ASSIGNMENT -6 (AI)

<Q.1>

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import KNeighborsRegressor

# Create the dataset

data = {

'Experience': [2, 3, 5, 7, 10, 12, 15],

'Written\_Score': [5, 6, 7, 8, 9, 9, 10],

'Interview\_Score': [6, 7, 8, 9, 9, 10, 10],

'Salary': [50000, 60000, 70000, 80000, 90000, 100000, 110000]

}

df = pd.DataFrame(data)

# Split data into features and target variable

X = df.drop('Salary', axis=1)

y = df['Salary']

# Split data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Standardize features

scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train)

X\_test\_scaled = scaler.transform(X\_test)

# Build KNN model

def train\_knn\_model(k):

knn\_model = KNeighborsRegressor(n\_neighbors=k)

knn\_model.fit(X\_train\_scaled, y\_train)

return knn\_model

# Predict function

def predict\_salary(knn\_model, experience, written\_score, interview\_score):

input\_data = scaler.transform([[experience, written\_score, interview\_score]])

predicted\_salary = knn\_model.predict(input\_data)

return predicted\_salary[0]

# Train KNN models with different values of k

k\_values = [3, 5, 7]

knn\_models = {}

for k in k\_values:

knn\_models[k] = train\_knn\_model(k)

# Predict salaries for given candidates

candidates = {

'Candidate A': (5, 8, 10),

'Candidate B': (8, 7, 6)

}

for candidate, data in candidates.items():

print(f"Predictions for {candidate}:")

for k, model in knn\_models.items():

predicted\_salary = predict\_salary(model, \*data)

print(f" Using k={k}: ${predicted\_salary:.2f}")

print()

<Q.2>

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

# Create the dataset

data = {

'Graduation\_Percentage': [85, 92, 78, 80, 95, 70, 88, 75, 82, 90,

85, 92, 78, 80, 95, 70, 88, 75, 82, 90,

85, 92, 78, 80, 95],

'Experience': [2, 3, 5, 7, 10, 12, 15, 4, 6, 8,

2, 3, 5, 7, 10, 12, 15, 4, 6, 8,

2, 3, 5, 7, 10],

'Written\_Score': [5, 6, 7, 8, 9, 9, 10, 7, 8, 8,

5, 6, 7, 8, 9, 9, 10, 7, 8, 8,

5, 6, 7, 8, 9],

'Interview\_Score': [6, 7, 8, 9, 9, 10, 10, 8, 9, 9,

6, 7, 8, 9, 9, 10, 10, 8, 9, 9,

6, 7, 8, 9, 9],

'Selection': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

1, 1, 1, 1, 1]

}

df = pd.DataFrame(data)

# Split data into features and target variable

X = df.drop('Selection', axis=1)

y = df['Selection']

# Split data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Build Bayesian model

bayes\_model = GaussianNB()

bayes\_model.fit(X\_train, y\_train)

# Predict selection status for unseen data

unseen\_data = {

'Graduation\_Percentage': [90, 75],

'Experience': [5, 8],

'Written\_Score': [8, 7],

'Interview\_Score': [10, 6]

}

unseen\_df = pd.DataFrame(unseen\_data)

predictions = bayes\_model.predict(unseen\_df)

# Calculate classification metrics

accuracy = accuracy\_score(y\_test, bayes\_model.predict(X\_test))

precision = precision\_score(y\_test, bayes\_model.predict(X\_test))

recall = recall\_score(y\_test, bayes\_model.predict(X\_test))

f1 = f1\_score(y\_test, bayes\_model.predict(X\_test))

# Save classification metrics in a CSV file

metrics\_data = {

'Accuracy': [accuracy],

'Precision': [precision],

'Recall': [recall],

'F1 Score': [f1]

