AML Practical

Practical 1A: Expert systems and AI Bot Design an Expert System using AI/ML.
Example: An expert system for responding to patient queries for identifying flu.
Flu Disease prediction

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
data = {
'Fever':[1,1,1,1,1,0,0,0,1,0,1,1,0,0,1,1,0,1,0,1],
'Cough':[1,1,1,0,1,1,1,0,0,0,1,1,1,0,1,1,0,0,1,1],
'SoreThroat':[1,1,0,1,1,1,0,1,0,0,1,0,1,0,1,1,1,0,0,1],
'Fatigue':[1,1,1,1,0,1,0,0,0,0,1,1,0,1,1,0,1,1,0,1],
'Headache':[1,1,1,1,1,0,0,0,1,0,0,0,1,0,1,0,0,1,1,1],
'BodyPain':[1,0,1,1,1,0,0,0,0,0,1,1,0,0,1,1,0,1,0,1],
'RunnyNose':[1,1,1,1,1,1,1,0,0,1,1,1,0,1,0,0,0,0,1,1],
'Chills':[1,1,1,0,1,0,0,0,0,0,0,1,0,0,1,1,0,1,0,0],
'Flu':[1,1,1,1,1,0,0,0,0,0,1,1,0,0,1,1,0,1,0,1]
}
df = pd.DataFrame(data)
df.head()
X = df.drop('Flu', axis=1)
y = df['Flu']
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

model = RandomForestClassifier(n_estimators=100,
random_state=42)
model.fit(X_train, y_train)

pred = model.predict(X_test)
print("Model Accuracy:", accuracy_score(y_test, pred))
features = ['Fever', 'Cough', 'Sore Throat', 'Fatigue',
'Headache', 'Body Pain', 'Runny Nose', 'Chills']
```

```
user_data = []

print("****** FLU EXPERT SYSTEM ******")

for symptom in features:
    value = int(input(f"Do you have {symptom}? (1=Yes,
0=No): "))
    user_data.append(value)

result = model.predict([user_data])
print("\nPrediction:")
if result[0] == 1:
  print("Patient is likely suffering from FLU.")
else:
  print("Patient is NOT likely suffering from FLU.")
```

Practical 1B: AIML Chatbot Design

```
!pip -q install python-aiml
aiml_content = '''<?xml version="1.0" encoding="UTF-8"?>
<aiml version="1.0.1" encoding="UTF-8">

<category>
<pattern>Hello</pattern>
<template>Hello! How i can help you?</template>
</category>

<category>
<pattern>HI</pattern>
<template>HI! Nice to meet you.</template>
</category>

<category>
<pattern>HOW ARE YOU</pattern>
<template>I am doing well, Thank you for asking!</template>
```

```
</category>

<category>
<pattern>WHAT IS YOUR NAME?</pattern>
<template>My name is AIML Bot.</template>
</category>

<category>
<pattern>WHAT IS AIML</pattern>
<template>AIML stands for Artificial Intelligence Markup
Language. It is an XML-based language used to create
rule-based chatbots.</template>
</category>

<category>
<pattern>WHAT IS MACHINE LEARNING</pattern>
<template>Machine learning is a branch of AI in which
computers learn patterns from the data to make predictions
or decisions.</template>
</category>

<category>
<pattern>WHAT IS ARTIFICIAL INTELLIGENCE</pattern>
<template>Artificial Intelligence is the field of creating
systems that can perform tasks requiring human-like
intelligence.</template>
</category>

<category>
<pattern>THANK YOU</pattern>
<template>You are welcome!</template>
</category>

<category>
<pattern>BYE</pattern>
```

```
<template>GOODBYE! Have a great day.</template>
</category>

<category>
<pattern>EXIT</pattern>
<template>GOODBYE! See you again.</template>
</category>

<category>
<pattern>*</pattern>
<template>Sorry I dont understand.</template>
</category>

</aiml>
'''


with open('chatbot.aiml', 'w', encoding='utf-8') as f:
 f.write(aiml_content)

print('AIML knowledge base created successfully:
chatbot.aiml')
import aiml
import os

kernel = aiml.Kernel()
kernel.learn('chatbot.aiml')

print('AIML Chatbot loaded successfully.')
test_inputs = {'HELLO',
'WHAT IS YOUR NAME',
'WHAT IS AIML',
'WHAT IS MACHINE LEARNING',
'THANK YOU'}
```

```
for user_input in test_inputs:
 response=kernel.respond(user_input)
 print(f'You: {user_input}')
 print(f'Bot: {response}')
 print('_' * 50)
print ('AIML Chatbot Started')
print('Type BYE or EXIT to end the conversation.')
print()

while True:
 user_input = input('You: ').strip()
 if not user_input:
  continue

 response=kernel.respond(user_input.upper())
 print('Bot: ',  response)

 if user_input.upper() in ['BYE', 'EXIT']:
  break
```

Practical 2: intelligent_agent_AI_application

```
environment={
    'A':"Dirty",
    'B':"Dirty",

}
agent_location='A'
print('Initial environmrnt:',environment)
print('agent location:',agent_location)
class VacumCleanerAgent:
    def __init__(self,location="A"):
        self.location=location
        self.actions=0
        self.cleaned_rooms=0
```

