UNIVERSITY SCHOOL OF INFORMATION, COMMUNICATION AND TECHNOLOGY
GURU GOBIND SINGH INDRAPRASTHA UNIVERSITY

PRACTICAL FILE

Subject: Artificial Intelligence Lab

Subject Code: ICT311P

Submitted to: MS. Heena Kalim

Submitted by: SACHIN YADAV

Course: B.TECH. CSE 5th Semester

Enrolment No: 03916403221

S.No.	Name of Experiment	Date	Sign
1.	Write a program to implement Depth First Search.		
2.	Write a program to implement Breadth First Search.		
3.	Write a program to implement the water jug problem.		
4.	Write a program to implement the 8-queen problem.		
5.	Write a program to implement A* Algorithms.		
6.	Write a program to implement AO* Algorithms.		
7.	Write a program to find the mean, mode, median, and standard deviation.		
8.	Write a program to implement linear regression.		
9.	Write a program to implement Logistic Regression classification.		
10.	Write a program to implement k-nearest neighbors classification.		
11.	Write a program to implement Naïve Bayes Classifier Algorithm.		
12.	Write a program to implement Decision Tree Classification Algorithm.		
13.	Write a program to implement K – Mean Clustering.		
14.	Write a program to reduce the number of dimensions using principal component analysis (PCA).		
15.	Write a program to implement a simple feedforward neural network for a basic task like binary classification.		
16.	Write a program to evaluate the performance of any classification model using metrics like accuracy, precision, recall, F1-score, etc.		
17.	Write a program to construct a simple Bayesian network.		
18.	Write a program to implement a deep learning model for image classification.		

EXPERIMENT 1: Write a program to implement Depth First Search.

THEORY

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. Extra memory, usually a stack, is needed to keep track of the nodes discovered so far along a specified branch which helps in backtracking of the graph.

MAIN.PY

```
# Program to implement Depth First Search.

class Graph:
  """A class representing graph as an adjacency list, having vertices as
numeric values begining from 0"""

  def __init__(self, vertices):
    self.graph = [[] for i in range(vertices)]
    self.vertices = [i for i in range(vertices)]
    self.size = vertices;

  def addEdge(self, u, v):
    """Add directed edge from node u to node v"""
    self.graph[u].append(v)

  def addVertex(self):
    """Add another vertex to graph"""
    self.graph.append([])
    self.vertices.append(self.size)
    self.size += 1

  def printDFT(self):
    color = [-1]*self.size
    pi = [None]*self.size
    d = [None]*self.size # discoverd time
    f = [None]*self.size # finished time
    time = [0];

    def dfs(u):
      color[u] += 1
      time[0] += 1
      d[u] = time[0]

      for v in self.graph[u]:
        if color[v] == -1:
          pi[v] = u
          dfs(v)

      color[u] += 1
      time[0] += 1
      f[u] = time[0]

      print("Node:", u, "discovered: ", d[u], "processed: ", f[u])

    for u in self.vertices:
      if color[u] == -1:
        dfs(u)

# Driver code
if __name__ == "__main__":
  myGraph = Graph(3)
  myGraph.addEdge(0, 1)
  myGraph.addEdge(1, 2)
  myGraph.addEdge(2, 0)
  myGraph.addEdge(2, 1)

  myGraph.printDFT()
```

OUTPUT

```
Node: 2 discovered:  3 processed:  4
Node: 1 discovered:  2 processed:  5
Node: 0 discovered:  1 processed:  6
```

EXPERIMENT 2: Write a program to implement Breadth First Search.

THEORY

Breadth-first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored.

MAIN.PY

```
# Program to implement Breadth First Search.

class Graph:
  """A class representing graph as an adjacency list, having vertices as
numeric values begining from 0"""

  def __init__(self, vertices):
    self.graph = [[] for i in range(vertices)]
    self.vertices = [i for i in range(vertices)]
    self.size = vertices;

  def addEdge(self, u, v):
    """Add directed edge from node u to node v"""
    self.graph[u].append(v)

  def addVertex(self):
    """Add another vertex to graph"""
    self.graph.append([])
    self.vertices.append(self.size)
    self.size += 1

  def printBFT(self, s):
    """Print BFT starting from vertex s"""
    print("BFT:", end=" ")
    color = [-1]*self.size
    pi = [None]*self.size
    d = [None]*self.size # smallest distance in terms of number of edges from s
to v
    d[s] = 0
    color[s] += 1;
    q = [s]
    while len(q) != 0:
      u = q.pop(0)
      for v in self.graph[u]:
        if color[v] == -1:
          d[v] = d[u] + 1
          pi[v] = u
          color[v] += 1
          q.append(v)
      color[u] += 1
      print(u, end=" ")

# Driver code
if __name__ == "__main__":
  myGraph = Graph(3)
  myGraph.addEdge(0, 1)
  myGraph.addEdge(1, 2)
  myGraph.addEdge(2, 0)
  myGraph.addEdge(2, 1)

  myGraph.printBFT(0)
```

OUTPUT

```
BFT: 0 1 2
```

EXPERIMENT 3: Write a program to implement the water jug problem.