}

metrics\_df = pd.DataFrame(metrics\_data)

metrics\_df.to\_csv('classification\_metrics.csv', index=False)

print("Predictions for unseen data:")

for i, prediction in enumerate(predictions):

print(f"Prediction for candidate {chr(97+i)}: {'Selected' if prediction == 1 else 'Not Selected'}")

<Q.3>

**import pandas as pd**

**from sklearn.datasets import load\_iris**

**from sklearn.model\_selection import train\_test\_split**

**from sklearn.tree import DecisionTreeClassifier**

**from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score**

**# Load Iris dataset**

**iris = load\_iris()**

**X = iris.data**

**y = iris.target**

**# Define function to train decision tree classifier with different percentages of training data**

**def train\_decision\_tree(train\_size):**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, train\_size=train\_size, random\_state=42)**

**clf = DecisionTreeClassifier(criterion='entropy')**

**clf.fit(X\_train, y\_train)**

**y\_pred = clf.predict(X\_test)**

**accuracy = accuracy\_score(y\_test, y\_pred)**

**precision = precision\_score(y\_test, y\_pred, average='weighted')**

**recall = recall\_score(y\_test, y\_pred, average='weighted')**

**f1 = f1\_score(y\_test, y\_pred, average='weighted')**

**return accuracy, precision, recall, f1**

**# Train decision tree classifier with different percentages of training data**

**train\_sizes = [0.5, 0.6, 0.7, 0.8, 0.9]**

**results = []**

**for train\_size in train\_sizes:**

**accuracy, precision, recall, f1 = train\_decision\_tree(train\_size)**

**results.append((train\_size, accuracy, precision, recall, f1))**

**# Display results**

**results\_df = pd.DataFrame(results, columns=['Train Size', 'Accuracy', 'Precision', 'Recall', 'F1 Score'])**

**print("Effect of different percentages of training data on quality parameters:")**

**print(results\_df)**

**# Explore effect of other decision tree parameters**

**# For example, let's vary max\_depth parameter**

**max\_depths = [None, 2, 3, 4, 5]**

**results\_max\_depth = []**

**for max\_depth in max\_depths:**

**clf = DecisionTreeClassifier(criterion='entropy', max\_depth=max\_depth)**

**clf.fit(X\_train, y\_train)**

**y\_pred = clf.predict(X\_test)**

**accuracy = accuracy\_score(y\_test, y\_pred)**

**precision = precision\_score(y\_test, y\_pred, average='weighted')**

**recall = recall\_score(y\_test, y\_pred, average='weighted')**

**f1 = f1\_score(y\_test, y\_pred, average='weighted')**

**results\_max\_depth.append((max\_depth, accuracy, precision, recall, f1))**

**# Display results for different max\_depth values**

**results\_max\_depth\_df = pd.DataFrame(results\_max\_depth, columns=['Max Depth', 'Accuracy', 'Precision', 'Recall', 'F1 Score'])**