```
    def perceive(self,environment):
        "Sense the condition of the current room."
        return environment[self.location]

    def decide_actions(self,percept):
      "Choose an action based on the current percept."
      if percept=="Dirty":
        return "Clean"
      elif self.location=="A":
        return "Move right"
      else:
        return "Move left"

    def act(self,action,environment):
      "Execute the action."
      self.actions +=1

      if action=="Clean":
        environment[self.location]="Cleaned"
        self.cleaned_rooms+=1
      elif action == 'move right':
        self.loaction = 'B'
      elif action == 'move left':
        self.location ='A'
agent = VacumCleanerAgent(location='A')

print('--- Intelligent Agent Execution ---')

max_steps = 18

for step in range(1, max_steps + 1):
  percept = agent.perceive(environment)
  action = agent.decide_actions(percept)
  agent.act(action, environment)
```

```
  print(f'Step {step}:')
  print(f'  Location: {agent.location}')
  print(f'  Percept: {percept}')
  print(f'  Action: {action}')
  print(f'  Rooms: {environment}')
  print()

  if all(status == 'Clean' for status in
environment.values()):
    print('Goal achieved: All rooms are clean.')
    break

clean_rooms = sum(1 for status in environment.values() if
status == 'Clean')
total_rooms = len(environment)
cleaning_percentage = (clean_rooms / total_rooms) * 100

print('--- Performance Evaluation ---')
print('Total Rooms:', total_rooms)
print('Clean Rooms:', clean_rooms)
print('Cleaning Rate:', f'{cleaning_percentage:.2f}%')
print('Actions Performed:', agent.actions)
print('Agent Location:', agent.location)


def get_room_status(room):
  while True:
    value = input(f'Enter status of Room {room}
(Clean/Dirty): ').strip().capitalize()
    if value in ['Clean', 'Dirty']:
      return value
    print('Please enter either Clean or Dirty.')

interactive_environment = {
```

```
  'A': get_room_status('A'),
  'B': get_room_status('B'),
}

interactive_agent = VacumCleanerAgent(location='A')

print('\nInitial Environment:', interactive_environment)

for step in range(1, 11):
  percept =
interactive_agent.perceive(interactive_environment)
  action = interactive_agent.decide_actions(percept)
  interactive_agent.act(action, interactive_environment)

  print(f'Step {step}: Percept={percept}, Action={action},
Environment={interactive_environment}')

  if all(status == 'Clean' for status in
interactive_environment.values()):
    print('Goal achieved: All rooms are clean.')
    break
```

Practical 3A: Implement Conditional Probability and Joint Probability using Python.

```
import pandas as pd

data = {
    'Student': ['A','B','C','D','E','F','G','H'],
    'Passed':  [1,1,0,1,0,1,0,1],
    'Studied': [1,1,1,0,0,1,0,0]
}

df = pd.DataFrame(data)
print(df)

total = len(df)

P_pass = len(df[df['Passed']==1])/total

P_study = len(df[df['Studied']==1])/total

P_pass_and_study = len(df[(df['Passed']==1) &
(df['Studied']==1)])/total

P_pass_given_study = P_pass_and_study/P_study

print("\nProbability of Passing:", round(P_pass,2))
print("Probability of Studying:", round(P_study,2))
print("Joint Probability P(Pass ∩ Study):",
round(P_pass_and_study,2))
print("Conditional Probability P(Pass | Study):",
round(P_pass_given_study,2))
```

Practical 3B: simple_ML_Train_Test_logistic_regression

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score,
classification_report, confusion_matrix
data = {
    'Age': [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
    'Study Hours': [2, 3, 4, 5, 6, 7, 8, 9, 10, 11],
    'Attendance': [60, 65, 70, 75, 80, 82, 85, 90, 92, 95],
    'Result': [0, 0, 0, 0, 1, 1, 1, 1, 1, 1]
}

df = pd.DataFrame(data)
print('Dataset:')
display(df)
X = df[['Age', 'Study Hours', 'Attendance']]
y = df['Result']

print('Features:')
display(X)

print('Target:')
display(y)
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.30, random_state=42, stratify=y)

print('Number of training instances:', len(X_train))
```

```
print('Number of testing instances:', len(X_test))

print('\nTraining Data:')
display(X_train.assign(Result=y_train))

print('\nTesting Data:')
display(X_test.assign(Result=y_test))
model = LogisticRegression(max_iter=1000)
model.fit(X_train, y_train)

print('Model training completed successfully.')
y_pred = model.predict(X_test)

results = X_test.copy()
results['Actual_Result'] = y_test.values
results['Predicted_Result'] = y_pred

print('Actual vs Predicted Results:')
display(results)
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy * 100:.2f}%')

print('\nClassification Report:')
print(classification_report(y_test, y_pred,
zero_division=0))

print('\nConfusion Matrix:')
print(confusion_matrix(y_test, y_pred))

new_instance = pd.DataFrame({
    'Age': [26],
    'Study Hours': [8],
    'Attendance': [88]
})
```

```
prediction = model.predict(new_instance)[0]
probability = model.predict_proba(new_instance)[0]

print('New Instance')
display(new_instance)

print('Predicted Result:', 'Pass' if prediction == 1 else
'Fail')
print('Prediction probabilities:', probability)
```