THEORY

It's a test of problem-solving and state space search, where the initial state is both jugs empty and the goal is to reach a state where one jug holds 'z' litres. Various operations like filling, emptying, and pouring between jugs are used to find an efficient sequence of steps to achieve the desired water measurement.

MAIN.PY

```python
# Program to implement the solution to water jug problem.

class Solution:
  def __init__(self, a, b, target):
    self.a = a
    self.b = b
    self.m = {(0, 0):None, (a, 0):(0, 0), (0, b):(0, 0)} # map to keep track of already formed pairs for optimization
    self.path = [] # path leading to solution if it exists
    self.queue = [(0, 0), (a, 0), (0, b)] # to implement bfs, initialized with (0,0) representing empty jugs
    self.solve(a, b, target)


  def valid(self, u):
    if (u[0], u[1]) in self.m: return False
    if (u[0] > self.a or u[1] > self.b or u[0] < 0 or u[1] < 0): return False
    return True


  def solve(self, a, b, target):
    isSolveable = False

    while(len(self.queue) != 0):
      u = self.queue.pop(0) # current state


      if(u[0] == target or u[1] == target):
        isSolveable = True
        if u[0] == target:
          if u[1]:
            self.m[(u[0], 0)] = u
            u = (u[0], 0)
        else:
          if u[0]:
            self.m[(0, u[1])] = u
            u = (0, u[1])

        while(u):
          self.path.append(u)
          u = self.m[u]

        self.path.reverse()
        [print(v[0], v[1]) for v in self.path]

        break

      self.queue.append((u[0], b)) # filled second jug
      if (u[0], b) not in self.m: self.m[(u[0], b)] = u
      self.queue.append((a, u[1])) # filled first jug
      if (a, u[1]) not in self.m: self.m[(a, u[1])] = u

      t = (u[0]+u[1], 0)
      if self.valid(t):
        self.queue.append(t)
        if t not in self.m: self.m[t] = u

      t = (a, u[0]+u[1]-a)
      if self.valid(t):
        self.queue.append(t)
        if t not in self.m: self.m[t] = u

      t = (0, u[0]+u[1])
      if self.valid(t):
        self.queue.append(t)
        if t not in self.m: self.m[t] = u

      t = (u[0]+u[1]-b, b)
      if self.valid(t):
        self.queue.append(t)
        if t not in self.m: self.m[t] = u


    if not isSolveable: print("Solution is not possible")


if __name__ == "__main__":
  Solution(4, 3, 2)
```

OUTPUT

```
0 0
0 3
3 0
3 3
4 2
0 2
```

EXPERIMENT 4: Write a program to implement the 8-queen problem.

THEORY

Backtracking is a recursive approach for solving any problem where we must search among all the possible solutions following some constraints. More precisely, we can say that it is an improvement over the brute force technique. In this blog, we will learn one popular DSA question: 8 queens problem using Backtracking.

MAIN.PY

```python
# program to implement solution for the 8-queen problem.

class Solution:
  def solveNQueens(slef, n: int) -> list[list[str]]:
    col = set()
    posDiagonal = set() # r+c = const
    negDiagonal = set() # r-c = const
    res = []
    board = [['.']*n for i in range(n)]

    def backtrack(r):
      if r == n:
        copy = [" ".join(row) for row in board]
        res.append(copy)
        return

      for c in range(n):
        if c in col or r+c in posDiagonal or r-c in negDiagonal:
          continue

        col.add(c)
        posDiagonal.add(r+c)
        negDiagonal.add(r-c)
        board[r][c] = 'Q'

        backtrack(r + 1)

        board[r][c] = '.'
        negDiagonal.remove(r-c)
        posDiagonal.remove(r+c)
        col.remove(c)

    backtrack(0)
    return res
```

```python
# driver code
if __name__ == "__main__":
  solnSet = Solution().solveNQueens(8)
  n = 0
  for soln in solnSet:
    n += 1
    print("Arrangement:", n)
    for row in soln:
      print(row)
    print()
```

OUTPUT

```
Arrangement: 1
Q.......
....Q...
.......Q
.....Q..
..Q.....
......Q.
.Q......
...Q....

Arrangement: 2
Q.......
.....Q..
.......Q
..Q.....
......Q.
...Q....
.Q......
....Q...

.
.
.

Arrangement: 92
.......Q
...Q....
Q.......
..Q.....
.....Q..
.Q......
......Q.
....Q...
```

EXPERIMENT 5: Write a program to implement A* Algorithms.

THEORY

It is a searching algorithm that is used to find the shortest path between an initial and a final point. It is a handy algorithm that is often used for map traversal to find the shortest path to be taken. A* was initially designed as a graph traversal problem, to help build a robot that can find its own course.