**print("\nEffect of different max\_depth values on quality parameters:")**

**print(results\_max\_depth\_df)**

**<Q.4>**

**import pandas as pd**

**from sklearn.model\_selection import train\_test\_split**

**from sklearn.preprocessing import StandardScaler**

**from sklearn.neighbors import KNeighborsClassifier**

**from sklearn.naive\_bayes import GaussianNB**

**from sklearn.tree import DecisionTreeClassifier**

**from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score**

**# Load the Classified Data**

**data = pd.read\_csv("classified\_data.csv")**

**# Preprocess the data (if needed)**

**# Assuming no preprocessing needed as the data is already classified**

**# Split the data into features and target variable**

**X = data.drop('TARGET CLASS', axis=1)**

**y = data['TARGET CLASS']**

**# Split the data into training and testing sets**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)**

**# Standardize features for KNN**

**scaler = StandardScaler()**

**X\_train\_scaled = scaler.fit\_transform(X\_train)**

**X\_test\_scaled = scaler.transform(X\_test)**

**# Train and evaluate KNN model with different k values**

**k\_values = [3, 5, 7]**

**knn\_results = []**

**for k in k\_values:**

**knn\_model = KNeighborsClassifier(n\_neighbors=k)**

**knn\_model.fit(X\_train\_scaled, y\_train)**

**y\_pred = knn\_model.predict(X\_test\_scaled)**

**accuracy = accuracy\_score(y\_test, y\_pred)**

**precision = precision\_score(y\_test, y\_pred)**

**recall = recall\_score(y\_test, y\_pred)**

**f1 = f1\_score(y\_test, y\_pred)**

**knn\_results.append(('KNN', k, accuracy, precision, recall, f1))**

**# Train and evaluate Bayesian Classifier**

**bayes\_model = GaussianNB()**

**bayes\_model.fit(X\_train, y\_train)**

**y\_pred\_bayes = bayes\_model.predict(X\_test)**

**accuracy\_bayes = accuracy\_score(y\_test, y\_pred\_bayes)**

**precision\_bayes = precision\_score(y\_test, y\_pred\_bayes)**

**recall\_bayes = recall\_score(y\_test, y\_pred\_bayes)**

**f1\_bayes = f1\_score(y\_test, y\_pred\_bayes)**

**bayes\_results = [('Bayesian Classifier', '-', accuracy\_bayes, precision\_bayes, recall\_bayes, f1\_bayes)]**

**# Train and evaluate Decision Tree model with different max\_depth values**

**max\_depths = [None, 2, 3, 4, 5]**

**dt\_results = []**

**for max\_depth in max\_depths:**

**dt\_model = DecisionTreeClassifier(max\_depth=max\_depth)**

**dt\_model.fit(X\_train, y\_train)**

**y\_pred\_dt = dt\_model.predict(X\_test)**

**accuracy\_dt = accuracy\_score(y\_test, y\_pred\_dt)**

**precision\_dt = precision\_score(y\_test, y\_pred\_dt)**

**recall\_dt = recall\_score(y\_test, y\_pred\_dt)**

**f1\_dt = f1\_score(y\_test, y\_pred\_dt)**

**dt\_results.append(('Decision Tree', max\_depth, accuracy\_dt, precision\_dt, recall\_dt, f1\_dt))**

**# Combine results**

**results = knn\_results + bayes\_results + dt\_results**

**# Save results in a CSV file**

**results\_df = pd.DataFrame(results, columns=['Model', 'Parameter', 'Accuracy', 'Precision', 'Recall', 'F1 Score'])**

**results\_df.to\_csv('model\_comparison\_results.csv', index=False)**

**print("Results saved in 'model\_comparison\_results.csv' file.")