Machine Learning

1) Write a program to implement S-algorithm to find the general hypothesis

```
table = [
    ['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']
]
hypothesis=['null', 'null', 'null', 'null', 'null', 'null']
h1=['null', 'null', 'null', 'null', 'null', 'null']
for i in range(4):
    if table[i][6]=='yes' and hypothesis==['null', 'null', 'null', 'null', 'null', 'null']:
        hypothesis=table[i][:6]
    elif table[i][6]=='yes':
        h1=table[i][:6]
        for j in range(6):
            if hypothesis[j]!=h1[j]:
                hypothesis[j]='?'
print(hypothesis)
```

Output:

```
['sunny', 'warm', '?', 'strong', '?', '?']
```

2) Write a program to implement candidate elimination algorithm

```
table = [
    ['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']
]
G = [
    ['?', '?', '?', '?' ,'?' ,'?'],
    ['?', '?', '?', '?' ,'?' ,'?'],
    ['?', '?', '?', '?' ,'?' ,'?'],
    ['?', '?', '?', '?' ,'?' ,'?'],
    ['?', '?', '?', '?' ,'?' ,'?'],
    ['?', '?', '?', '?' ,'?' ,'?']
]
S = ['null', 'null', 'null', 'null', 'null', 'null']
for i in range(4):
    if table[i][6]=='yes' and S==['null', 'null', 'null', 'null', 'null', 'null']:
        S=table[i][:6]
    elif table[i][6]=='yes':
        S1=table[i][:6]
```

```
        for j in range(6):
            if S[j]!=S1[j]:
                S[j]='?'
    else:
        for j in range(6):
            if table[i][j]!=S[j]:
                for k in range(6):
                    if G[k]==['?', '?', '?', '?' ,'?' ,'?']:
                        G[k][j]=S[j]
                        break
for i in range(6):
    for j in range(6):
        if G[i][j] not in S and G[i][j]!='?':
            G[i][j]='?'
print("The General Hypothesis is : ")
for i in range(6):
    if G[i]!=['?', '?', '?', '?' ,'?' ,'?']:
        print(G[i])
print("The Specific Hypothesis is : ")
print(S)
```

Output:

```
The General Hypothesis is :
['sunny', '?', '?', '?', '?', '?']
['?', 'warm', '?', '?', '?', '?']
The Specific Hypothesis is :
['sunny', 'warm', '?', 'strong', '?', '?']
```

3) The probability that it is Friday and that a student is absent is 3 %. Since there are 5 school days in a week, the probability that it is Friday is 20 %. What is the probability that a student is absent given that today is Friday? Apply Baye's rule in python to get the result. (Ans: 15%)

```
def calculate_probability_A_given_B(P_A, P_B, P_B_given_A):
    P_A_given_B = (P_B_given_A * P_A) / P_B
    return P_A_given_B
P_A = 0.03  # Probability that a student is absent (P(A))
P_B = 0.20  # Probability that today is Friday (P(B))
P_B_given_A = 1.0  # Probability that today is Friday given that a student is absent (P(B/A))
# Calculate the probability that a student is absent given that today is Friday (P(A/B))
P_A_given_B = calculate_probability_A_given_B(P_A, P_B, P_B_given_A)
print("The probability that a student is absent given that today is Friday is:", P_A_given_B)
```

Output:

```
The probability that a student is absent given that today is Friday is: 0.15
```

4) Extract the data from database using python

```
# Open the CSV file
with open("testfile.csv", "r") as file:
    # Read the lines of the CSV file
```

```
    lines = file.readlines()

    # Extract the data from the lines
    data = []
    for line in lines:
        # Remove newline characters and split the line into values
        values = line.strip().split(",")

        # Append the values to the data list
        data.append(values)

# Display the first few rows of the data
for row in data[:5]:
    print(row)
```

Output:

```
['name', 'age', 'city']
['John', '25', 'NewYork']
['Alice', '32', 'San Francisco']
['Michael', '45', 'Chicago']
```

5) Implement k-nearest neighbors classification using python

```
def classify(points,p,k):
    distance=[]
    for g in points:
        for xy in points[g]:
            eucdis=((xy[0]-p[0])**2+(xy[1]-p[1])**2)**0.5
            distance.append((eucdis,g))
    distance=sorted(distance)[:k]
    f1,f2=0,0
    for d in distance:
        if d[1]==0:
            f1+=1
        elif d[1]==1:
            f2+=1
    return 0 if f1>f2 else 1

points={1:[(1,1),(2,2),(3,1)],0:[(5,3),(4,4),(6,5)]}
p=(1,2)
k=3
ans=classify(points,p,k)
print(f"The value classified to unknown pointer is:{ans}")
```