MAIN.PY

```
import heapq
class Node:
 def __init__(self, state, parent=None, cost=0, heuristic=0):
 self.state = state
 self.parent = parent
 self.cost = cost
 self.heuristic = heuristic
 def __lt__(self, other):
 return (self.cost + self.heuristic) < (other.cost + other.heuristic)
def astar(graph, start, goal):
 open_list = []
 closed_set = set()
 start_node = Node(start, None, 0, heuristic_cost(start, goal))
 heapq.heappush(open_list, start_node)
 while open_list:
 current_node = heapq.heappop(open_list)
 if current_node.state == goal:
 return reconstruct_path(current_node)
 closed_set.add(current_node.state)
 for neighbor, cost in graph[current_node.state]:
 if neighbor in closed_set:
 continue
 tentative_cost = current_node.cost + cost
 neighbor_node = Node(neighbor, current_node, tentative_cost,
heuristic_cost(neighbor, goal))
 if not any(node.cost < tentative_cost and node.state == neighbor for node in
open_list):
 heapq.heappush(open_list, neighbor_node)
 return None # No path found
def heuristic_cost(node, goal):
 # Example heuristic: Manhattan distance
 x1, y1 = node
 x2, y2 = goal
 return abs(x1 - x2) + abs(y1 - y2)
def reconstruct_path(node):
 path = []
 while node:
 path.insert(0, node.state)
 node = node.parent
 return path
# Example usage:
graph = {
 (0, 0): [((0, 1), 1), ((1, 0), 1)],
 (0, 1): [((0, 0), 1), ((0, 2), 1)],
 (0, 2): [((0, 1), 1), ((1, 2), 1)],
 (1, 0): [((0, 0), 1), ((1, 1), 1)],
 (1, 1): [((1, 0), 1), ((1, 2), 1)],
 (1, 2): [((0, 2), 1), ((1, 1), 1)],
}
start = (0, 0)
goal = (1, 2)
path = astar(graph, start, goal)
if path:
 print("Path found:", path)
else:
 print("No path found.")
```

OUTPUT

```
Path found: [(0, 0), (0, 1), (0, 2), (1, 2)]
```

EXPERIMENT 6: Write a program to implement AO* Algorithms.

THEORY

Best-first search is what the AO* algorithm does. The AO* method divides any given difficult problem into a smaller group of problems that are then resolved using the AND-OR graph concept. AND OR graphs are specialized graphs that are used in problems that can be divided into smaller problems.

MAIN.PY

```
import heapq
class Node:
 def __init__(self, state, parent=None, g=0, h=0, f=0):
 self.state = state
 self.parent = parent
 self.g = g # Cost from start to this node
 self.h = h # Heuristic estimate to the goal
 self.f = f # f = g + h
 def __lt__(self, other):
 return self.f < other.f
def aostar(graph, start, goal):
 open_list = []
 closed_set = set()
 start_node = Node(start)
 start_node.g = 0
 start_node.h = heuristic_cost(start, goal)
 start_node.f = start_node.g + start_node.h
 heapq.heappush(open_list, start_node)
 while open_list:
 current_node = heapq.heappop(open_list)
 if current_node.state == goal:
 return reconstruct_path(current_node)
 closed_set.add(current_node.state)
 for neighbor, cost in graph[current_node.state]:
 if neighbor in closed_set:
 continue
 tentative_g = current_node.g + cost
 neighbor_node = Node(neighbor)
 if neighbor_node not in open_list or tentative_g < neighbor_node.g:
 neighbor_node.parent = current_node
 neighbor_node.g = tentative_g
 neighbor_node.h = heuristic_cost(neighbor, goal)
 neighbor_node.f = neighbor_node.g + neighbor_node.h
 if neighbor_node not in open_list:
 heapq.heappush(open_list, neighbor_node)
 return None # No path found
def heuristic_cost(node, goal):
 # Example heuristic: Manhattan distance
 x1, y1 = node
 x2, y2 = goal
 return abs(x1 - x2) + abs(y1 - y2)
def reconstruct_path(node):
 path = []
 while node:
 path.insert(0, node.state)
 node = node.parent
 return path
# Example usage:
graph = {
 (0, 0): [((0, 1), 1), ((1, 0), 1)],
 (0, 1): [((0, 0), 1), ((0, 2), 1)],
 (0, 2): [((0, 1), 1), ((1, 2), 1)],
 (1, 0): [((0, 0), 1), ((1, 1), 1)],
 (1, 1): [((1, 0), 1), ((1, 2), 1)],
 (1, 2): [((0, 2), 1), ((1, 1), 1)],
}
start = (0, 0)
goal = (1, 2)
path = aostar(graph, start, goal)
if path:
 print("Path found:", path)
else:
 print("No path found.")
```

OUTPUT

```
Path found: [(0, 0), (0, 1), (0, 2), (1, 2)]
```

EXPERIMENT 7: Write a program to find the mean, mode, median, and standard deviation.

