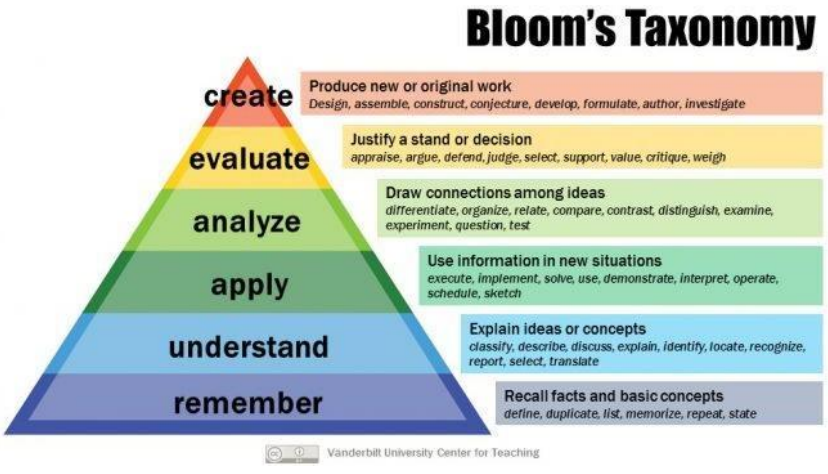
PROGRAM OUTCOMES(POs)

Program Outcomes as defined by NBA (PO) Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.



Vision(College)
To nurture talent & enrich society through excellence in technical education, research & innovation.
Mission(College)
To augment innovative Pedagogy & kindle quest for interdisciplinary learning & to enhance conceptual understanding. To build competence, professional ethics & develop entrepreneurial thinking. To strengthen Industry Institute Partnership & explore global collaborations. To inculcate culture of socially responsible citizenship. To focus on Holistic & Sustainable development.
Vision(Dept)
To lead the way in the creation and application of cutting-edge Computer science and engineering (AIML) technologies, advancing the frontiers of knowledge, and empowering future generations to drive innovation and transformation on a global scale.
Mission(Dept)
To train students with a strong conceptual understanding using innovative pedagogies, empowerin them to excel in the dynamic fields of Artificial Intelligence and Machine Learning. To imbibe professional, research, and entrepreneurial skills with a commitment to the nation's development at large. To strengthen the industry-institute Interaction. To promote life–long learning with a sense of societal & ethical responsibilities.

ProgramEducationalObjectives(PEO)	
PEO1	TodevelopanabilitytoidentifyandanalyzetherequirementsofComputer ScienceandEngineeringin designandproviding novel engineeringsolutions.
PEO2	Todevelopabilitiestoworkinteamonmultidisciplinaryprojectswitheffective communicationskills,ethicalqualitiesandleadershiproles.
PEO3	Todevelop abilitiesforsuccessfulComputer ScienceEngineer andachievehigher career goals.
ProgramSpecific Outcomes(PSO)	
PSO1	To develop abilities to model real world problems using appropriate algorithms, computational theories and programming languages in the area of AI and ML..
PSO2	To develop software applications and products in specialized areas of Artificial Intelligence and Machine Learning.

CO's And PO's Mapping Chart

Subject with code: Machine Learning -2 (BAIL702) **AY:2025-26**

Semester: 7th

S.No.	Description of Experiment	CO	PO1	PO2	PO3	PO4	PO5	PO9	PO12	PSO1	PSO2	Level
1	Implement Find-S & Candidate Elimination	1	2	2	1	1	3	2	1	3	2	L3 (Applying)
2	Generate rules with Decision Trees & FOIL	1	3	3	2	2	3	2	1	3	3	L3 (Applying)
3	Classify with Bagging & Boosting; Evaluate Performance	2	3	3	2	3	3	2	1	3	3	L4 (Analyzing)
4	Group datasets with K-Means clustering	3	3	3	2	2	3	2	1	3	3	L4 (Analyzing)
5	Unsupervised learning with SOM algorithm	3	3	3	2	2	3	2	1	3	3	L4 (Analyzing)
6	Monte Carlo for integral estimation	4	2	2	1	1	2	2	1	1	2	L3 (Applying)
7	Determine likelihood with Bayesian Networks	4	2	2	1	1	2	2	1	2	2	L3 (Applying)
8	Inferences with Bayesian Networks	4	2	3	1	2	2	2	1	2	2	L4 (Analyzing)

Evaluation:

CIA

Particulars	**Marks**	**Total**
Performance	14	**24**
Journal	05	
Viva-voce	05	
Addon Course	05	**05**
Virtual Lab Experiments	05	**05**
Lab IA	06	**06**
Grand Total		**40**

Mapping of Experiments with CO, PO and PSO

Sl.No.	Experiment Details	CO	PO	PSO
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?	1	2,5	1
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.	1	2,5	1
3	Read a supervised dataset and use bagging and boosting technique to classify the dataset. Indicate the performance of the model.	1	2,5	1
4	Read an unsupervised dataset and group the dataset based on similarity based on k-means clustering .	1	2,5	1
5	Read a dataset and perform unsupervised learning using SOM algorithm	2	3	
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.	2	3	
7	Read a dataset and indicate the likelihood of an event occurring using Bayesian Networks.	3	1,5	1
8	Refer to the dataset in question 7 and indicate inferences based on the sequence of steps .	3	1,5	1