Output:

```
The value classified to unknown pointer is:1
```

6) Given the following data, which specify classifications for nine combinations of VAR1 and VAR2 predict a classification for a case where VAR1=0.906 and VAR2=0.606,

using the result of k-means clustering with 3 means (i.e., 3 centroids)

VAR1	**VAR2**	**CLASS**
1.713	**1.586**	**0**
0.180	**1.786**	**1**
0.353	**1.240**	**1**
0.940	**1.566**	**0**
1.486	**0.759**	**1**
1.266	**1.106**	**0**
1.540	**0.419**	**1**
0.459	**1.799**	**1**
0.773	**0.186**	**1**

```
def euclidean_distance(point1, point2):
    return ((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2)**0.5

def k_means_classification(data, unknown_case, k):
    centroids = data[:k]

    min_distance = float('inf')
    assigned_centroid = None
    for centroid in centroids:
        distance = euclidean_distance(unknown_case, centroid)
        if distance < min_distance:
            min_distance = distance
            assigned_centroid = centroid

    prediction = None
    for i in range(len(data)):
        if data[i] == assigned_centroid:
            prediction = data[i][-1]
            break

    return prediction

data = [
    [1.713, 1.586, 0],
    [0.180, 1.786, 1],
    [0.353, 1.240, 1],
    [0.940, 1.566, 0],
    [1.486, 0.759, 1],
    [1.266, 1.106, 0],
    [1.540, 0.419, 1],
    [0.459, 1.799, 1],
    [0.773, 0.186, 1]
]

unknown_case = [0.906, 0.606]
k = 3
```

```
prediction = k_means_classification(data, unknown_case, k)

print("Predicted Classification:", prediction)
```

Output:
Predicted Classification: 1

7) The following training examples map descriptions of individuals onto high, medium and low credit-worthiness.
medium skiing design single twenties no -> highRisk
high golf trading married forties yes -> lowRisk
low speedway transport married thirties yes -> medRisk
medium football banking single thirties yes -> lowRisk
high flying media married fifties yes -> highRisk
low football security single twenties no -> medRisk
medium golf media single thirties yes -> medRisk
medium golf transport married forties yes -> lowRisk
high skiing banking single thirties yes -> highRisk
low golf unemployed married forties yes -> highRisk
Input attributes are (from left to right) income, recreation, job, status, age-group, home-owner.
Find the unconditional probability of `golf' and the conditional probability of `single' given `medRisk' in the dataset?

```
train_data = [
    ['medium', 'skiing', 'design', 'single', 'twenties', 'no', 'highRisk'],
    ['high', 'golf', 'trading', 'married', 'forties', 'yes', 'lowRisk'],
    ['low', 'speedway', 'transport', 'married', 'thirties', 'yes', 'medRisk'],
    ['medium', 'football', 'banking', 'single', 'thirties', 'yes', 'lowRisk'],
    ['high', 'flying', 'media', 'married', 'fifties', 'yes', 'highRisk'],
    ['low', 'football', 'security', 'single', 'twenties', 'no', 'medRisk'],
    ['medium', 'golf', 'media', 'single', 'thirties', 'yes', 'medRisk'],
    ['medium', 'golf', 'transport', 'married', 'forties', 'yes', 'lowRisk'],
    ['high', 'skiing', 'banking', 'single', 'thirties', 'yes', 'highRisk'],
    ['low', 'golf', 'unemployed', 'married', 'forties', 'yes', 'highRisk']
]

num_golf = sum(1 for example in train_data if example[1] == 'golf')

uncond_prob_golf = num_golf / len(train_data)
print('Unconditional probability of \'golf\':', uncond_prob_golf)

num_single_medrisk = sum(1 for example in train_data if example[3] == 'single' and example[6] == 'medRisk')
num_medrisk = sum(1 for example in train_data if example[6] == 'medRisk')

cond_prob_single_medrisk = num_single_medrisk / num_medrisk
print('Conditional probability of \'single\' given \'medRisk\':', cond_prob_single_medrisk)
```