**

**ASSIGNMENT UCS411 AI**

**<Q.1>**

**from collections import deque**

**def bfs(graph, start):**

**visited = set()**

**queue = deque([start])**

**visited.add(start)**

**while queue:**

**vertex = queue.popleft()**

**print(vertex, end=' ')**

**for neighbor in graph[vertex]:**

**if neighbor not in visited:**

**visited.add(neighbor)**

**queue.append(neighbor)**

**# Example usage**

**graph = {**

**'A': ['B', 'C'],**

**'B': ['A', 'D', 'E'],**

**'C': ['A', 'F'],**

**'D': ['B'],**

**'E': ['B', 'F'],**

**'F': ['C', 'E']**

**}**

**print("BFS traversal:")**

**bfs(graph, 'A')**

**<Q.2>**

**def dfs(graph, start, visited=None):**

**if visited is None:**

**visited = set()**

**visited.add(start)**

**print(start, end=' ')**

**for neighbor in graph[start]:**

**if neighbor not in visited:**

**dfs(graph, neighbor, visited)**

**# Example usage**

**print("\nDFS traversal:")**

**dfs(graph, 'A')**

**<Q.3>**

**import heapq**

**def ucs(graph, start, goal):**

**visited = set()**

**heap = [(0, start)]**

**while heap:**

**cost, vertex = heapq.heappop(heap)**

**if vertex not in visited:**

**print(vertex, end=' ')**

**visited.add(vertex)**

**if vertex == goal:**

**return**

**for neighbor, weight in graph[vertex].items():**

**if neighbor not in visited:**

**heapq.heappush(heap, (cost + weight, neighbor))**

**# Example usage**

**graph = {**

**'A': {'B': 1, 'C': 3},**

**'B': {'A': 1, 'D': 5, 'E': 7},**

**'C': {'A': 3, 'F': 2},**

**'D': {'B': 5},**

**'E': {'B': 7, 'F': 1},**

**'F': {'C': 2, 'E': 1}**

**}**

**print("\nUCS traversal:")**

**ucs(graph, 'A', 'F')**

**ASSIGNMENT -5**

Consider the following 2-SAT problem with 4 Boolean variables a, b, c, d: F=(¬a∨d)𝖠(c∨b) 𝖠 ( ¬c∨¬d) 𝖠 ( ¬d∨¬b) 𝖠 (¬a∨¬d) The MOVEGEN function to generate new solution be arbitrary changing value of any one variable Let the candidate solution be of the order (abcd) and the initial candidate solution be (1111). Let heuristic to evaluate each solution be number of clauses satisfied in the formula. Apply Simulated Annealing (Consider T= 500 and cooling function = T-50) (Assume the following 3 random numbers:0.655,0.254.0.432) Accept every good move and accept a bad move if probability is greater than 50%.

ChatGPT

**ChatGPT**

To solve the 2-SAT problem using Simulated Annealing with the given parameters, we'll proceed as follows:

1. Initialize the candidate solution as (1111) and set the initial temperature T = 500.
2. Define a MOVEGEN function to generate new solutions by randomly changing the value of any one variable.
3. Define a heuristic to evaluate each solution as the number of clauses satisfied in the formula.
4. Repeat the following steps until the temperature drops below a certain threshold: a. Generate a new solution using the MOVEGEN function. b. Calculate the change in heuristic value (ΔH) between the new solution and the current solution. c. If the new solution is better (ΔH < 0), accept it as the current solution. d. If the new solution is worse (ΔH > 0), accept it with a probability determined by the Metropolis criterion and the current temperature. e. Update the temperature using the cooling function.
5. Return the final solution.