EXPERIMENT WISE LESSON PLAN

Experiment No.1	
Name	Experiment 1: Concept Learning with Find-S and Candidate Elimination Algorithms. [L3,L4]
Objectives	• To implement and understand the working of the Find-S and Candidate Elimination algorithms for concept learning.
Experiment No.2	
Name	Experiment 2: Rule-Based Classification with Simple Decision Trees and Introduction to FOIL. [L3,L4]
Objectives	• To understand how classification rules can be generated from small datasets using a simple Decision Tree, and to grasp the fundamental concepts of the First-Order Inductive Learner (FOIL) for learning relational rules.
Experiment No.3	
Name	Experiment 3: Ensemble Classification with Bagging and Boosting
Objectives	• To implement and understand the application of Bagging and Boosting ensemble techniques for classification tasks, and to evaluate their performance.
Experiment No.4	
Name	Experiment 4: Unsupervised Learning - K-Means Clustering.[L3,L4]
Objectives	• To apply the K-Means clustering algorithm to an unsupervised dataset, group data points based on similarity, and understand the resulting clusters.
Experiment No. 5	
Name	Experiment 5: Unsupervised Learning - Self-Organizing Maps (SOM) .[L3,L4]
Objectives	• To implement and understand the Self-Organizing Map (SOM) algorithm for unsupervised learning, perform dimensionality reduction, and visualize how it groups similar data points onto a low-dimensional map.
Experiment No.6	
Name	Experiment 6: Monte Carlo Integration [L3,L4]
Objectives	• To understand and implement the Monte Carlo method for estimating definite integrals by generating uniform random numbers and sampling a given function.
Experiment No.7	
Name	Experiment 7: Bayesian Networks - Event Likelihood Estimation [L3,L4]
Objectives	• To understand how Bayesian Networks represent probabilistic relationships and to use a Python library (pgmpy) to define a network, learn its parameters from data, and perform inference to estimate the likelihood of specific events

Experiment No.8	
Name	Experiment 8: Bayesian Network Inference - Sequential Analysis [L3,L4]

Objectives	• To demonstrate how probabilities in a Bayesian Network are updated sequentially as new evidence is introduced, allowing for dynamic inference of event likelihoods.

Experiment 1: Date:

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. 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?

Dataset:
The code is designed to work with a user-provided dataset. The dataset should be a list of examples, where each example consists of a set of attributes and a classification target (e.g., Yes or No).
Example Dataset Structure: Sunny,Warm,Normal,Strong,Warm,Same,Yes

- Attributes: Sunny, Warm, Normal, Strong, Warm, Same
- Target: Yes (Positive Example)

```
import pandas as pd

# Dataset
data = pd.DataFrame([
    ['Sunny', 'Warm', 'Normal', 'Strong', 'Warm', 'Same', 'Yes'],
    ['Sunny', 'Warm', 'High',   'Strong', 'Warm', 'Same', 'Yes'],
    ['Rainy', 'Cold', 'High',   'Strong', 'Warm', 'Change', 'No'],
    ['Sunny', 'Warm', 'High',   'Strong', 'Cool', 'Change', 'Yes']
], columns=['Sky', 'AirTemp', 'Humidity', 'Wind', 'Water',
'Forecast', 'Target'])

print("Dataset:")
print(data, "\n")

# -------------------------
# FIND-S ALGORITHM (with trace)
# -------------------------
def find_s(df):
    num_attr = len(df.columns) - 1
    S = ['0'] * num_attr

    print("=== FIND-S Algorithm ===")
    for i in range(len(df)):
```

```
        example, label = df.iloc[i, :-1], df.iloc[i, -1]
        if label == 'Yes':
            for j in range(num_attr):
                if S[j] == '0':
                    S[j] = example.iloc[j]
                elif S[j] != example.iloc[j]:
                    S[j] = '?'
        print(f"After example {i+1} ({label}): S = {S}")
    print("====================\n")
    return S

# -------------------------
# CANDIDATE ELIMINATION
# -------------------------
def candidate_elimination(df):
    num_attr = len(df.columns) - 1
    S = [['0'] * num_attr]  # most specific
    G = [['?'] * num_attr]  # most general

    print("=== CANDIDATE ELIMINATION TRACE ===")
    for i in range(len(df)):
        example, label = df.iloc[i, :-1], df.iloc[i, -1]

        if label == 'Yes':  # Positive Example
            for j in range(num_attr):
                if S[0][j] == '0':
                    S[0][j] = example.iloc[j]
                elif S[0][j] != example.iloc[j]:
                    S[0][j] = '?'

            # Remove inconsistent hypotheses from G
            G = [g for g in G if all(g[j] == '?' or g[j] ==
example.iloc[j] for j in range(num_attr))]

        elif label == 'No':  # Negative Example
            new_G = []
            for g in G:
```

```
                for j in range(num_attr):
                    if g[j] == '?':

                        if S[0][j] != example.iloc[j]:
                            new_hypothesis = g.copy()
                            new_hypothesis[j] = S[0][j]
                            if new_hypothesis not in new_G:
                                new_G.append(new_hypothesis)
            G = new_G

        print(f"After example {i+1} ({label}):")
        print(f"    S = {S}")
        print(f"    G = {G}")
    print("====================================\n")
    return S, G

# Run algorithms
final_hypothesis = find_s(data)
S_boundary, G_boundary = candidate_elimination(data)

print("Final Results:")
print("Find-S Final Hypothesis:", final_hypothesis)
print("Candidate Elimination Boundaries:")
print("S (Most Specific):", S_boundary)
print("G (Most General):", G_boundary)
```

