banana
IF Color = Red AND Shape = Small AND Size = Round THEN Apple
IF Color = Red AND Shape = Large AND Size = Round THEN Apple
IF Color = Green AND Shape = Small AND Size = Round THEN Apple
IF Color = Green AND Shape = Large AND Size = Oblong THEN Watermelon
IF Color = Yellow AND Shape = Medium AND Size = Oblong THEN Banana
IF Color = Red AND Shape = Medium AND Size = Round THEN Cherry
IF Color = Yellow AND Shape = Small AND Size = Round THEN Banana
IF Color = Green AND Shape = Large AND Size = Round THEN Watermelon

IF Color = Green AND Shape = Medium AND Size = Oblong THEN Watermelon

==== FOIL-style Rules to predict: Apple ===

IF Color = Red AND Shape = Small AND Size = Oblong THEN Apple

IF Color = Green AND Shape = Medium AND Size = Round THEN Apple

IF Color = Red AND Shape = Small AND Size = Round THEN Apple

IF Color = Red AND Shape = Large AND Size = Round THEN Apple

IF Color = Green AND Shape = Small AND Size = Round THEN Apple

Experiment 3: Read a supervised dataset and use bagging and boosting technique to classify the dataset. Indicate the performance of the model. Code

Code Implementation:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.ensemble import BaggingClassifier, GradientBoostingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Step 1: Load dataset
data = pd.read_csv("exp2&3_data.csv") # Make sure your file is named 'fruit_data.csv'

# Step 2: Encode categorical features
le = LabelEncoder()
for column in data.columns:
    data[column] = le.fit_transform(data[column])

# Step 3: Split data
X = data.drop("Fruit", axis=1) # Features: Color, Shape, Size
y = data["Fruit"] # Target: Fruit type
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Step 4: Bagging
bagging = BaggingClassifier(DecisionTreeClassifier(), n_estimators=10, random_state=42)
bagging.fit(X_train, y_train)
bagging_preds = bagging.predict(X_test)
bagging_acc = accuracy_score(y_test, bagging_preds)

# Step 5: Boosting (Gradient Boosting)
boosting = GradientBoostingClassifier(n_estimators=10, max_depth=3, random_state=42)
boosting.fit(X_train, y_train)
boosting_preds = boosting.predict(X_test)
boosting_acc = accuracy_score(y_test, boosting_preds)
```

Step 6: Show results