**final\_s**

**import random**

**import math**

**# Define the 2-SAT formula**

**def sat\_formula(a, b, c, d):**

**return (not a or d) and (c or b) and (not c or not d) and (not d or not b) and (not a or not d)**

**# Define the MOVEGEN function to generate new solutions**

**def movegen(solution):**

**# Randomly change the value of one variable in the solution**

**idx = random.randint(0, 3)**

**new\_solution = list(solution)**

**new\_solution[idx] = 1 - new\_solution[idx]**

**return tuple(new\_solution)**

**# Define the heuristic to evaluate each solution**

**def evaluate(solution):**

**a, b, c, d = solution**

**clauses\_satisfied = sat\_formula(a, b, c, d)**

**return sum(clauses\_satisfied)**

**# Define the cooling function**

**def cool(T):**

**return T - 50**

**# Simulated Annealing algorithm**

**def simulated\_annealing(initial\_solution, T, cooling\_func, random\_numbers):**

**current\_solution = initial\_solution**

**current\_energy = evaluate(current\_solution)**

**while T > 0:**

**# Generate a new solution**

**new\_solution = movegen(current\_solution)**

**new\_energy = evaluate(new\_solution)**

**# Calculate the change in energy**

**delta\_energy = new\_energy - current\_energy**

**# Accept the new solution if it's better or with a probability determined by the Metropolis criterion**

**if delta\_energy < 0 or random\_numbers[0] < math.exp(-delta\_energy / T):**

**current\_solution = new\_solution**

**current\_energy = new\_energy**

**# Update temperature**

**T = cooling\_func(T)**

**return current\_solution**

**# Initial candidate solution and parameters**

**initial\_solution = (1, 1, 1, 1)**

**T = 500**

**cooling\_func = cool**

**random\_numbers = [0.655, 0.254, 0.432]**

**# Apply simulated annealing**

**final\_solution = simulated\_annealing(initial\_solution, T, cooling\_func, random\_numbers)**

**print("Final solution:", final\_solution)olution**)

A thief enters a house for robbing it. He can carry a maximal weight of 9 kg into his bag. There are 4 items in the house with the following weights and values. The thief has to plan the items he should take to maximize the total value if he either takes the item completely or leaves it completely? Item Item Name Weight (in Kg) Value (in $) A Mirror 2 3 B Silver Nugget 3 5 C Painting 4 7 D Vase 5 9 The problem is solved using Genetic Algorithm with population size 4 and each individual encoded as {XA, XB, XC, XD} where Xi ={0,1 } and i=A, B, C, D. Consider initial population as 1111, 1000, 1010, and 1001. Generate the population for next iteration as follows: Select the 1st and 2nd fittest individual as it is in the next iteration. Apply 1-point crossover in the middle between 3rd and 4th fittest chromosome followed by single bit mutation of first offspring (produced through crossover). Bit chosen for mutation follows this cyclic order {XC,XA,XD,XB} Output the result after four iterations.

ChatGPT

**ChatGPT**

To solve this problem using a Genetic Algorithm, we'll follow these steps:

1. Initialize the population with the given initial population.
2. Evaluate the fitness of each individual in the population based on the total value of items they can carry.
3. Select the fittest individuals for reproduction (crossover).
4. Perform crossover to create new offspring.
5. Perform mutation on the offspring to introduce diversity.
6. Replace the weakest individuals in the population with the new offspring.
7. Repeat steps 2-6 for a specified number of iterations.

Let's implement this algorithm:

**import random**

**# Define the items**

**items = {'A': (2, 3), 'B': (3, 5), 'C': (4, 7), 'D': (5, 9)}**

**# Define the initial population**

**population = ['1111', '1000', '1010', '1001']**

**# Function to calculate fitness of an individual**

**def calculate\_fitness(individual):**

**total\_weight = 0**

**total\_value = 0**

**for i in range(len(individual)):**

**if individual[i] == '1':**

**total\_weight += items[chr(65 + i)][0]**

**total\_value += items[chr(65 + i)][1]**

**return total\_value if total\_weight <= 9 else 0**

**# Function to perform 1-point crossover**

**def crossover(parent1, parent2):**

**crossover\_point = len(parent1) // 2**

**offspring1 = parent1[:crossover\_point] + parent2[crossover\_point:]**

**offspring2 = parent2[:crossover\_point] + parent1[crossover\_point:]**

**return offspring1, offspring2**

**# Function to perform mutation**

**def mutate(individual):**

**mutation\_order = ['XC', 'XA', 'XD', 'XB']**

**mutation\_index = mutation\_order.index(individual[2])**

**mutation\_index = (mutation\_index + 1) % len(mutation\_order)**

**mutation\_char = mutation\_order[mutation\_index]**

**return individual[:2] + mutation\_char + individual[3:]**

**# Genetic Algorithm**

**def genetic\_algorithm(population, iterations):**

**for \_ in range(iterations):**

**# Calculate fitness for each individual**

**fitness\_scores = [(individual, calculate\_fitness(individual)) for individual in population]**

**# Sort individuals by fitness**

**sorted\_population = sorted(fitness\_scores, key=lambda x: x[1], reverse=True)**

**# Select fittest individuals**

**selected\_individuals = [individual for individual, \_ in sorted\_population[:2]]**

**# Perform crossover**

**offspring1, offspring2 = crossover(sorted\_population[2][0], sorted\_population[3][0])**

**# Perform mutation on first offspring**

**offspring1 = mutate(offspring1)**

**# Replace weakest individuals with offspring**

**population = selected\_individuals + [offspring1, offspring2]**

**return population**

**# Run the genetic algorithm for 4 iterations**

**final\_population = genetic\_algorithm(population, 4)**

**# Output the result**

**print("Final population after 4 iterations:")**

**for individual in final\_population:**

**print(individual)**

1 Solve the given 0/1 knapsack problem by considering the following points: Name Weight Value A 45 3 B 40 5 C 50 8 D 90 10 Chromosome is a 4-bit string. – {xA xB xC xD} Population size = 4, Maximum Capacity of the bag (W) = 100. First two fittest chromosomes selected as it is. 3 rd and 4th fittest use for one-point crossover in the middle followed by single bit mutation of first offspring. Bits chosen for mutation follows this cyclic order (xD, xC, xB, xA). Initial population: {1 1 1 1, 1 0 0 0, 1 0 1 0, 1 0 0 1}. Output the result after 10 iterations.

ChatGPT

**ChatGPT**

To solve the given 0/1 knapsack problem using a Genetic Algorithm, we'll follow these steps:

1. Initialize the population with the given initial population.
2. Evaluate the fitness of each individual in the population based on the total value of items they can carry without exceeding the maximum capacity.
3. Select the fittest individuals for reproduction (crossover).
4. Perform crossover to create new offspring.
5. Perform mutation on the offspring to introduce diversity.
6. Replace the weakest individuals in the population with the new offspring.
7. Repeat steps 2-6 for a specified number of iterations.

**import random**

**# Define the items**

**items = {'A': (45, 3), 'B': (40, 5), 'C': (50, 8), 'D': (90, 10)}**

**# Define the maximum capacity of the bag**

**W = 100**

**# Define the initial population**

**population = ['1111', '1000', '1010', '1001']**

**# Function to calculate fitness of an individual**

**def calculate\_fitness(individual):**

**total\_weight = 0**

**total\_value = 0**

**for i in range(len(individual)):**

**if individual[i] == '1':**

**total\_weight += items[chr(65 + i)][0]**

**total\_value += items[chr(65 + i)][1]**

**return total\_value if total\_weight <= W else 0**

**# Function to perform 1-point crossover**

**def crossover(parent1, parent2):**

**crossover\_point = len(parent1) // 2**

**offspring1 = parent1[:crossover\_point] + parent2[crossover\_point:]**

**offspring2 = parent2[:crossover\_point] + parent1[crossover\_point:]**

**return offspring1, offspring2**

**# Function to perform mutation**

**def mutate(individual):**

**mutation\_order = ['xD', 'xC', 'xB', 'xA']**

**mutation\_index = mutation\_order.index(individual[3])**

**mutation\_index = (mutation\_index + 1) % len(mutation\_order)**

**mutation\_char = mutation\_order[mutation\_index]**

**return individual[:3] + mutation\_char**

**# Genetic Algorithm**

**def genetic\_algorithm(population, iterations):**

**for \_ in range(iterations):**

**# Calculate fitness for each individual**

**fitness\_scores = [(individual, calculate\_fitness(individual)) for individual in population]**

**# Sort individuals by fitness**

**sorted\_population = sorted(fitness\_scores, key=lambda x: x[1], reverse=True)**

**# Select fittest individuals**

**selected\_individuals = [individual for individual, \_ in sorted\_population[:2]]**

**# Perform crossover**

**offspring1, offspring2 = crossover(sorted\_population[2][0], sorted\_population[3][0])**

**# Perform mutation on first offspring**

**offspring1 = mutate(offspring1)**

**# Replace weakest individuals with offspring**

**population = selected\_individuals + [offspring1, offspring2]**

**return population**

**# Run the genetic algorithm for 10 iterations**

**final\_population = genetic\_algorithm(population, 10)**

**# Output the result**

**print("Final population after 10 iterations:")**

**for individual in final\_population:**

**print(individual)**