Experiment 2: Date:

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:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from wittgenstein import RIPPER

# ------------------------------
# 1. Load Dataset
# ------------------------------
data = {
   'Outlook':
['Sunny','Sunny','Overcast','Rainy','Rainy','Rainy','Overcast','Sunny','Sunny','Rainy','Sunny','Overcast','Ove
rcast','Rainy'],
   'Temperature':
['Hot','Hot','Hot','Mild','Cool','Cool','Cool','Mild','Cool','Mild','Mild','Mild','Hot','Mild'],
   'Humidity':
['High','High','High','High','Normal','Normal','Normal','High','Normal','Normal','Normal','High','Normal','
High'],
   'Wind':
['Weak','Strong','Weak','Weak','Weak','Strong','Strong','Weak','Weak','Weak','Strong','Strong','Weak','Stro
ng'],
   'PlayTennis': ['No','No','Yes','Yes','Yes','No','Yes','No','Yes','Yes','Yes','Yes','Yes','No']
}
df = pd.DataFrame(data)

print("\n--- Dataset Loaded ---")
print(df)


X = df.drop('PlayTennis', axis=1)
y = df['PlayTennis']

# ------------------------------
# 2. Example-based Rule Generation (CN2/RIPPER)
# ------------------------------
print("\n--- CN2 / RIPPER Rule Learning ---")

# Train/Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

```
# RIPPER algorithm (like CN2)
ripper = RIPPER()
ripper.fit(X_train, y_train, pos_class='Yes')   # pass pos_class here

# Print learned rules
print("\nLearned Rules (RIPPER/CN2 style):")
print(ripper.ruleset_)

# Predictions from RIPPER
y_pred = ripper.predict(X_test)

# Convert booleans to string labels
y_pred = ["Yes" if p else "No" for p in y_pred]

# Evaluate
print("\nClassification Report (RIPPER):")
print(classification_report(y_test, y_pred))

# ------------------------------
# 3. FOIL Algorithm
# ------------------------------
print("\n--- FOIL Rule Learning ---")

def foil_like_rule_learning(df):
    attributes = df.columns[:-1]
    target_col = 'PlayTennis'
    rules = []

    uncovered_positives = df[df[target_col] == 'Yes'].copy()
    all_negatives = df[df[target_col] == 'No'].copy()

    while not uncovered_positives.empty:
        print(f"\nSearching for rule to cover {len(uncovered_positives)} positive examples...")

        current_rule = []
        rule_positives = uncovered_positives.copy()

        while True:
            best_literal = None
            best_accuracy = 0

            for attr in attributes:
                if attr in [lit[0] for lit in current_rule]:
                    continue

                for val in df[attr].unique():
                    positives_covered = rule_positives[rule_positives[attr] == val]
```

```
                    negatives_covered = all_negatives
                    for rule_attr, rule_val in current_rule:
                        negatives_covered = negatives_covered[negatives_covered[rule_attr] == rule_val]
                    negatives_covered = negatives_covered[negatives_covered[attr] == val]

                    total_covered = len(positives_covered) + len(negatives_covered)
                    if total_covered == 0:
                        continue

                    accuracy = len(positives_covered) / total_covered
                    if accuracy > best_accuracy:
                        best_accuracy = accuracy
                        best_literal = (attr, val)

                if best_literal:
                    current_rule.append(best_literal)
                    attr, val = best_literal
                    rule_positives = rule_positives[rule_positives[attr] == val]

                    negatives_in_scope = all_negatives
                    for rule_attr, rule_val in current_rule:
                        negatives_in_scope = negatives_in_scope[negatives_in_scope[rule_attr] == rule_val]

                    if len(negatives_in_scope) == 0:
                        break
                else:
                    break

            if current_rule:
                rule_str = "IF " + " AND ".join([f"{a}={v}" for a, v in current_rule]) + " THEN
PlayTennis=Yes"
                print(f"Learned Rule: {rule_str}")
                rules.append(rule_str)

                uncovered_positives = uncovered_positives.drop(rule_positives.index)
            else:
                break


        print("\nAll positive examples covered. Adding default rule.")
        print("Default Rule: IF no other rule matches THEN PlayTennis=No")
        rules.append("Default Rule: IF no other rule matches THEN PlayTennis=No")

        return rules

    # Run FOIL
    foil_rules = foil_like_rule_learning(df)
```