Practical 4: Find S

```
import pandas as pd
import numpy as np
#To read the data in the csv file
data = pd.read_csv("/data.csv")
print(data,"n")
#making an array of all the attributes
d = np .array(data)[:,:-1]
print("\nThe attributes are: ",d)

#segragrating the target that has positive and negative
examples
target = np.array (data)[:,-1]
print("\n The target is: ", target)

def train(c,t):
```

```
    # Initialize specific_hypothesis to None. This will be
updated with the first positive example.
    specific_hypothesis = None

    # Find the first positive example to initialize the
specific hypothesis
    for i, val in enumerate(t):
        # Assuming 'A' represents the positive examples
based on your target data
        if val == "A":
            specific_hypothesis = c[i].copy()
            break

    # If no positive examples ('A') are found in the target
data, the algorithm cannot proceed.
    if specific_hypothesis is None:
        print("Warning: No positive examples ('A') found in
target data. Cannot form a hypothesis.")
        return [] # Return an empty list or handle as
appropriate for your application

    # Iterate through all training examples to refine the
hypothesis
    for i, val in enumerate(c):
        # Only consider positive examples to generalize the
hypothesis
        if t[i] == "A":
            for x in range(len(specific_hypothesis)):
                # Generalize if the current attribute value
differs from the specific hypothesis
                # And the hypothesis attribute is not
already a wildcard '?'
                if specific_hypothesis[x] != '?' and val[x]
!= specific_hypothesis[x]:
                    specific_hypothesis[x] = '?'
```

```
    return specific_hypothesis

# Obtaining the final hypothesis
print("\n The final hypothesis is:",train(d,target))
```

Practical 5A: Dimensionality Reduction PCA

```
from sklearn.datasets import load_breast_cancer
breast = load_breast_cancer()
breast
```

```
breast_data = breast.data
breast_data
breast_data.shape
breast_labels = breast.target
breast_labels
breast_labels.shape
import numpy as np

labels = np.reshape(breast_labels,(569,1))
labels
final_breast_data =
np.concatenate([breast_data,labels],axis=1)

final_breast_data.shape
final_breast_data
import pandas as pd
breast_dataset = pd.DataFrame(final_breast_data)
breast_dataset
final_breast_data
features = breast.feature_names
features
features_labels = np.append(features,'label')
breast_dataset.columns = features_labels
breast_dataset.head()
breast_dataset['label'].replace(0, 'Benign',inplace=True)
breast_dataset['label'].replace(1,
'Malignant',inplace=True)
breast_dataset.tail()
from sklearn.preprocessing import StandardScaler
x = breast_dataset.loc[:, features].values
x = StandardScaler().fit_transform(x) # normalizing the
features
x.shape
np.mean(x),np.std(x)
feat_cols = ['feature'+str(i) for i in range(x.shape[1])]
```

```
normalised_breast = pd.DataFrame(x,columns=feat_cols)
normalised_breast.head()
normalised_breast.tail()
from sklearn.decomposition import PCA
pca_breast = PCA(n_components=2)
principalComponents_breast = pca_breast.fit_transform(x)
principal_breast_Df = pd.DataFrame(data =
principalComponents_breast, columns = ['principal component
1', 'principal component 2'])
principal_breast_Df.tail()
principal_breast_Df
print('Explained variation per principal component:
{}'.format(pca_breast.explained_variance_ratio_))
import matplotlib.pyplot as plt
plt.figure()
plt.figure(figsize=(10,10))
plt.xticks(fontsize=12)
plt.yticks(fontsize=14)

plt.xlabel('Principal Component - 1',fontsize=20)
plt.ylabel('Principal Component - 2',fontsize=20)
plt.title("Principal Component Analysis of Breast Cancer
Dataset",fontsize=20)
targets = ['Benign', 'Malignant']
colors = ['r', 'g']
for target, color in zip(targets,colors):
    indicesToKeep = breast_dataset['label'] == target
    plt.scatter(principal_breast_Df.loc[indicesToKeep,
'principal component 1']
               , principal_breast_Df.loc[indicesToKeep,
'principal component 2'], c = color, s = 50)

plt.legend(targets,prop={'size': 15})
```

Practical 5B: Candidate_Elimination

```
import pandas as pd
import numpy as np
dataset=pd.read_csv("/content/Dhoni Six records.csv")
dataset
print("The attributes are
{}".format(list(dataset.columns[:-1])))
print("The Label Class is {}".format(dataset.columns[-1]))
attribute_classes=[]
for i in range(len(dataset.columns[:-1])):

attribute_classes.append(list(dataset.iloc[:,i].unique()))

def candidate_elimination(data_samples):
  label=data_samples.iloc[:,-1].values
  data_samples=data_samples.iloc[:,:-1].values
  s=['Pi']*(data_samples.shape[1])
  g=[['?' for i in range(len(s))]for j in range(len(s))]
  for i,h in enumerate(data_samples):
    if label[i] == "Yes":
      if s[0]=='Pi':
        s=h.flatten().tolist()
        print("Initial Specific Hypothesis boundary :", s)
        print("Initial General Hypothesis boundary :", g)
      for x in range(len(s)):
        if h[x] != s[x]:
          s[x] = '?'
```

```
            g[x][x ]= '?'
      if label[i] == "No":
        for x in range(len(s)):
          if h[x] != s[x]:
            g[x][x] = s[x]
          else:
            g[x][x] = '?'
    indices = [i for i, val in enumerate(g) if val == ['?',
'?', '?', '?', '?', '?']]
    for i in indices:
      g.remove(['?', '?', '?', '?', '?', '?'])
    return s,g
s,g=candidate_elimination(dataset)
print("\n \nMost Generalized Specific Hypothesis :", s)
print("Most Specialized Generic Hypothesis :", g)
```