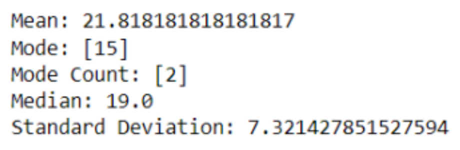
THEORY

Mean, median, and mode convey the average, middle, and most frequent values in data. The mean is the sum divided by the count, the median is the middle value, and the mode is the most common. Standard deviation gauges data spread around the mean, encapsulating variability. Together, they succinctly capture central tendency and dispersion characteristics in a dataset.

MAIN.PY

```python
import numpy as np
from scipy import stats

data = [12, 15, 15, 17, 18, 19, 22, 25, 30, 32, 35]
mean = np.mean(data)
mode_result = stats.mode(data)
# Access the mode value(s)
mode_value = mode_result.mode
mode_count = mode_result.count
print("Mean:", mean)
print("Mode:", mode_value)
print("Mode Count:", mode_count)
median = np.median(data)
std_dev = np.std(data)
print("Median:", median)
print("Standard Deviation:", std_dev)
```

OUTPUT



```
Mean: 21.818181818181817
Mode: [15]
Mode Count: [2]
Median: 19.0
Standard Deviation: 7.321427851527594
```

EXPERIMENT 8: Write a program to implement linear regression.

THEORY

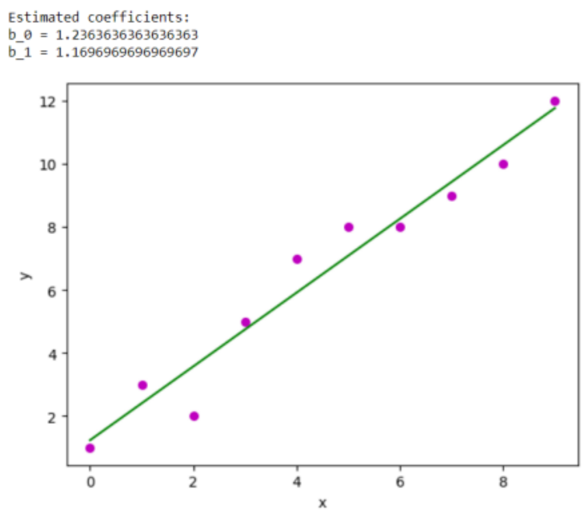
Linear regression analysis is used to predict the value of a variable based on the value of another variable. The variable you want to predict is called the dependent variable. The variable you are using to predict the other variable's value is called the independent variable.

MAIN.PY

```python
import numpy as np
import matplotlib.pyplot as plt


def estimate_coef(x, y):
    # number of observations/points
    n = np.size(x)
    # mean of x and y vector
    m_x = np.mean(x)
    m_y = np.mean(y)
    # calculating cross-deviation and deviation about x
    SS_xy = np.sum(y * x) - n * m_y * m_x
    SS_xx = np.sum(x * x) - n * m_x * m_x
    # calculating regression coefficients
    b_1 = SS_xy / SS_xx
    b_0 = m_y - b_1 * m_x
    return (b_0, b_1)


def plot_regression_line(x, y, b):
    # plotting the actual points as scatter plot
    plt.scatter(x, y, color="m", marker="o", s=30)
    # predicted response vector
    y_pred = b[0] + b[1] * x
    # plotting the regression line
    plt.plot(x, y_pred, color="g")
    # putting labels
    plt.xlabel("x")
    plt.ylabel("y")
    # function to show plot
    plt.show()


x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
y = np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])
# estimating coefficients
```

```
b = estimate_coef(x, y)
print(
    "Estimated coefficients:\nb_0 = {} \
\nb_1 = {}".format(
        b[0], b[1]
    )
)
# plotting regression line
plot_regression_line(x, y, b)
```

OUTPUT



```
Estimated coefficients:
b_0 = 1.2363636363636363
b_1 = 1.1696969696969697
```

EXPERIMENT 9: Write a program to implement Logistic Regression classification.

THEORY

Logistic regression is a statistical analysis method to predict a binary outcome, such as yes or no, based on prior observations of a data set. A logistic regression model predicts a dependent data variable by analyzing the relationship between one or more existing independent variables.