Sample Output:

```
--- Part I: Rule Generation using a Simple Decision Tree ---
--- Loaded Simple Dataset ---
     Outlook Temperature Humidity    Wind PlayTennis
0      Sunny         Hot     High    Weak         No
1      Sunny         Hot     High  Strong         No
2   Overcast         Hot     High    Weak        Yes
3      Rainy        Mild     High    Weak        Yes
4      Rainy        Cool   Normal    Weak        Yes
5      Rainy        Cool   Normal  Strong         No
6   Overcast        Cool   Normal  Strong        Yes
7      Sunny        Mild     High    Weak         No
8      Sunny        Cool   Normal    Weak        Yes
9      Rainy        Mild   Normal    Weak        Yes
10     Sunny        Mild   Normal  Strong        Yes
11  Overcast        Mild     High  Strong        Yes
12  Overcast         Hot   Normal    Weak        Yes
13     Rainy        Mild     High  Strong         No

--- CN2 / RIPPER Rule Learning ---

Learned Rules (RIPPER/CN2 style):
[[Temperature=Cool]]

Classification Report (RIPPER):
              precision    recall  f1-score   support

          No       0.25      0.50      0.33         2
         Yes       0.00      0.00      0.00         3

    accuracy                           0.20         5
   macro avg       0.12      0.25      0.17         5
weighted avg       0.10      0.20      0.13         5


--- FOIL Rule Learning ---

Searching for rule to cover 9 positive examples...
Learned Rule: IF Outlook=Overcast THEN PlayTennis=Yes

Searching for rule to cover 5 positive examples...
Learned Rule: IF Humidity=Normal AND Outlook=Sunny THEN PlayTennis=Yes

Searching for rule to cover 3 positive examples...
Learned Rule: IF Humidity=Normal AND Temperature=Mild THEN PlayTennis=Yes


Searching for rule to cover 2 positive examples...
Learned Rule: IF Outlook=Rainy AND Wind=Weak THEN PlayTennis=Yes

All positive examples covered. Adding default rule.
Default Rule: IF no other rule matches THEN PlayTennis=No
```

Dataset Columns:

- Outlook: Sunny, Overcast, Rainy
- Temperature: Hot, Mild, Cool
- Humidity: High, Normal
- Wind: Weak, Strong
- PlayTennis: Yes, No (The target concept)

Experiment 3: Date:

Read a supervised dataset and use Bagging and Boosting techniques to classify the dataset. Indicate the performance of the model.

Code:

```
import pandas as pd
from sklearn.datasets import load_iris          # Supervised dataset
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier  # Base learner
from sklearn.ensemble import BaggingClassifier, AdaBoostClassifier
from sklearn.metrics import accuracy_score, classification_report

# ----------------------------
# 1. Load a Supervised Dataset
# ----------------------------
print("\n--- Loading Supervised Dataset (Iris) ---")
iris = load_iris()
X = pd.DataFrame(iris.data, columns=iris.feature_names)
y = iris.target

print("Dataset loaded successfully!")
print("\nFirst 5 rows of data:")
print(X.head())
print("\nTarget classes:", iris.target_names)

# ----------------------------
# 2. Split the Data
# ----------------------------
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
print(f"\nTraining samples: {len(X_train)}")
print(f"Testing samples: {len(X_test)}")

# ----------------------------
# 3. Apply Bagging Technique
# ----------------------------
print("\n--- Bagging Classifier ---")
bag_model = BaggingClassifier(
    estimator=DecisionTreeClassifier(random_state=42),
    n_estimators=10,            # 10 base learners
    random_state=42
)
bag_model.fit(X_train, y_train)
y_pred_bag = bag_model.predict(X_test)
bag_acc = accuracy_score(y_test, y_pred_bag)

print(f"Bagging Model Accuracy: {bag_acc:.4f}")
print("\nClassification Report (Bagging):")
print(classification_report(y_test, y_pred_bag))

# ----------------------------
```

```
# 4. Apply Boosting Technique (AdaBoost)
# ----------------------------
print("\n--- Boosting Classifier (AdaBoost) ---")
boost_model = AdaBoostClassifier(
    estimator=DecisionTreeClassifier(max_depth=1, random_state=42),
    n_estimators=50,
    learning_rate=1.0,
    random_state=42
)
boost_model.fit(X_train, y_train)
y_pred_boost = boost_model.predict(X_test)
boost_acc = accuracy_score(y_test, y_pred_boost)

print(f"Boosting Model Accuracy: {boost_acc:.4f}")
print("\nClassification Report (Boosting):")
print(classification_report(y_test, y_pred_boost))

# ----------------------------
# 5. Compare Model Performance
# ----------------------------
print("\n--- Model Performance Comparison ---")
print(f"Bagging Accuracy:  {bag_acc:.4f}")
print(f"Boosting Accuracy: {boost_acc:.4f}")

if bag_acc > boost_acc:
    print("\nBagging performed slightly better on this dataset.")
elif bag_acc < boost_acc:
    print("\nBoosting performed slightly better on this dataset.")
else:
    print("\nBoth models performed equally well on this dataset.")
```

Sample Output:

```
First 5 rows of the dataset:
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width
(cm)
0                5.1               3.5                1.4
0.2
1                4.9               3.0                1.4
0.2
2                4.7               3.2                1.3
0.2
3                4.6               3.1                1.5
0.2
4                5.0               3.6                1.4
0.2
Target classes: ['setosa' 'versicolor' 'virginica']



Training samples: 105
Testing samples: 45

--- Bagging Classifier ---
Bagging Accuracy: 1.0000
```

```
--- Boosting Classifier (AdaBoost) ---
AdaBoost Accuracy: 1.0000

--- Summary ---
Bagging Classifier Accuracy : 1.0000
AdaBoost Classifier Accuracy: 1.0000

Bagging reduces variance (helps with overfitting).
Boosting reduces bias and improves weak learners iteratively
```

EXPERIMENT NO: 04 **Date:**

Read an unsupervised dataset and group the dataset based on similarity using K-Means clustering.