Practical 6A: Decision Tree and Random Forest Classification

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score,
confusion_matrix, ConfusionMatrixDisplay,
classification_report

print('Libraries imported successfully.')
#Load the Iris dataset
iris = load_iris()

X = iris.data
```

```
y = iris.target

feature_names = iris.feature_names
class_names = iris.target_names

print('Features:', feature_names)
print('Classes:', class_names)
print('Dataset shape:', X.shape)
#Convert dataset into a Dataframe for easy understanding
df = pd.DataFrame(X, columns=feature_names)
df['Target'] = y
df['Species'] =
df['Target'].map(dict(enumerate(class_names)))

display(df.head(10))
print('\nDataset information:')
print(df.info())
#Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.20, random_state=42, stratify=y)

print('Training samples:', X_train.shape[0])
print('Testing samples:', X_test.shape[0])
#Create and Train Decision Tree model
dt_model = DecisionTreeClassifier(criterion='entropy',
random_state=42)

dt_model.fit(X_train, y_train)

print('Decision Tree model trained successfully.')
#Prediction and test score for Decision Tree
dt_pred = dt_model.predict(X_test)
dt_test_score = dt_model.score(X_test, y_test)
dt_accuracy = accuracy_score(y_test, dt_pred)
```

```
print('Decision Tree Predictions:')
print(dt_pred)
print('\nDecision Tree test score:', round(dt_test_score,
4))
print('Decision Tree accuracy:', round(dt_accuracy, 4))

print('\nClassification Report:')
print(classification_report(y_test, dt_pred,
target_names=class_names))
#Confusion Matrix - Decision Tree
dt_cm = confusion_matrix(y_test, dt_pred)

print('Decision Tree Confusion Matrix:')
print(dt_cm)

disp = ConfusionMatrixDisplay(confusion_matrix=dt_cm,
display_labels=class_names)
disp.plot()
plt.title('Decision Tree - Confusion Matrix')
plt.show()
#Visualize the Decision Tree
plt.figure(figsize=(16, 9))
plot_tree(dt_model, feature_names=feature_names,
class_names=class_names, filled=True, rounded=True)
plt.title('Decision Tree Visualization')
plt.show()
#Create and train Random forest model
rf_model = RandomForestClassifier(n_estimators=100,
criterion='entropy', random_state=42)

rf_model.fit(X_train, y_train)

print('Random Forest model trained successfully.')
#Predition and test score for Random Forest
rf_pred = rf_model.predict(X_test)
```

```
rf_test_score = rf_model.score(X_test, y_test)
rf_accuracy = accuracy_score(y_test, rf_pred)

print('Random Forest Predictions:')
print(rf_pred)
print('\nRandom Forest test score:', round(rf_test_score,
4))
print('Random Forest accuracy:', round(rf_accuracy, 4))

print('\nClassification Report:')
print(classification_report(y_test, rf_pred,
target_names=class_names))
#Confusion Matrix - Random Forest
rf_cm = confusion_matrix(y_test, rf_pred)

print('Random Forest Confusion Matrix:')
print(rf_cm)

disp = ConfusionMatrixDisplay(confusion_matrix=rf_cm,
display_labels=class_names)
disp.plot()
plt.title('Random Forest - Confusion Matrix')
plt.show()
#Predict the class of a new flower
#[sepal length, sepal width, petal length, petal width]
new_data = np.array([[5.1, 3.5, 1.4, 0.2]])

dt_new = dt_model.predict(new_data)[0]
rf_new = rf_model.predict(new_data)[0]

print('New Data:', new_data[0])

print('Decision Tree Prediction:', class_names[dt_new])
print('Random Forest Prediction:', class_names[rf_new])
#Compare both models
```

```
comparison = pd.DataFrame({'Model':['Decision Tree',
'Random Forest'],
    'Test Score': [dt_test_score, rf_test_score],
    'Accuracy': [dt_accuracy, rf_accuracy]})

comparison['Test Score'] = comparison['Test
Score'].round(4)
comparison['Accuracy'] = comparison['Accuracy'].round(4)
display(comparison)
```

Practical 7: Least_square_regression

```
#Import
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from google.colab import files
#Upload cSV files
uploaded = files.upload()

filename = next(iter(uploaded))
df = pd.read_csv("/content/test.csv")

print("Dataset loaded successfully!")
display(df.head())
#Display dataset information
print("Shape: ", df.shape)
print("Columns: ", list(df.columns))
print("\nData types: ")
print(df.dtypes)
print("\nMissing values:")
print(df.isnull().sum())
#Change these column names according to your csv files
X_COLUMN = df.columns[0]
Y_COLUMN = df.columns[1]

X = df[X_COLUMN].astype(float).values
Y = df[Y_COLUMN].astype(float).values

print("Independent variable (X):", X_COLUMN)
print("Dependent variable (Y):", Y_COLUMN)
#Least Square Regression implementation from scratch

n = len(X)
```

```
sum_x = np.sum(X)
sum_y = np.sum(Y)
sum_xy = np.sum(X * Y)
sum_x2 = np.sum(X ** 2)

denominator = n * sum_x2 - (sum_x ** 2)

if denominator == 0:
    raise ValueError("Cannot calculate regression: all X
values are identical.")