MAIN.PY

```
import pandas as pd
import numpy as np
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
def sigmoid(z):
 return 1 / (1 + np.exp(-z))
def cost_function(X, y, theta):
 m = len(y)
 h = sigmoid(np.dot(X, theta))
 epsilon = 1e-15 # Small value to prevent taking the logarithm of 0
 cost = (-1 / m) * np.sum(y * np.log(h + epsilon) + (1 - y) * np.log(1 - h +
epsilon))
 return cost
def gradient_descent(X, y, theta, alpha, iterations):
 m = len(y)
 costs = []
 for _ in range(iterations):
 h = sigmoid(np.dot(X, theta))
 gradient = np.dot(X.T, (h - y)) / m
 theta -= alpha * gradient
 cost = cost_function(X, y, theta)
 costs.append(cost)
 return theta, costs
wine = load_wine()
data = pd.DataFrame(data=np.c_[wine['data'], wine['target']],
columns=wine['feature_names'] + ['target'])
X = data.drop('target', axis=1)
y = data['target']
y = y.values.reshape(-1, 1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
X_train_scaled = np.c_[np.ones((X_train_scaled.shape[0], 1)), X_train_scaled]
X_test_scaled = np.c_[np.ones((X_test_scaled.shape[0], 1)), X_test_scaled]
theta = np.zeros((X_train_scaled.shape[1], 1))
alpha = 0.01
iterations = 1000
theta, costs = gradient_descent(X_train_scaled, y_train, theta, alpha,
iterations)
print('Learned parameters:')
print(theta)
predictions = sigmoid(np.dot(X_test_scaled, theta))
predictions = (predictions >= 0.5).astype(int)
accuracy = np.mean(predictions == y_test)
print(f'Accuracy: {accuracy * 100:.2f}%')
```

OUTPUT

```
Learned parameters: [[ 4.24363336] [-0.96411231] [ 1.68178213] [-0.15090813] [ 1.87973966] [-0.24934178] [-
1.95864786] [-2.90115963] [ 1.17606736] [-1.19265113] [ 1.69866703] [-2.18456384] [-2.96309004] [-2.09726936]]
Accuracy: 69.44%
```

EXPERIMENT 10: Write a program to implement k-nearest neighbors classification.

THEORY

The k-nearest neighbors algorithm, also known as KNN or k-NN, is a non-parametric, supervised learning classifier, which uses proximity to make classifications or predictions about the grouping of an individual data point

MAIN.PY

```
from csv import reader
from sys import exit
from math import sqrt
from operator import itemgetter
def load_data_set(filename):
 try:
 with open(filename, newline='') as iris:
 return list(reader(iris, delimiter=','))
 except FileNotFoundError as e:
 raise e
def convert_to_float(data_set, mode):
 new_set = []
 try:
 if mode == 'training':
 for data in data_set:
 new_set.append([float(x) for x in data[:len(data)-1]] + [data[len(data)-1]])
 elif mode == 'test':
 for data in data_set:
 new_set.append([float(x) for x in data])
 else:
 print('Invalid mode, program will exit.')
 exit()
 return new_set
 except ValueError as v:
 print(v)
 print('Invalid data set format, program will exit.')
 exit()
def get_classes(training_set):
 return list(set([c[-1] for c in training_set]))
def find_neighbors(distances, k):
 return distances[0:k]
def find_response(neighbors, classes):
 votes = [0] * len(classes)
 for instance in neighbors:
 for ctr, c in enumerate(classes):
```

```
 if instance[-2] == c:
 votes[ctr] += 1
 return max(enumerate(votes), key=itemgetter(1))
def knn(training_set, test_set, k):
 distances = []
 dist = 0
 limit = len(training_set[0]) - 1
 classes = get_classes(training_set)
 try:
 for test_instance in test_set:
 for row in training_set:
 for x, y in zip(row[:limit], test_instance):
 dist += (x-y) * (x-y)
 distances.append(row + [sqrt(dist)])
 dist = 0
 distances.sort(key=itemgetter(len(distances[0])-1))
 neighbors = find_neighbors(distances, k)
 index, value = find_response(neighbors, classes)
 print('The predicted class for sample ' + str(test_instance) + ' is : ' +
classes[index])
 print('Number of votes : ' + str(value) + ' out of ' + str(k))
 distances.clear()
 except Exception as e:
 print(e)
try:
 k = int(input('Enter the value of k : '))
 training_file = input('Enter name of training data file : ')
 test_file = input('Enter name of test data file : ')
 training_set = convert_to_float(load_data_set(training_file), 'training')
 test_set = convert_to_float(load_data_set(test_file), 'test')
 if not training_set:
 print('Empty training set')
 elif not test_set:
 print('Empty test set')
 elif k > len(training_set):
 print('Expected number of neighbors is higher than number of training data
instances')
 else:
 knn(training_set, test_set, k)
except ValueError as v:
 print(v)
except FileNotFoundError:
 print('File not found')
```