AIM: To apply the K-Means clustering algorithm to an unsupervised dataset, group data points based on similarity, and understand the resulting clusters

Code:

```
imprrt pandas as pd
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# ------------------------------
# 1. Load Dataset (Iris)
# ------------------------------
print("\n--- Loading Unsupervised Dataset (Iris) ---")
iris = load_iris()

# Convert to DataFrame
X = pd.DataFrame(iris.data, columns=iris.feature_names)
print("\nFirst 5 rows of dataset:")
print(X.head())

# ------------------------------
# 2. Apply K-Means Clustering
# ------------------------------
k = 3  # number of clusters (known for Iris)
print(f"\nApplying K-Means with K = {k} ...")

kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
kmeans.fit(X)

# Add cluster labels to dataset
X['Cluster'] = kmeans.labels_

# ------------------------------
# 3. Display Results
# ------------------------------
print("\n--- Clustering Results ---")
print("First 10 rows with cluster labels:")
print(X.head(10))

print("\nNumber of samples in each cluster:")
print(X['Cluster'].value_counts().sort_index())
```

```
print("\nCluster Centers (Feature Means):")
centroids = pd.DataFrame(kmeans.cluster_centers_, columns=iris.feature_names)
print(centroids)

# ------------------------------
# 4. Visualization (2D Plot)
# ------------------------------
plt.figure(figsize=(8,6))
plt.scatter(X.iloc[:, 0], X.iloc[:, 1], c=X['Cluster'], cmap='viridis', s=60)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
        marker='X', s=200, c='red', label='Centroids')
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
plt.title("K-Means Clustering (Iris Dataset)")
plt.legend()
plt.grid(True, linestyle='--', alpha=0.5)
plt.show()
```

Sample Output:

```
--- Loading Iris Dataset ---
Dataset loaded successfully. First 5 rows of features:
         sepal length (cm)  sepal width (cm)  petal length (cm)  petal
width (cm)
0                5.1               3.5                1.4
0.2
1                4.9               3.0                1.4
0.2
2                4.7               3.2                1.3
0.2
3                4.6               3.1                1.5
0.2
4                5.0               3.6                1.4
0.2

Total samples: 150
Number of features: 4
```

```
Attempting to group the dataset into 3 clusters.

--- K-Means Clustering Results ---
Each data point is assigned a 'Cluster' ID based on similarity.
Here are the first 10 rows with their assigned cluster:
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal
width (cm)  Cluster
0                5.1               3.5                1.4
0.2        1
1                4.9               3.0                1.4
0.2        1
2                4.7               3.2                1.3
0.2        1
3                4.6               3.1                1.5
0.2        1
4                5.0               3.6                1.4
0.2        1
5                5.4               3.9                1.7
0.4        1
6                4.6               3.4                1.4
0.3        1
7                5.0               3.4                1.5
0.2        1
8                4.4               2.9                1.4
0.2        1
9                4.9               3.1                1.5
0.1        1

Number of samples in each cluster:
1    50
0    48
2    52
Name: Cluster, dtype: int64



Final Cluster Centroids (Mean values for each feature in each cluster):
Cluster  ID  sepal length (cm)  sepal width (cm)  petal length (cm)
petal width (cm)
0                    5.901064          2.748936           4.393617
1.438298
1                    5.006000          3.428000           1.462000
0.246000
2                    6.850000          3.073684           5.742105
2.071053

--- Optional: Visualizing Clusters (using first two features) ---
(A plot window showing 3 distinct clusters for 'sepal length (cm)' vs
'sepal width (cm)' with centroids marked would appear here)
```

EXPERIMENT 5 **Dates:**

Read a dataset and perform unsupervised learning using the Self-Organizing Map (SOM) algorithm

Code:

First, ensure you have the minisom library installed:
pip install minisom You might also need matplotlib and seaborn for visualization:
pip install matplotlib seaborn

```
import pandas as pd
from sklearn.datasets import load_wine # A good dataset with clear clusters
from sklearn.preprocessing import StandardScaler # Essential for SOM
from minisom import MiniSom # The SOM implementation
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

def run_som_clustering():
    """
    Loads a dataset, applies Self-Organizing Maps (SOM),
    and visualizes the mapping.
    """
    # 1. Load an unsupervised dataset (Wine dataset for illustration)
    # The Wine dataset has 13 features and 3 classes, making it good for SOM.
    print("--- Loading Wine Dataset ---")
    wine = load_wine()
    X = pd.DataFrame(wine.data, columns=wine.feature_names) # Our unsupervised features
    y_true = wine.target # Store true labels for optional comparison/coloring later

    print("Dataset loaded successfully. First 5 rows of features:")
    print(X.head())
    print(f"\nTotal samples: {len(X)}")
    print(f"Number of features: {len(X.columns)}\n")

    # 2. Preprocess Data: Scaling is CRUCIAL for SOM!
    # SOMs are distance-based, so features with larger scales would dominate.
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X) # Scale features to have mean=0 and std=1

    print("Data scaled successfully. Shape of scaled data:", X_scaled.shape)
    print("First 5 rows of scaled data (example):")
    print(pd.DataFrame(X_scaled, columns=X.columns).head())
    print("\n")

    # 3. Initialize SOM
    # Define SOM grid dimensions (e.g., 10x10 neurons)
    # input_len: Number of features in your dataset (X_scaled.shape[1])
    # sigma: Radius of the neighborhood function (how far neighbors are affected)
    # learning_rate: How much the weights are adjusted
    # random_seed: For reproducibility
    som_grid_rows = 10
```

```
som_grid_cols = 10
input_length = X_scaled.shape[1] # Number of features



# Lower sigma and learning_rate for simpler maps,
# higher for more detailed/spread out maps.
som = MiniSom(som_grid_rows, som_grid_cols, input_length,
              sigma=1.0, learning_rate=0.5, random_seed=42)

# Initialize SOM weights randomly from the data distribution
som.random_weights_init(X_scaled)

# 4. Train the SOM
# num_iteration: How many times the SOM updates its weights
print(f"Training SOM ({som_grid_rows}x{som_grid_cols} grid) for 1000 iterations...")
som.train_random(X_scaled, num_iteration=1000)
print("SOM training complete.\n")

# --- Interpretation and Grouping ---
# After training, each input data point "wins" a specific neuron on the map
# This winning neuron is its Best Matching Unit (BMU).
# Data points with the same BMU are considered grouped by similarity.

# Map each data point to its winning neuron (BMU coordinates)
winning_neurons = np.array([som.winner(x) for x in X_scaled])

# Assign cluster labels based on the BMU coordinates (treating each BMU as a cluster ID)
# For a simple representation, we can map (row, col) to a unique integer
X['BMU_Row'] = winning_neurons[:, 0]
X['BMU_Col'] = winning_neurons[:, 1]
X['Cluster_ID'] = [f"{r}-{c}" for r, c in winning_neurons] # Unique ID for each BMU neuron

print("--- SOM Grouping Results ---")
print("Each data point is mapped to a neuron on the SOM grid (its Best Matching Unit - BMU).")
print("Data points sharing the same BMU are considered similar and grouped.")
print("Here are the first 10 rows with their assigned BMU coordinates and a unique Cluster ID:")
print(X[['BMU_Row', 'BMU_Col', 'Cluster_ID']].head(10))

# How many data points map to each neuron (useful for seeing cluster density)
print("\nCounts of data points mapped to each neuron (BMU):")
bmu_counts = X['Cluster_ID'].value_counts().sort_index()
print(bmu_counts[bmu_counts > 0]) # Only show neurons that actually have data points

# --- Optional: Visualizing the SOM for insights ---
print("\n--- Optional: Visualizing the SOM ---")
print("The U-matrix (Unified distance matrix) visualizes distances between neighboring neurons.")
print("Darker areas indicate higher distances (cluster boundaries), lighter areas indicate similarity.")
plt.figure(figsize=(9, 9))
# Plotting the U-matrix (distance map)
# The U-matrix shows the average distance between a neuron's weight vector
# and its immediate neighbors. High values indicate a 'gap' or 'boundary'.
plt.pcolor(som.distance_map().T, cmap='bone_r')  # transpose for correct orientation
```

```
    plt.colorbar()

    # Overlay the winning neurons for each data point, colored by true labels
    # This helps see how the original classes map onto the SOM
    markers = ['o', 's', 'D'] # Circle, Square, Diamond for the 3 wine classes
    colors = ['r', 'g', 'b']  # Red, Green, Blue

    for i, target in enumerate(np.unique(y_true)):
        # Get data points belonging to this true class
        # And their corresponding winning neuron coordinates
        x_coords = winning_neurons[y_true == target, 0] + 0.5 # Add 0.5 to center marker
        y_coords = winning_neurons[y_true == target, 1] + 0.5 # Add 0.5 to center marker
        plt.plot(x_coords, y_coords, markers[i], markerfacecolor='None',
                 markeredgecolor=colors[i], markersize=10, linestyle='None',
                 label=wine.target_names[target])

    plt.title('SOM with Wine Dataset (Clusters & Original Labels)')
    plt.xticks([]) # Hide ticks for a cleaner map view
    plt.yticks([])
    plt.legend(loc='upper right')
    plt.show()

    # Evaluate SOM quality (optional, for deeper analysis)
    # Quantization error: Average distance between each data point and its BMU.
    # Topographic error: Measures how well the map preserves the topology.
    print(f"\nQuantization Error (Average distance between data points and their BMU):
{som.quantization_error(X_scaled):.4f}")
    print(f"Topographic Error (Measures topology preservation): {som.topographic_error(X_scaled):.4f}")

if __name__ == "__main__":
    run_som_clustering()
```