b1 = (n * sum_xy - sum_x * sum_y) / denominator
b0 = (sum_y - b1 * sum_x) / n

print(f"Intercept (b0): {b0:.4f}")
print(f"Slope (b1): {b1:.4f}")
print(f"Regression equation: Y = {b0:.4f} + {b1:.4f}X")
#Make predictions
Y_pred = b0 + b1 * X

results = pd.DataFrame({
    "Actual": Y,
    "Predicted": Y_pred,
    "Residual": Y - Y_pred
})
display(results)
#Evaluate
#Calculate MSE and R-squared without using sklearn

mse = np.mean((Y - Y_pred) ** 2)

ss_res = np.sum((Y - Y_pred) ** 2)
ss_tot = np.sum((Y - np.mean(Y)) ** 2)
```

```
r2 = 1 - (ss_res / ss_tot) if ss_tot != 0 else 0

print(f"MSE: {mse:.4f}")
print(f"R-squared: {r2:.4f}")
#Plot actual data and Least square regression line

plt.figure(figsize=(8, 5))
plt.scatter(X, Y, label='Training Data')
plt.plot(X, Y_pred, label='Least Square Regression Line')
plt.xlabel(X_COLUMN)
plt.ylabel(Y_COLUMN)
plt.title('Least Square Regression Line')
plt.legend()
plt.grid(True)
plt.show()
#Predict for a new input values

new_x = float(input(f"Enter a new (X_COLUMN) value:"))
new_y = b0 + b1 * new_x

print(f"Predicted (Y_COLUMN) : {new_y:.4f}")
```

Practical 8: Bayesian Network for Heart Disease Diagnosis Aim To construct a Bayesian Network considering medical data and demonstrate the diagnosis of heart patients using the standard Heart Disease Dataset.

```
!pip install -q pgmpy
# Install pgmpy if it is not already installed
# Run this cell if required:
# %pip install pgmpy pandas numpy scikit-learn
import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score,
classification_report

from pgmpy.models import DiscreteBayesianNetwork
from pgmpy.estimators import BayesianEstimator
from pgmpy.inference import VariableElimination

print("Libraries imported successfully.")
# Load the dataset/
file_path = "/content/Heart.csv"
```

```
data = pd.read_csv("/content/Heart.csv")

print("Dataset shape:", data.shape)
display(data.head())

# Display dataset information
print("Column names:")
print(data.columns.tolist())

print("\nMissing values:")
print(data.isnull().sum())

print("\nTarget distribution:")
print(data["AHD"].value_counts())
# Remove the index column if present
if "Unnamed: 0" in data.columns:
    data = data.drop(columns=["Unnamed: 0"])

# Handle missing values
if "Ca" in data.columns:
    data["Ca"] = data["Ca"].fillna(data["Ca"].mode()[0])

if "Thal" in data.columns:
    data["Thal"] =
data["Thal"].fillna(data["Thal"].mode()[0])

print("Missing values after treatment:")
print(data.isnull().sum())
# Create categorical variables

data["AgeGroup"] = pd.cut(
    data["Age"],
    bins=[0, 40, 50, 60, 100],
    labels=["Young", "Middle", "Senior", "Old"]
).astype(str)
```

```
data["MaxHRGroup"] = pd.cut(
    data["MaxHR"],
    bins=[0, 100, 130, 160, 250],
    labels=["Low", "Moderate", "High", "VeryHigh"]
).astype(str)

# Convert categorical variables to strings
data["ChestPain"] = data["ChestPain"].astype(str)
data["ExAng"] = data["ExAng"].astype(str)
data["Ca"] = data["Ca"].astype(int).astype(str)
data["Thal"] = data["Thal"].astype(str)
data["AHD"] = data["AHD"].astype(str)

display(data[[
    "AgeGroup", "ChestPain", "MaxHRGroup",
    "ExAng", "Ca", "Thal", "AHD"
]].head())
# Select variables for the Bayesian Network
bn_data = data[
    [
        "AgeGroup",
        "ChestPain",
        "MaxHRGroup",
        "ExAng",
        "Ca",
        "Thal",
        "AHD"
    ]
].copy()

print("Bayesian Network data shape:", bn_data.shape)
display(bn_data.head())

# Split into training and testing datasets
```

```
train_data, test_data = train_test_split(
    bn_data,
    test_size=0.20,
    random_state=42,
    stratify=bn_data["AHD"]
)

print("Training records:", len(train_data))
print("Testing records:", len(test_data))
# Construct Bayesian Network
model = DiscreteBayesianNetwork([
    ("AgeGroup", "AHD"),
    ("ChestPain", "AHD"),
    ("MaxHRGroup", "AHD"),
    ("ExAng", "AHD"),
    ("Ca", "AHD"),
    ("Thal", "AHD")
])

print("Bayesian Network structure:")
print(list(model.edges()))
# Learn the Conditional Probability Tables (CPTs)

estimator = BayesianEstimator(model, train_data)

cpds = estimator.get_parameters(
    prior_type="BDeu",
    equivalent_sample_size=10
)

model.add_cpds(*cpds)

print("Model valid:", model.check_model())
# Display all learned CPTs
print("===== CONDITIONAL PROBABILITY TABLES =====")
```

```
for cpd in model.get_cpds():
    print("\n", cpd)