```
print("Bagging Accuracy:", round(bagging_acc * 100, 2), "%")
```

```
print("Boosting Accuracy:", round(boosting_acc * 100, 2), "%")
```

Output:

Bagging Accuracy: 71.43 %

Boosting Accuracy: 71.43 %

Experiment 4: Read an unsupervised dataset and group the dataset based on similarity based on k-means clustering .

Code Implementation:

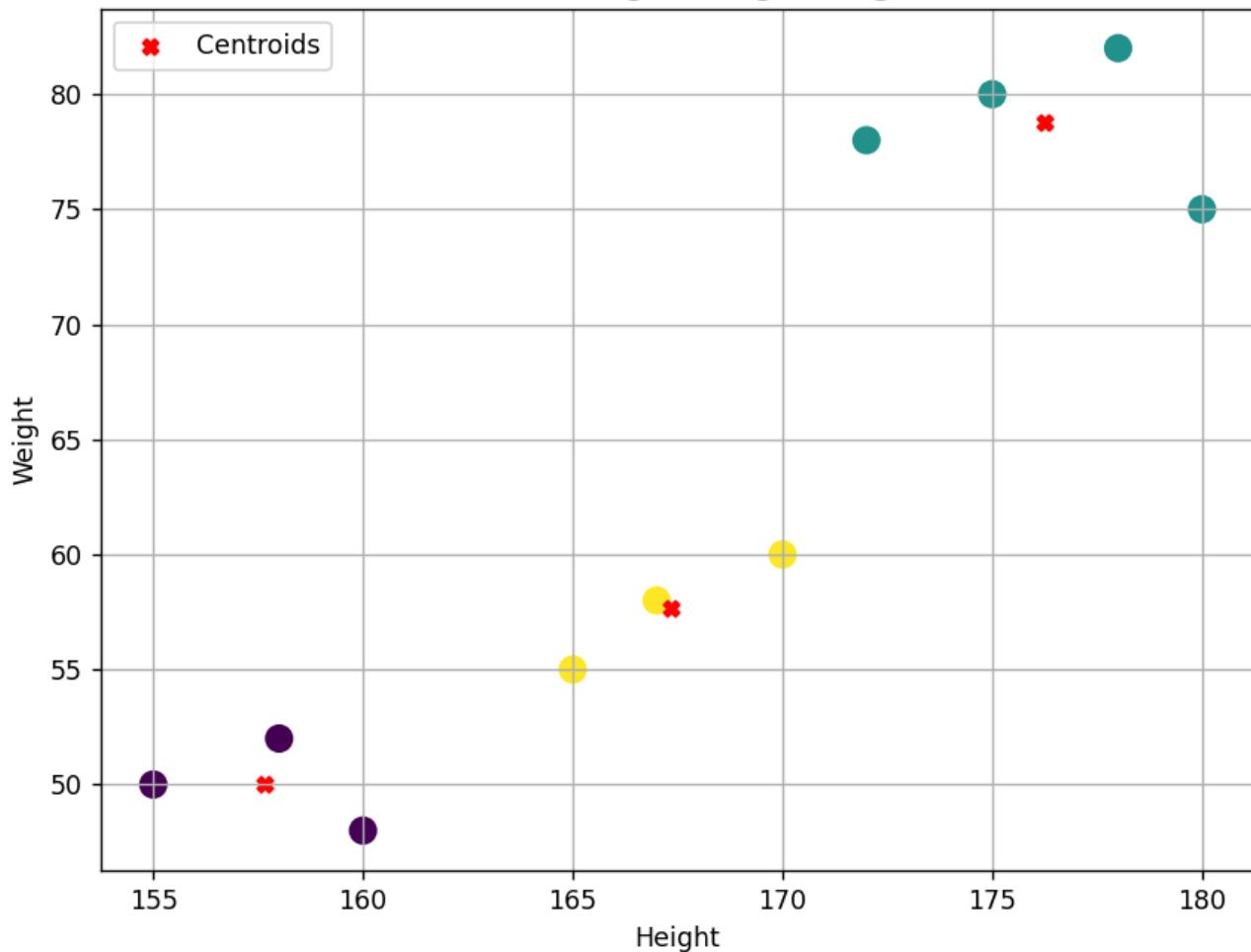
```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
# Step 1: Load the dataset
data = pd.read_csv('exp4&5_data.csv')
# Step 2: Apply K-Means Clustering
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans.fit(data)
# Step 3: Add cluster labels to the dataset
data['Cluster'] = kmeans.labels_
# Step 4: Display the clustered data
print(data)
# Step 5: Visualize the clusters
plt.figure(figsize=(8,6))
plt.scatter(data['Height'], data['Weight'], c=data['Cluster'], cmap='viridis', s=100)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], color='red',
marker='X', label='Centroids')
plt.xlabel("Height")
plt.ylabel("Weight")
plt.title("K-Means Clustering on Height-Weight Data")
plt.legend()
plt.grid(True)
plt.show()
```

Output:

Height Weight Cluster

0	170	60	2
1	165	55	2
2	180	75	1
3	155	50	0
4	160	48	0
5	175	80	1
6	172	78	1
7	178	82	1
8	158	52	0
9	167	58	2

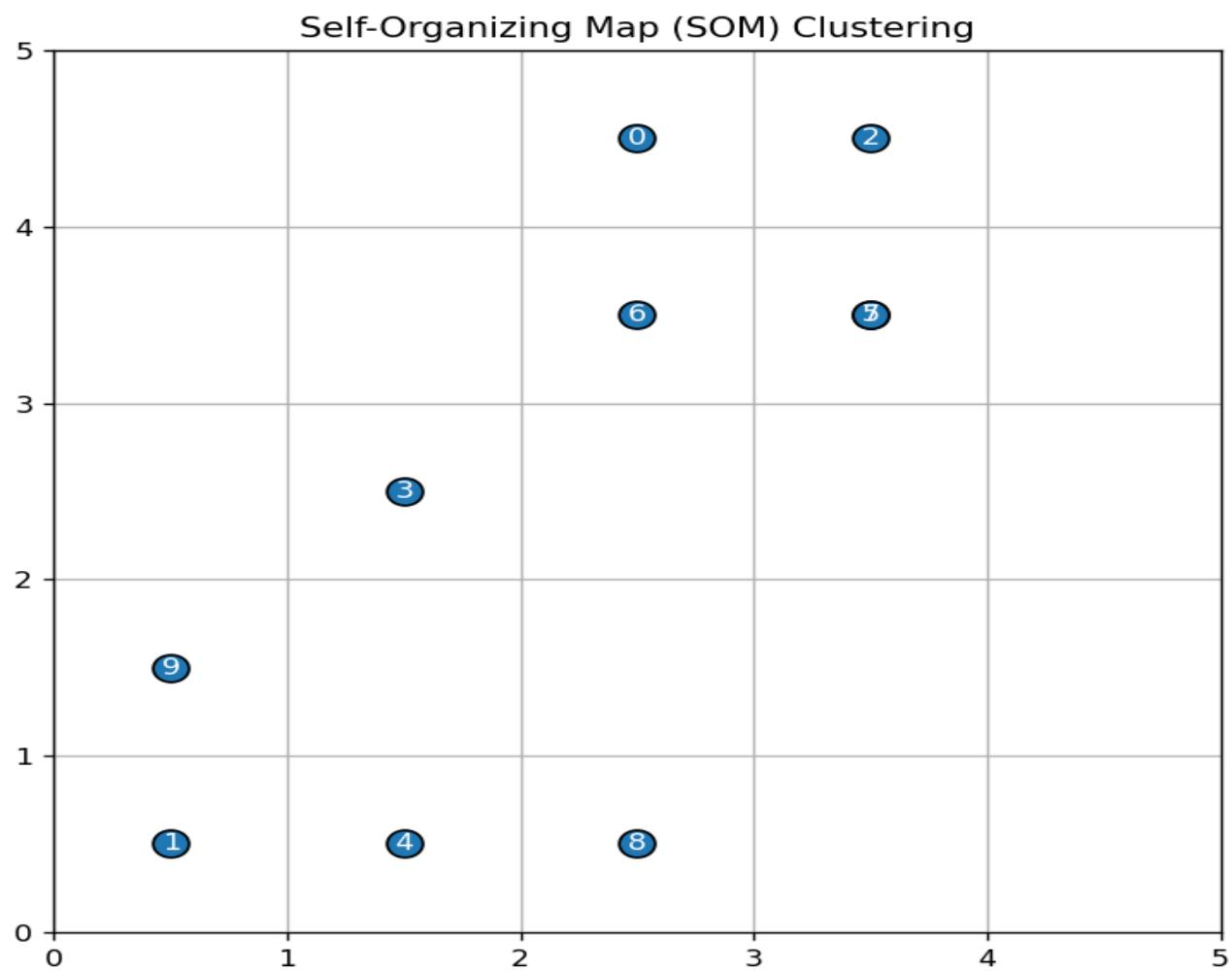
K-Means Clustering on Height-Weight Data



Experiment 5: Read a dataset and perform unsupervised learning using SOM algorithm.

Code Implementation:

```
import pandas as pd
import numpy as np
from minisom import MiniSom
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
# Load data
data = pd.read_csv('exp4&5_data.csv')
X = data.values
# Normalize data
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)
# Initialize and train SOM
som = MiniSom(x=5, y=5, input_len=2, sigma=1.0, learning_rate=0.5, random_seed=0)
som.train_random(X_scaled, 100)
# Plot the SOM with markers
plt.figure(figsize=(7, 7))
for i, x in enumerate(X_scaled):
    w = som.winner(x)
    plt.plot(w[0] + 0.5, w[1] + 0.5, 'o', markerfacecolor='C0',
             markeredgecolor='k', markersize=12)
    plt.text(w[0] + 0.5, w[1] + 0.5, str(i), color='white', ha='center', va='center')
plt.xlim(0, 5)
plt.ylim(0, 5)
plt.title('Self-Organizing Map (SOM) Clustering')
plt.grid(True)
plt.show()
```

Output:

Experiment 6: Write a function to generate uniform random numbers in the interval [0, 1]. Use this function to generate 10 random samples and evaluate f(x) for each sample. What are the sampled function values? Using the samples generated in the previous step, estimate the integral I using the Monte Carlo method.

Code Implementation:

```

import random

# 1. Generate a uniform random number in [0, 1]

def generate_uniform():
    return random.random()

# 2. Define the function f(x) = x^2 (you can change this)

def f(x):
    return x ** 2

# 3. Generate 10 random samples and evaluate f(x), More samples → Better approximation.

samples = [generate_uniform() for _ in range(10)]
function_values = [f(x) for x in samples]
print("Generated Samples:")

for i, x in enumerate(samples):
    print(f'x[{i+1}] = {x:.4f}, f(x) = {function_values[i]:.4f}')

# 4. Monte Carlo Integration over [0,1]

monte_carlo_estimate = sum(function_values) / len(samples)
print(f"\nEstimated Integral (Monte Carlo): {monte_carlo_estimate:.4f}")

```

Output:

Generated Samples:

```

x[1] = 0.1661, f(x) = 0.0276
x[2] = 0.6421, f(x) = 0.4123
x[3] = 0.7312, f(x) = 0.5347
x[4] = 0.7810, f(x) = 0.6100
x[5] = 0.2966, f(x) = 0.0880
x[6] = 0.7893, f(x) = 0.6229

```

$x[7] = 0.9423, f(x) = 0.8880$

$x[8] = 0.4592, f(x) = 0.2109$

$x[9] = 0.2868, f(x) = 0.0823$

$x[10] = 0.0209, f(x) = 0.0004$

Experiment 7: Read a dataset and indicate the likelihood of an event occurring using Bayesian Networks.**Code Implementation:**