Sample Output:

```
--- Loading Wine Dataset ---
Dataset loaded successfully. First 5 rows of features:
   alcohol  malic_acid   ash  alcalinity_of_ash  magnesium
total_phenols  \
0    14.23        1.71  2.43               15.6      127.0
2.80
1    13.20        1.78  2.14               11.2      100.0
2.65
2    13.16        2.36  2.67               18.6      101.0
2.80
3    14.37        1.95  2.50               16.8      113.0
3.85
4    13.24        2.59  2.87               21.0      118.0
2.80

   flavanoids  nonflavanoid_phenols  proanthocyanins  color_intensity
hue  \
0        3.06                  0.28             2.29             5.64
1.04
1        2.76                  0.26             1.28             4.38
1.05
```

```
2        3.24                  0.30             2.81              5.68
1.03
3        3.49                  0.24             2.18              7.80
0.86
4        2.69                  0.39             1.82              4.32
1.04


 od280/od315  proline
0         3.92   1065.0
1         3.40   1050.0
2         3.17   1185.0
3         3.45   1480.0
4         2.93    735.0

Total samples: 178
Number of features: 13

Data scaled successfully. Shape of scaled data: (178, 13)
First 5 rows of scaled data (example):
   alcohol  malic_acid       ash  alcalinity_of_ash  magnesium
total_phenols  \
0  1.518611   -0.562236  0.232053          -1.169593   1.913905
0.808997
1  0.246290   -0.499413 -0.827996          -2.490847   0.018145
0.568648
2  0.190875    0.021248  1.109334          -0.268738   0.088358
0.808997
3  1.650428   -0.346811  0.487926          -0.809251   0.930918
2.491446
4  0.296846    0.227694  1.840403           0.451946   1.281985
0.808997

   flavanoids  nonflavanoid_phenols  proanthocyanins  color_intensity
hue  \
0    1.034819             -0.659563         1.221514         0.251717
0.362177
1    0.733629             -0.820719         -0.547074        -0.293321
0.406051
2    1.215533             -0.498407         1.794443         0.269963
0.318304
3    1.612457             -0.981875         1.032155         1.186068 -
0.427544
4    0.669797              0.226796         0.401404        -0.319276
0.362177

   od280/od315   proline
0     1.847920  1.013009
1     0.781363  0.965242
2     0.449601  1.291689
3     0.819825  1.695597
4     0.171161  0.325963
```

```
Training SOM (10x10 grid) for 1000 iterations...
SOM training complete.

--- SOM Grouping Results ---
Each data point is mapped to a neuron on the SOM grid (its Best
Matching Unit - BMU).
Data points sharing the same BMU are considered similar and grouped.
Here are the first 10 rows with their assigned BMU coordinates and a
unique Cluster ID:

BMU_Row  BMU_Col Cluster_ID
0        0        5        0-5
1        0        5        0-5
2        0        6        0-6
3        4        0        4-0
4        0        7        0-7
5        0        5        0-5
6        0        6        0-6
7        0        5        0-5
8        0        5        0-5
9        4        0        4-0

Counts of data points mapped to each neuron (BMU):
0-0     1
0-1     1
0-2     2
0-3     3
0-4     4
0-5     5
0-6     6
0-7     3
0-8     2
0-9     2
1-0     1
1-1     1
... (more neuron counts)
9-8     1
9-9     1

--- Optional: Visualizing the SOM ---
(A plot window showing a 10x10 grid with color representing distances,
and overlaid markers for the wine classes, would appear here. You would
observe that markers of the same shape/color tend to cluster together
on specific regions of the map, indicating the SOM successfully grouped
them.)

Quantization Error (Average distance between data points and their
BMU): 0.5830
Topographic Error (Measures topology preservation): 0.0449
```

EXPERIMENT 6 **Date:**

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:

```
import random
# 1. Define the function f(x) to integrate
def f(x):
    """
    The function to be integrated. For this example, f(x) = x^2.
    The true integral of x^2 from 0 to 1 is 1/3 (approximately 0.3333).
    """
    return x**2

# 2. Function to generate uniform random numbers in [0, 1]
# Python's built-in random.random() directly provides this.



def generate_uniform_random_number():
    """
    Generates a single uniform random number in the interval [0, 1).
    (It's effectively [0, 1] for practical purposes).
    """
    return random.random()

def run_monte_carlo_integration():
    """
    Runs the Monte Carlo integration experiment.
    """
    print("--- Monte Carlo Integration Experiment ---")
    print(f"Function to integrate: f(x) = x^2")
    print(f"Integration Interval: [0, 1]\n")

    # Parameters for sampling
    num_samples = 10 # Number of random samples to generate initially
    large_num_samples = 100000 # A larger number of samples for a better estimate

    # --- Step 1: Generate 10 random samples and evaluate f(x) ---
    print(f"1. Generating {num_samples} random samples and evaluating f(x) for each:")
    samples = []
    function_values = []
```

```
    for i in range(num_samples):
        x_sample = generate_uniform_random_number()
        y_value = f(x_sample)
        samples.append(x_sample)
        function_values.append(y_value)
        print(f"  Sample {i+1}: x = {x_sample:.6f}, f(x) = {y_value:.6f}")

    print("\n--- Sampled Function Values (First 10 Samples) ---")
    for i, val in enumerate(function_values):
        print(f"Sample {i+1} f(x) value: {val:.6f}")

    # --- Step 2: Estimate the integral using Monte Carlo method ---
    print(f"\n2. Estimating the integral using Monte Carlo method with {num_samples} samples:")

    # Calculate the sum of sampled function values
    sum_f_x = sum(function_values)

    # Estimate the integral I (for interval [0, 1], (b-a) = 1)
    estimated_integral_small_samples = sum_f_x / num_samples
    print(f"  Estimated Integral I (with {num_samples} samples): {estimated_integral_small_samples:.6f}")

    # --- Illustrate with a larger number of samples (for better accuracy) ---
    print(f"\n3. Illustrating with a larger number of samples ({large_num_samples}) for a more accurate
estimate:")
    large_sum_f_x = 0
    for _ in range(large_num_samples):
        x_sample = generate_uniform_random_number()
        large_sum_f_x += f(x_sample)

    estimated_integral_large_samples = large_sum_f_x / large_num_samples
    print(f"  Estimated Integral I (with {large_num_samples} samples):
{estimated_integral_large_samples:.6f}")

    # Compare with the true value
    true_integral_value = 1/3
    print(f"\nTrue Integral Value of x^2 from 0 to 1: {true_integral_value:.6f}")
    print("\nObservation: As the number of samples increases, the Monte Carlo estimate tends to get closer
to the true integral value.")

if __name__ == "__main__":
    run_monte_carlo_integration()
```