# Create inference engine
inference = VariableElimination(model)
print("Inference engine created successfully.")
# Evidence for a sample patient
patient = {
    "AgeGroup": "Senior",
    "ChestPain": "asymptomatic",
    "MaxHRGroup": "Moderate",
    "ExAng": "1",
    "Ca": "1",
    "Thal": "reversable"
}

result = inference.query(
    variables=["AHD"],
    evidence=patient
)

print("===== PATIENT DIAGNOSIS =====")
print(result)
# Extract posterior probabilities
state_names = result.state_names["AHD"]
probabilities = dict(zip(state_names, result.values))

probability_yes = probabilities.get("Yes", 0)
probability_no = probabilities.get("No", 0)

print(f"Probability of Heart Disease (Yes):
{probability_yes:.4f}")
print(f"Probability of No Heart Disease (No):
{probability_no:.4f}")
```

```
if probability_yes > probability_no:
    print("\nDiagnosis: HEART DISEASE PRESENT")
else:
    print("\nDiagnosis: NO HEART DISEASE")
# Evaluate Bayesian Network on test data

predictions = []
actual = []

feature_columns = [
    "AgeGroup",
    "ChestPain",
    "MaxHRGroup",
    "ExAng",
    "Ca",
    "Thal"
]

for _, row in test_data.iterrows():

    evidence = {column: row[column] for column in
feature_columns}

    result = inference.query(
        variables=["AHD"],
        evidence=evidence
    )

    predicted_class = result.state_names["AHD"][
        np.argmax(result.values)
    ]

    predictions.append(predicted_class)
    actual.append(row["AHD"])
```

```
accuracy = accuracy_score(actual, predictions)

print("===== MODEL PERFORMANCE =====")
print(f"Accuracy: {accuracy:.4f}")
print(f"Accuracy (%): {accuracy * 100:.2f}%")

print("\nClassification Report:")
print(classification_report(actual, predictions,
zero_division=0))

# Diagnose one actual patient from the test set

sample = test_data.iloc[0]

evidence = {column: sample[column] for column in
feature_columns}

result = inference.query(
    variables=["AHD"],
    evidence=evidence
)

predicted_class = result.state_names["AHD"][
    np.argmax(result.values)
]

print("Patient evidence:")
for key, value in evidence.items():
    print(f"{key}: {value}")

print("\nActual diagnosis:", sample["AHD"])
print("Predicted diagnosis:", predicted_class)

print("\nPosterior probabilities:")
```

```
print(result)
```

Practical 9: locally_weighted_regression_LWR

```
%pip install -q numpy pandas matplotlib scikit-learn
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
np.random.seed(42)
#Generate a reproducible non-linear dataset
X = np.linspace(0, 10, 100)
y = np.sin(X) + np.random.normal(0, 0.15, size=len(X))

data = pd.DataFrame({'X': X, 'Y': y})

display(data.head())
print("Number of observations:", len(data))
plt.figure(figsize=(9, 5))
plt.scatter(X, y, label='Observed data')
plt.xlabel('X')
plt.ylabel('Y')
```

```
plt.title('Original Non-Linear Dataset')
plt.grid(True, alpha=0.3)
plt.legend()
plt.show()
def locally_weighted_regression(x_train, y_train, x_query,
tau=0.5):
  x_train = np.asarray(x_train, dtype=float).reshape(-1)
  y_train = np.asarray(x_train, dtype=float).reshape(-1)

  #Design matrix for local linear regression
  X_design = np.column_stack((np.ones(len(x_train)),
x_train))

  #Gaussian weights
  weights = np.exp(-((x_train - x_query) ** 2) / (2 * tau
** 2))
  W = np.diag(weights)

  #Weighted least squares
  A = X_design.T @ W @ X_design
  B = X_design.T @ W @ y_train

  #Pseudo-inverse makes the implementation numerically
safer
  theta = np.linalg.pinv(A) @ B

  #Prediction at query point
  x_query_design = np.array([1.0, x_query])

  return float(x_query_design @ theta)

tau = 0.5

x_test = np.linspace(X.min(), X.max(), 400)
```

```
y_lwr = np.array([
    locally_weighted_regression(X, y, xq, tau = tau)
    for xq in x_test
])

plt.figure(figsize=(10, 6))
plt.scatter(X, y, label="Observed data")
plt.plot(x_test, y_lwr, linewidth=2, label=f"LWR Fit
(tau={tau})")
plt.xlabel("X")
plt.ylabel("Y")
plt.title("Locally Weighted Regression")
plt.grid(True, alpha=0.3)
plt.legend()
plt.show()
taus = [0.2, 0.5, 1.0, 2.0]

plt.figure(figsize=(11, 7))
plt.scatter(X, y, label="Observed data", alpha=0.6)

for tau_value in taus:
  y_fit = np.array([
      locally_weighted_regression(X, y, xq, tau =
tau_value)
      for xq in x_test
  ])
  plt.plot(x_test, y_fit, linewidth=2, label=f"tau =
{tau_value}")

plt.xlabel("X")
plt.ylabel("Y")
plt.title("Effect of Bandwidth (Tau) on LWR")
plt.grid(True, alpha=0.3)
plt.legend()
plt.show()
```