OUTPUT

```
Enter the value of k : 2

Enter name of training data file : iris-dataset.csv

Enter name of test data file : iris-test.csv

The predicted class for sample [4.3, 2.9, 1.7, 0.3] is : Iris-setosa

Number of votes : 2 out of 2

The predicted class for sample [4.6, 2.7, 1.5, 0.2] is : Iris-setosa

Number of votes : 2 out of 2

The predicted class for sample [5.3, 3.4, 1.6, 0.2] is : Iris-setosa

Number of votes : 2 out of 2

The predicted class for sample [5.2, 4.1, 1.5, 0.1] is : Iris-setosa

Number of votes : 2 out of 2

The predicted class for sample [6.0, 2.2, 4.2, 1.0] is : Iris-versicolor

Number of votes : 2 out of 2

The predicted class for sample [6.2, 2.3, 4.5, 1.5] is : Iris-versicolor

Number of votes : 2 out of 2

The predicted class for sample [5.0, 2.1, 3.6, 1.2] is : Iris-versicolor

Number of votes : 2 out of 2

The predicted class for sample [6.6, 2.8, 5.4, 2.0] is : Iris-virginica

Number of votes : 2 out of 2

The predicted class for sample [6.4, 3.2, 5.3, 2.3] is : Iris-virginica


Number of votes : 2 out of 2

The predicted class for sample [7.0, 3.1, 5.5, 1.8] is : Iris-virginica

Number of votes : 2 out of 2

The predicted class for sample [6.2, 3.3, 5.9, 2.1] is : Iris-virginica

Number of votes : 2 out of 2

The predicted class for sample [6.6, 2.9, 5.3, 2.3] is : Iris-virginica

Number of votes : 2 out of 2
```

EXPERIMENT 11: Write a program to implement Naïve Bayes Classifier Algorithm.

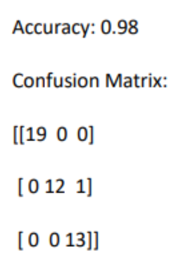
THEORY

It is a classification technique based on Bayes' Theorem with an independence assumption among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature

MAIN.PY

```python
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
from sklearn.naive_bayes import GaussianNB
from sklearn import metrics

iris = load_iris()
X = iris.data
y = iris.target
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)
nb_classifier = GaussianNB()
# Train the classifier using the training data
nb_classifier.fit(X_train, y_train)
y_pred = nb_classifier.predict(X_test)
accuracy = metrics.accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
conf_matrix = metrics.confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(conf_matrix)
```

OUTPUT



```
Accuracy: 0.98

Confusion Matrix:

[[19  0  0]

 [ 0 12  1]

 [ 0  0 13]]
```

EXPERIMENT 12: Write a program to implement Decision Tree Classification Algorithm.

THEORY

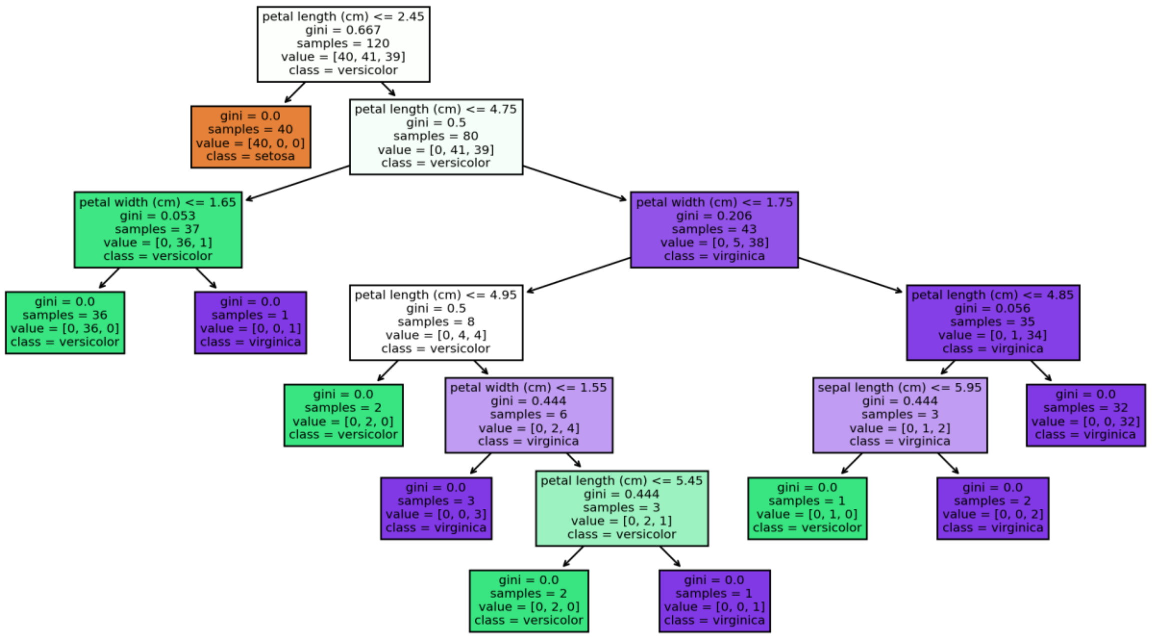
The decision tree classifier creates the classification model by building a decision tree. Each node in the tree specifies a test on an attribute, each branch descending from that node corresponds to one of the possible values for that attribute.