Output:
Unconditional probability of 'golf': 0.4

Conditional probability of 'single' given 'medRisk': 0.6666666666666666

8) Implement Perceptron Algorithm in python

```
# Training dataset
training_data = [
    ([2, 3], 0),
    ([4, 5], 0),
    ([1, 6], 0),
    ([6, 7], 1),
    ([8, 9], 1),
    ([9, 10], 1)
]

# Initialize weights and bias
weights = [0, 0]
bias = 0

# Learning rate
learning_rate = 0.1

# Perceptron training
epochs = 100
for _ in range(epochs):
    errors = 0
    for input_data, target in training_data:
        # Calculate activation
        activation = bias
        for i in range(len(input_data)):
            activation += weights[i] * input_data[i]

        # Apply step function
        if activation >= 0:
            prediction = 1
        else:
            prediction = 0

        # Update weights and bias
        if prediction != target:
            errors += 1
            error = target - prediction
            for i in range(len(weights)):
                weights[i] += learning_rate * error * input_data[i]
            bias += learning_rate * error

    # Check for convergence
    if errors == 0:
        break
```

```
# Test the trained perceptron
test_data = [
    [3, 4],
    [7, 8],
    [2, 7]
]

print("Test Results:")
for input_data in test_data:
    activation = bias
    for i in range(len(input_data)):
        activation += weights[i] * input_data[i]

    if activation >= 0:
        prediction = 1
    else:
        prediction = 0

    print(f"Input: {input_data}, Prediction: {prediction}")
```

Output:

```
Test Results:
Input: [3, 4], Prediction: 0
Input: [7, 8], Prediction: 1
Input: [2, 7], Prediction: 0
```

9)Implement an algorithm to demonstrate the significance of genetic algorithm

```
population_size = 100
chromosome_length = 20
mutation_rate = 0.5
generations = 10
def create_individual():
    return [random.randint(0, 1) for _ in range(chromosome_length)]
def evaluate_fitness(individual):
    return sum(individual)
def mutate(individual):
    for i in range(len(individual)):
        if random.random() < mutation_rate:
            individual[i] = 1 - individual[i]
    return individual
def crossover(parent1, parent2):
    point = random.randint(1, chromosome_length - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2
def genetic_algorithm():
    population = [create_individual() for _ in range(population_size)]
```

```
    for generation in range(generations):
        # Evaluate fitness of each individual
        fitness_scores = [evaluate_fitness(individual) for individual in population]

        # Select parents for reproduction
        parents = random.choices(population, weights=fitness_scores, k=2)

        # Perform crossover and mutation to create new offspring
        offspring = crossover(parents[0], parents[1])
        offspring = [mutate(child) for child in offspring]

        # Replace least fit individuals with offspring
        population.extend(offspring)
        population = sorted(population, key=evaluate_fitness, reverse=True)
        population = population[:population_size]

        # Print best individual of each generation
        best_individual = population[0]
        print(f"Generation {generation + 1}: {best_individual}, Fitness: {evaluate_fitness(best_individual)}")
if __name__ == '__main__':
    genetic_algorithm()
```

Output:

Generation 1: [1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], Fitness: 16
Generation 2: [1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], Fitness: 16
Generation 3: [1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], Fitness: 16
Generation 4: [1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], Fitness: 16
Generation 5: [1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], Fitness: 16
Generation 6: [1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], Fitness: 16
Generation 7: [1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], Fitness: 16
Generation 8: [1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], Fitness: 16
Generation 9: [1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], Fitness: 16
Generation 10: [1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], Fitness: 16