```
from sklearn.linear_model import LinearRegression

linear_model = LinearRegression()
linear_model.fit(X.reshape(-1, 1), y)

y_linear = linear_model.predict(x_test.reshape(-1, 1))

plt.figure(figsize=(10, 6))
plt.scatter(X, y, label="Observed data")
plt.plot(x_test, y_linear, linewidth=2, label="Ordinary
Linear Regression")
plt.plot(x_test, y_lwr, linewidth=2, label=f"LWR
(tau={tau})")
plt.xlabel("X")
plt.ylabel("Y")
plt.title("LWR vs Ordinary Linear Regression")
plt.grid(True, alpha=0.3)
plt.legend()
plt.show()
y_pred_training = np.array([
    locally_weighted_regression(X, y, xq, tau = tau)
    for xq in X
])

mse_lwr = mean_squared_error(y, y_pred_training)
r2_lwr = r2_score(y, y_pred_training)

y_linaer_training = linear_model.predict(X.reshape(-1, 1))
mse_linear = mean_squared_error(y, y_linaer_training)
r2_linear = r2_score(y, y_linaer_training)

results = pd.DataFrame({
    "Model": ["Locally Weighted Regression", "Ordinary
Linear Regression"],
    "MSE": [mse_lwr, mse_linear],
```

```
    "R2 Score": [r2_lwr, r2_linear]
})

display(results)
query_points = [2.0, 5.0, 8.0]

prediction_table = pd.DataFrame({
    "Query X": query_points,
    "LWR Prediction": [
        locally_weighted_regression(X, y, q, tau = tau)
        for q in query_points
    ]
})

display(prediction_table)
```

Practical 10A: Experiment_A_Pandas_Generative_AI

```
# Install required packages
# Install compatible setuptools and required packages
!pip install -q "setuptools<82" "jedi>=0.16"
!pip install -q pandasai pandas matplotlib
# Import libraries
import pandas as pd
import matplotlib.pyplot as plt
from pathlib import Path

print("Pandas version:", pd.__version__)
# Create a sample dataset
data = {
    "Student_ID": range(1, 16),
    "Name": [
        "Aarav","Diya","Rohan","Anaya","Vivaan",
        "Isha","Kabir","Meera","Arjun","Sara",
        "Aditya","Nisha","Rahul","Kiara","Dev"
    ],
    "Department": [
        "Computer Science","IT","Computer
Science","Management","IT",
        "Computer Science","Management","IT","Computer
Science","Management",
        "IT","Computer Science","Management","IT","Computer
Science"
    ],
```

```
    "Attendance":
[92,85,78,90,88,95,72,81,89,76,94,87,80,91,96],
    "Internal_Marks":
[45,39,36,44,41,47,34,38,43,35,46,40,37,42,48],
    "Assignment_Marks":
[18,17,15,19,16,20,14,16,18,15,19,17,16,18,20],
    "Final_Marks":
[88,76,69,91,82,94,65,73,86,68,92,79,71,84,96]
}

df = pd.DataFrame(data)
df

# Basic Pandas analysis
print("Shape:", df.shape)
print("\nColumns:", list(df.columns))
print("\nAverage Final Marks:",
round(df["Final_Marks"].mean(), 2))
print("\nHighest Final Marks:", df["Final_Marks"].max())
print("\nTop 5 students:")
display(df.nlargest(5, "Final_Marks")[["Name",
"Department", "Final_Marks"]])

# PandasAI setup
!pip install --upgrade pip wheel
# Downgrade setuptools to avoid conflict with torch
(requires setuptools<82)
!pip install setuptools==75.2.0 --force-reinstall
!pip -q install pandasai from pandasai import
SmartDataFrame

# PandasAI uses an LLM to translate natural-language
questions into analysis/code.
# For current PandasAI versions, an LLM provider/API key
may be required.
```

```
print("PandasAI imported successfully.")
# Natural-language-style analysis examples without an API
key

queries = {
    "Average final marks": df["Final_Marks"].mean(),
    "Highest final marks": df["Final_Marks"].max(),
    "Lowest final marks": df["Final_Marks"].min(),
    "Average attendance": df["Attendance"].mean()
}

for question, answer in queries.items():
    print(f"{question}: {answer:.2f}")

# Department-wise performance
dept_avg = (
    df.groupby("Department", as_index=False)["Final_Marks"]
      .mean()
      .sort_values("Final_Marks", ascending=False)
)

display(dept_avg)

dept_avg.plot(
    x="Department",
    y="Final_Marks",
    kind="bar",
    legend=False,
    figsize=(8, 5),
    title="Average Final Marks by Department"
)
plt.ylabel("Average Final Marks")
plt.xticks(rotation=20)
plt.tight_layout()
plt.show()
```

```
# Generate a simple analytical summary
best_student = df.loc[df["Final_Marks"].idxmax()]
best_department = dept_avg.iloc[0]

print("GENERATED DATA INSIGHTS")
print("-" * 50)
print(f"Highest-scoring student: {best_student['Name']}
({best_student['Final_Marks']} marks)")
print(f"Best-performing department:
{best_department['Department']} "
      f"({best_department['Final_Marks']:.2f} average
marks)")
print(f"Average class final marks:
{df['Final_Marks'].mean():.2f}")
print(f"Average attendance:
{df['Attendance'].mean():.2f}%")
```

Practical 10B: Implement a Generative Adversarial Network using Python and TensorFlow/Keras to generate synthetic handwritten digit images.