```
import pandas as pd  
from pgmpy.models import DiscreteBayesianNetwork  
from pgmpy.estimators import MaximumLikelihoodEstimator  
from pgmpy.inference import VariableElimination  
  
# Step 1: Load the dataset  
data = pd.read_csv('exp7&8_data.csv')  
  
# Step 2: Define the Bayesian Network structure  
model = DiscreteBayesianNetwork([  
    ('Difficulty', 'Grade'),  
    ('Intelligence', 'Grade')  
])  
  
# Step 3: Train the model using Maximum Likelihood Estimation  
model.fit(data, estimator=MaximumLikelihoodEstimator)  
  
# Step 4: Inference  
inference = VariableElimination(model)  
  
# Step 5: Query probabilities  
print("P(Grade):")  
print(inference.query(variables=['Grade']))  
print("\nP(Grade | Intelligence=High):")  
print(inference.query(variables=['Grade'], evidence={'Intelligence': 'High'}))
```

Output:

INFO:pgmpy: Datatype (N=numerical, C=Categorical Unordered, O=Categorical Ordered) inferred from data:

```
{'Difficulty': 'C', 'Intelligence': 'C', 'Grade': 'C'}
```

P(Grade):

Grade	phi(Grade)
Grade(A)	0.2500
Grade(B)	0.5000
Grade(C)	0.2500

P(Grade | Intelligence=High):

Grade	phi(Grade)
Grade(A)	0.5000
Grade(B)	0.5000
Grade(C)	0.0000

Experiment 8: Refer to the dataset in question 7 and indicate inferences based on the sequence of steps .**Code Implementation:**

```
import pandas as pd
from pgmpy.models import DiscreteBayesianNetwork
from pgmpy.estimators import MaximumLikelihoodEstimator
from pgmpy.inference import VariableElimination
# Step 1: Load the dataset
data = pd.read_csv('exp7&8_data.csv')
# Step 2: Define the Bayesian Network structure
model = DiscreteBayesianNetwork([
    ('Difficulty', 'Grade'),
    ('Intelligence', 'Grade')
])
# Step 3: Train the model using MLE
model.fit(data, estimator=MaximumLikelihoodEstimator)
# Step 4: Perform inference
inference = VariableElimination(model)
# Step 5: Inference - Marginal probability of Grade
grade_dist = inference.query(variables=['Grade'])
print("\n📊 P(Grade):\n", grade_dist)
print("\n🧠 Inference 1:")
print("- Grade B is most common.")
print("- A and C are equally less frequent.")
# Step 6: Inference - P(Grade | Intelligence=High)
grade_given_intel = inference.query(variables=['Grade'], evidence={'Intelligence': 'High'})
print("\n📊 P(Grade | Intelligence=High):\n", grade_given_intel)
print("\n🧠 Inference 2:")
print("- High intelligence increases chances of Grade A or B.")
```

```

print("- Grade C becomes unlikely.")

# Step 7: Inference - P(Grade | Difficulty=Hard)

grade_given_difficulty = inference.query(variables=['Grade'], evidence={'Difficulty': 'Hard'})

print("\n📊 P(Grade | Difficulty=Hard):\n", grade_given_difficulty)

# Manually verify actual counts from dataset for Difficulty = Hard

print("\n📊 Actual frequencies from data (Difficulty=Hard):")

hard_df = data[data['Difficulty'] == 'Hard']

print(hard_df['Grade'].value_counts(normalize=True))

print("\n🧠 Inference 3:")

print("- From data: 60% C, 40% B for hard subjects.")

print("- If model shows 50-50, it might be due to inference smoothing or CPT grouping.")

print("- Grade A does not occur under Hard difficulty in data.")

```

Output:

INFO:pgmpy: Datatype (N=numerical, C=Categorical Unordered, O=Categorical Ordered) inferred from data:

{'Difficulty': 'C', 'Intelligence': 'C', 'Grade': 'C'}

📊 P(Grade):

Grade	phi(Grade)
Grade(A)	0.2500
Grade(B)	0.5000
Grade(C)	0.2500

🧠 Inference 1:

- Grade B is most common.
- A and C are equally less frequent.

 P(Grade | Intelligence=High):

Grade	phi(Grade)
Grade(A)	0.5000
Grade(B)	0.5000
Grade(C)	0.0000

 Inference 2:

- High intelligence increases chances of Grade A or B.
- Grade C becomes unlikely.

 P(Grade | Difficulty=Hard):

Grade	phi(Grade)
Grade(A)	0.0000
Grade(B)	0.5000
Grade(C)	0.5000

 Actual frequencies from data (Difficulty=Hard):

Grade

C 0.6

B 0.4

Name: proportion, dtype: float64

 Inference 3:

- From data: 60% C, 40% B for hard subjects.
- If model shows 50-50, it might be due to inference smoothing or CPT grouping.
- Grade A does not occur under Hard difficulty in data.