10) Implement decision-tree algorithm

```
# Node class for Decision Tree
class Node:
    def __init__(self, feature=None, threshold=None, label=None):
        self.feature = feature
        self.threshold = threshold
        self.label = label
        self.left = None
        self.right = None

# Function to calculate Gini Index
def gini_index(groups, classes):
    total_samples = sum([len(group) for group in groups])
    gini = 0.0
```

```
    for group in groups:
        group_size = float(len(group))
        if group_size == 0:
            continue
        score = 0.0
        for class_val in classes:
            p = [row[-1] for row in group].count(class_val) / group_size
            score += p * p
        gini += (1.0 - score) * (group_size / total_samples)
    return gini

# Function to split the dataset based on a feature and threshold
def split_dataset(dataset, feature, threshold):
    left, right = [], []
    for row in dataset:
        if row[feature] < threshold:
            left.append(row)
        else:
            right.append(row)
    return left, right

# Function to find the best split point for a dataset
def find_best_split(dataset):
    class_values = list(set(row[-1] for row in dataset))
    best_feature, best_threshold, best_gini, best_groups = None, None, float('inf'), None
    for feature in range(len(dataset[0]) - 1):
        for row in dataset:
            groups = split_dataset(dataset, feature, row[feature])
            gini = gini_index(groups, class_values)
            if gini < best_gini:
                best_feature, best_threshold, best_gini, best_groups = feature, row[feature], gini, groups
    return {'feature': best_feature, 'threshold': best_threshold, 'groups': best_groups}

# Function to create a terminal node with the most common class label
def create_terminal_node(group):
    class_labels = [row[-1] for row in group]
    return max(set(class_labels), key=class_labels.count)

# Recursive function to build the Decision Tree
def build_tree(node, max_depth, min_size, depth):
    left, right = node['groups']
    del(node['groups'])

    # Check for no split
    if not left or not right:
        node['left'] = node['right'] = create_terminal_node(left + right)
        return

    # Check for maximum depth
```

```
    if depth >= max_depth:
        node['left'], node['right'] = create_terminal_node(left), create_terminal_node(right)
        return

    # Process left child
    if len(left) <= min_size:
        node['left'] = create_terminal_node(left)
    else:
        node['left'] = find_best_split(left)
        build_tree(node['left'], max_depth, min_size, depth + 1)

    # Process right child
    if len(right) <= min_size:
        node['right'] = create_terminal_node(right)
    else:
        node['right'] = find_best_split(right)
        build_tree(node['right'], max_depth, min_size, depth + 1)

# Function to build the Decision Tree
def decision_tree(dataset, max_depth, min_size):
    root = find_best_split(dataset)
    build_tree(root, max_depth, min_size, 1)
    return root

# Function to make a prediction with the Decision Tree
def predict(node, row):
    if row[node['feature']] < node['threshold']:
        if isinstance(node['left'], dict):
            return predict(node['left'], row)
        else:
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else:
            return node['right']

# Example usage
dataset = [
    [2.771244718, 1.784783929, 0],
    [1.728571309, 1.169761413, 0],
    [3.678319846, 2.81281357, 0],
    [3.961043357, 2.61995032, 0],
    [2.999208922, 2.209014212, 0],
    [7.497545867, 3.162953546, 1],
    [9.00220326, 3.339047188, 1],
    [7.444542326, 0.476683375, 1],
    [10.12493903, 3.234550982, 1],
    [6.642287351, 3.319983761, 1]
```

```
]

tree = decision_tree(dataset, max_depth=3, min_size=1)

# Test the Decision Tree
test_data = [
    [3.095607236, 1.783283623],
    [8.675418651, 0.242820951],
    [7.673756466, 3.508563011]
]

print("Test Results:")
for data in test_data:
    prediction = predict(tree, data)
    print(f"Input: {data}, Prediction: {prediction}")
```

Output:

```
Test Results:
Input: [3.095607236, 1.783283623], Prediction: 0
Input: [8.675418651, 0.242820951], Prediction: 1
Input: [7.673756466, 3.508563011], Prediction: 1
```

11) Implement Naïve Bayes theorem to classify the English text

```
# Define the training dataset
training_data = [
    ["I love this car", "positive"],
    ["This view is amazing", "positive"],
    ["I feel great", "positive"],
    ["I'm not happy with the product", "negative"],
    ["This is a terrible place", "negative"],
    ["I don't like this movie", "negative"]
]

# Create an empty vocabulary set
vocabulary = set()

# Add words from training data to the vocabulary
for data in training_data:
    sentence = data[0]
    words = sentence.split()
    vocabulary.update(words)

# Count the occurrences of each class in the training data
class_counts = {}
for data in training_data:
    label = data[1]
    if label in class_counts:
        class_counts[label] += 1
```

```
    else:
        class_counts[label] = 1

# Compute the probabilities of each class
total_data = len(training_data)
class_probabilities = {}
for label, count in class_counts.items():
    class_probabilities[label] = count / total_data
# Create a dictionary to store word probabilities
word_probabilities = {}

# Count the occurrences of each word in each class
word_counts = {}
for data in training_data:
    sentence = data[0]
    label = data[1]
    words = sentence.split()

    if label not in word_counts:
        word_counts[label] = {}

    for word in words:
        if word in word_counts[label]:
            word_counts[label][word] += 1
        else:
            word_counts[label][word] = 1

# Compute the probabilities of each word given a class
for label in word_counts:
    word_probabilities[label] = {}
    total_words = sum(word_counts[label].values())
    for word in vocabulary:
        if word in word_counts[label]:
            word_probabilities[label][word] = word_counts[label][word] / total_words
        else:
            word_probabilities[label][word] = 0.0

def classify_text(text):
    words = text.split()

    # Initialize the class probabilities
    class_scores = {}

    for label in class_probabilities:
        # Start with the class probability
        score = class_probabilities[label]

        for word in words:
            # Check if the word is in the vocabulary
```

```
            if word in vocabulary:
                # Multiply the score by the word probability
                score *= word_probabilities[label][word]

        class_scores[label] = score

    # Select the class with the highest probability
    predicted_class = max(class_scores, key=class_scores.get)
    return predicted_class

# Test the classifier
test_text = "I like this place"
predicted_label = classify_text(test_text)
print("Predicted Label:", predicted_label)
```

Output:
Predicted Label: negative