```
# If required in Google Colab, uncomment the next line:
# !pip install -q tensorflow matplotlib numpy
```

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf

from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Flatten,
Reshape, LeakyReLU, Input
from tensorflow.keras.optimizers import Adam

print("TensorFlow version:", tf.__version__)
(X_train, _), (_, _) = tf.keras.datasets.mnist.load_data()

print("Original dataset shape:", X_train.shape)

# Convert pixels from [0, 255] to [-1, 1]
X_train = X_train.astype("float32")
X_train = (X_train - 127.5) / 127.5

# Add channel dimension
X_train = np.expand_dims(X_train, axis=-1)

print("Preprocessed dataset shape:", X_train.shape)

plt.figure(figsize=(8, 8))

for i in range(16):
    plt.subplot(4, 4, i + 1)
    plt.imshow(X_train[i].reshape(28, 28), cmap="gray")
    plt.axis("off")

plt.suptitle("Real MNIST Images")
plt.show()

IMG_SIZE = 28
```

```
CHANNELS = 1
LATENT_DIM = 100
BATCH_SIZE = 128
EPOCHS = 30

print("Latent dimension:", LATENT_DIM)
print("Batch size:", BATCH_SIZE)
print("Epochs:", EPOCHS)

def build_generator():
    model = Sequential(name="Generator")
    model.add(Input(shape=(LATENT_DIM,)))
    model.add(Dense(128))
    model.add(LeakyReLU(negative_slope=0.2))
    model.add(Dense(256))
    model.add(LeakyReLU(negative_slope=0.2))
    model.add(Dense(28 * 28, activation="tanh"))
    model.add(Reshape((28, 28, 1)))
    return model

generator = build_generator()
generator.summary()

def build_discriminator():
    model = Sequential(name="Discriminator")
    model.add(Input(shape=(28, 28, 1)))
    model.add(Flatten())
    model.add(Dense(256))
    model.add(LeakyReLU(negative_slope=0.2))
    model.add(Dense(128))
    model.add(LeakyReLU(negative_slope=0.2))
    model.add(Dense(1, activation="sigmoid"))
    return model

discriminator = build_discriminator()
```

```
discriminator.compile(
    optimizer=Adam(learning_rate=0.0002, beta_1=0.5),
    loss="binary_crossentropy",
    metrics=["accuracy"]
)

discriminator.summary()

discriminator.trainable = False

gan = Sequential(name="GAN")
gan.add(generator)
gan.add(discriminator)

gan.compile(
    optimizer=Adam(learning_rate=0.0002, beta_1=0.5),
    loss="binary_crossentropy"
)

gan.summary()

def generate_images(epoch, rows=4, cols=4):
    noise = np.random.normal(0, 1, (rows * cols,
LATENT_DIM))
    generated = generator.predict(noise, verbose=0)
    generated = (generated + 1) / 2.0

    plt.figure(figsize=(8, 8))
    for i in range(rows * cols):
        plt.subplot(rows, cols, i + 1)
        plt.imshow(generated[i].reshape(28, 28),
cmap="gray")
        plt.axis("off")
```

```
    plt.suptitle(f"Generated Images — Epoch {epoch}")
    plt.tight_layout()
    plt.show()

def train_gan(epochs, batch_size):
    half_batch = batch_size // 2

    for epoch in range(1, epochs + 1):

        # Train Discriminator on real images
        idx = np.random.randint(0, X_train.shape[0],
half_batch)
        real_images = X_train[idx]
        real_labels = np.ones((half_batch, 1))

        # Generate fake images
        noise = np.random.normal(0, 1, (half_batch,
LATENT_DIM))
        fake_images = generator.predict(noise, verbose=0)
        fake_labels = np.zeros((half_batch, 1))

        d_loss_real =
discriminator.train_on_batch(real_images, real_labels)
        d_loss_fake =
discriminator.train_on_batch(fake_images, fake_labels)

        d_loss = 0.5 * (
            np.array(d_loss_real) + np.array(d_loss_fake)
        )

        # Train Generator through GAN
        noise = np.random.normal(0, 1, (batch_size,
LATENT_DIM))
        valid_labels = np.ones((batch_size, 1))
```

```
        g_loss = gan.train_on_batch(noise, valid_labels)

        if epoch == 1 or epoch % 5 == 0:
            print(
                f"Epoch {epoch:02d}/{epochs} | "
                f"D Loss: {d_loss[0]:.4f} | "
                f"D Accuracy: {100*d_loss[1]:.2f}% | "
                f"G Loss: {float(g_loss):.4f}"
            )
            generate_images(epoch)

train_gan(EPOCHS, BATCH_SIZE)

noise = np.random.normal(0, 1, (25, LATENT_DIM))

final_images = generator.predict(noise, verbose=0)
final_images = (final_images + 1) / 2.0

plt.figure(figsize=(10, 10))

for i in range(25):
    plt.subplot(5, 5, i + 1)
    plt.imshow(final_images[i].reshape(28, 28),
cmap="gray")
    plt.axis("off")

plt.suptitle("Final Synthetic Images Generated by GAN",
fontsize=16)
plt.tight_layout()
plt.show()

generator.save("mnist_gan_generator.keras")
print("Generator saved as mnist_gan_generator.keras")

noise = np.random.normal(0, 1, (1, LATENT_DIM))
```

```
new_image = generator.predict(noise, verbose=0)
new_image = (new_image + 1) / 2.0

plt.figure(figsize=(4, 4))
plt.imshow(new_image[0].reshape(28, 28), cmap="gray")
plt.title("New Synthetic Digit")
plt.axis("off")
plt.show()
```