MAIN.PY

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn import metrics
import matplotlib.pyplot as plt
from sklearn.tree import plot_tree

# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target
# 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
)
# Create a Decision Tree classifier
decision_tree = DecisionTreeClassifier()
# Train the classifier
decision_tree.fit(X_train, y_train)
# Make predictions on the test set
y_pred = decision_tree.predict(X_test)
# Calculate and print the accuracy of the model
accuracy = metrics.accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
# Plot the decision tree
plt.figure(figsize=(12, 8))
plot_tree(
    decision_tree,
    feature_names=iris.feature_names,
    class_names=iris.target_names,
    filled=True,
)
plt.show()
```

OUTPUT

```
Accuracy: 1.00
```


petal length (cm) <= 2.45 gini = 0.667 samples = 120 value = [40, 41, 39] class = versicolor
gini = 0.0 samples = 40 value = [40, 0, 0] class = setosa
petal length (cm) <= 4.75 gini = 0.5 samples = 80 value = [0, 41, 39] class = versicolor
petal width (cm) <= 1.65 gini = 0.053 samples = 37 value = [0, 36, 1] class = versicolor
petal width (cm) <= 1.75 gini = 0.206 samples = 43 value = [0, 5, 38] class = virginica
gini = 0.0 samples = 36 value = [0, 36, 0] class = versicolor
gini = 0.0 samples = 1 value = [0, 0, 1] class = virginica
petal length (cm) <= 4.95 gini = 0.5 samples = 8 value = [0, 4, 4] class = versicolor
petal length (cm) <= 4.85 gini = 0.056 samples = 35 value = [0, 1, 34] class = virginica
gini = 0.0 samples = 2 value = [0, 2, 0] class = versicolor
petal width (cm) <= 1.55 gini = 0.444 samples = 6 value = [0, 2, 4] class = virginica
sepal length (cm) <= 5.95 gini = 0.444 samples = 3 value = [0, 1, 2] class = virginica
gini = 0.0 samples = 32 value = [0, 0, 32] class = virginica
gini = 0.0 samples = 3 value = [0, 0, 3] class = virginica
petal length (cm) <= 5.45 gini = 0.444 samples = 3 value = [0, 2, 1] class = versicolor
gini = 0.0 samples = 1 value = [0, 1, 0] class = versicolor
gini = 0.0 samples = 2 value = [0, 0, 2] class = virginica
gini = 0.0 samples = 2 value = [0, 2, 0] class = versicolor
gini = 0.0 samples = 1 value = [0, 0, 1] class = virginica

EXPERIMENT 13: Write a program to implement K - Mean Clustering.

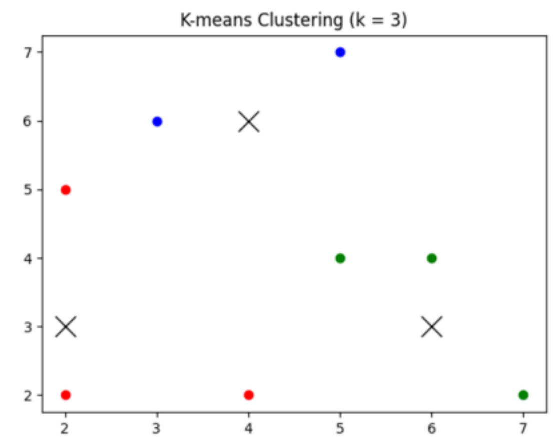
THEORY

k-means clustering is a method of vector quantization, originally from signal processing, that aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster. This results in a partitioning of the data space into Voronoi cells. k-means clustering minimizes within-cluster variances (squared Euclidean distances), but not regular Euclidean distances, which would be the more difficult Weber problem: the mean optimizes squared errors, whereas only the geometric median minimizes Euclidean distances. For instance, better Euclidean solutions can be found using k-medians and k-medoids.

MAIN.PY