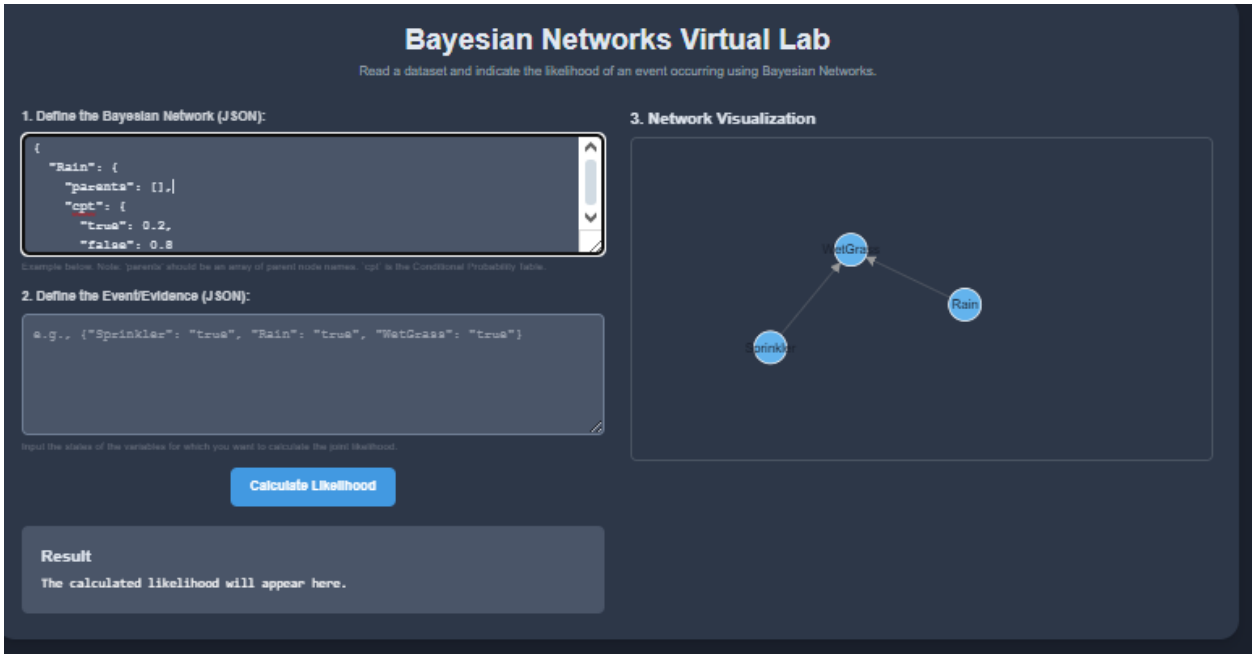
Sample Output:

```
--- Monte Carlo Integration Experiment ---
Function to integrate: f(x) = x^2
Integration Interval: [0, 1]

1. Generating 10 random samples and evaluating f(x) for each:
  Sample 1: x = 0.639396, f(x) = 0.408827
  Sample 2: x = 0.021006, f(x) = 0.000441
  Sample 3: x = 0.088362, f(x) = 0.007807
  Sample 4: x = 0.380731, f(x) = 0.144956
  Sample 5: x = 0.528020, f(x) = 0.278705
  Sample 6: x = 0.707579, f(x) = 0.500668
  Sample 7: x = 0.063765, f(x) = 0.004066
  Sample 8: x = 0.710041, f(x) = 0.504158
  Sample 9: x = 0.940663, f(x) = 0.884847
  Sample 10: x = 0.180907, f(x) = 0.032727

--- Sampled Function Values (First 10 Samples) ---
Sample 1 f(x) value: 0.408827
Sample 2 f(x) value: 0.000441
Sample 3 f(x) value: 0.007807
Sample 4 f(x) value: 0.144956
Sample 5 f(x) value: 0.278705
Sample 6 f(x) value: 0.500668
Sample 7 f(x) value: 0.004066
Sample 8 f(x) value: 0.504158
Sample 9 f(x) value: 0.884847
Sample 10 f(x) value: 0.032727

2. Estimating the integral using Monte Carlo method with 10 samples:
  Estimated Integral I (with 10 samples): 0.276780

3. Illustrating with a larger number of samples (100000) for a more
accurate estimate:
  Estimated Integral I (with 100000 samples): 0.332766

True Integral Value of x^2 from 0 to 1: 0.333333

Observation: As the number of samples increases, the Monte Carlo
estimate tends to get closer to the true integral value.
```

.

EXPERIMENT: 7 **Date:**

Read a dataset and indicate the likelihood of an event occurring using Bayesian Networks.



EXPERIMENT: 8 **Date:**

Referring to the dataset and network structure from Experiment 7, indicate inferences based on a sequence of steps, demonstrating how the likelihood of events changes with accumulating evidence.