```
import numpy as np
import matplotlib.pyplot as plt
data = np.array([[2, 5], [4, 2], [5, 4], [2, 2], [3, 6], [5, 7], [7, 2], [6,
4]])
k = 3
centroids = np.array([[3, 3], [5, 5], [7, 7]])
def euclidean_distance(x1, x2):
 return np.sqrt(np.sum((x1 - x2)**2))
def assign_clusters(data, centroids):
 clusters = []
 for point in data:
 min_distance = float('inf')
 cluster_index = -1
 for i, centroid in enumerate(centroids):
 distance = euclidean_distance(point, centroid)
 if distance < min_distance:
 min_distance = distance
 cluster_index = i
 clusters.append(cluster_index)
 return clusters
def update_centroids(data, clusters, centroids):
 for i in range(k):
 cluster_data = data[np.array(clusters) == i]
 if len(cluster_data) > 0:
 centroids[i] = np.mean(cluster_data, axis=0)
 return centroids
def kmeans(data, k, centroids):
 clusters = assign_clusters(data, centroids)
 while True:
 new_centroids = update_centroids(data, clusters, centroids)
 new_clusters = assign_clusters(data, new_centroids)
 if np.array_equal(clusters, new_clusters):
 break
 centroids = new_centroids
 clusters = new_clusters
 return clusters, centroids
clusters, centroids = kmeans(data, k, centroids)
colors = ['r', 'g', 'b']
for i, cluster in enumerate(clusters):
 plt.plot(data[i, 0], data[i, 1], marker='o', color=colors[cluster])
for i, centroid in enumerate(centroids):
 plt.plot(centroid[0], centroid[1], marker='x', color='k', markersize=15)
plt.title('K-means Clustering (k = 3)')
plt.show()
```

OUTPUT



EXPERIMENT 14: Write a program to reduce the number of dimensions using principal component analysis (PCA).

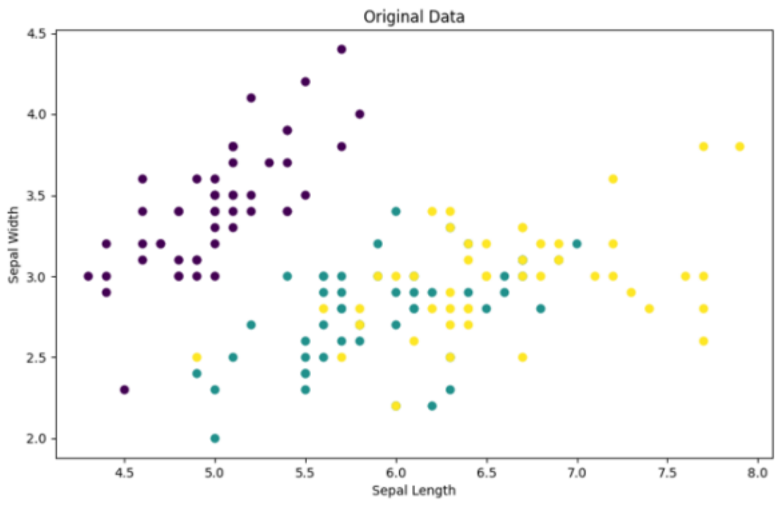
THEORY

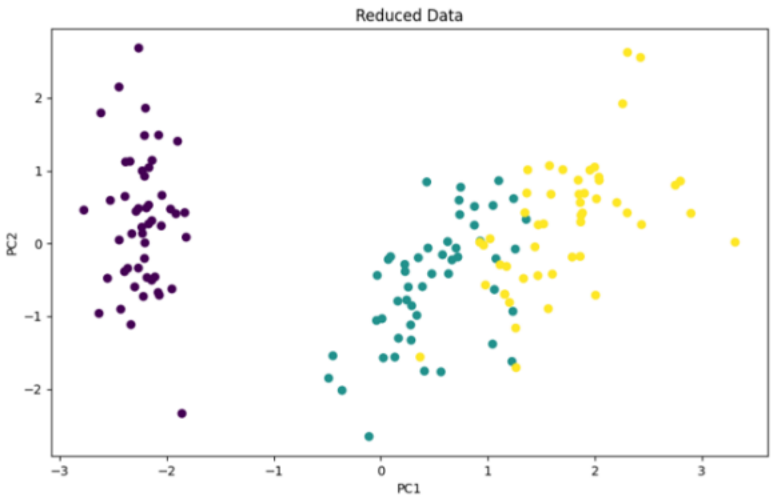
Principal component analysis, or PCA, is a dimensionality reduction method that is often used to reduce the dimensionality of large data sets, by transforming a large set of variables into a smaller one that still contains most of the information in the large set.

MAIN.PY

```
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris

iris = load_iris()
data = pd.DataFrame(
    data=np.c_[iris["data"], iris["target"]],
    columns=iris["feature_names"] + ["species"],
)
data.to_csv("iris.csv", index=False)
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

data = pd.read_csv("iris.csv")
features = ["sepal_length", "sepal_width", "petal_length", "petal_width"]
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
standardized_data = scaler.fit_transform(data[features])
pca = PCA(n_components=2)
pca.fit(standardized_data)
reduced_data = pca.transform(standardized_data)
print(reduced_data)
plt.figure(figsize=(10, 6))
plt.scatter(data["sepal_length"], data["sepal_width"], c=data["species"])
plt.title("Original Data")
plt.xlabel("Sepal Length")
plt.ylabel("Sepal Width")
plt.show()
plt.figure(figsize=(10, 6))
plt.scatter(reduced_data[:, 0], reduced_data[:, 1], c=data["species"])
plt.title("Reduced Data")
plt.xlabel("PC1")
plt.ylabel("PC2")
plt.show()
```

OUTPUT





EXPERIMENT 15: Write a program to implement a simple feedforward neural network for a basic task like binary classification.

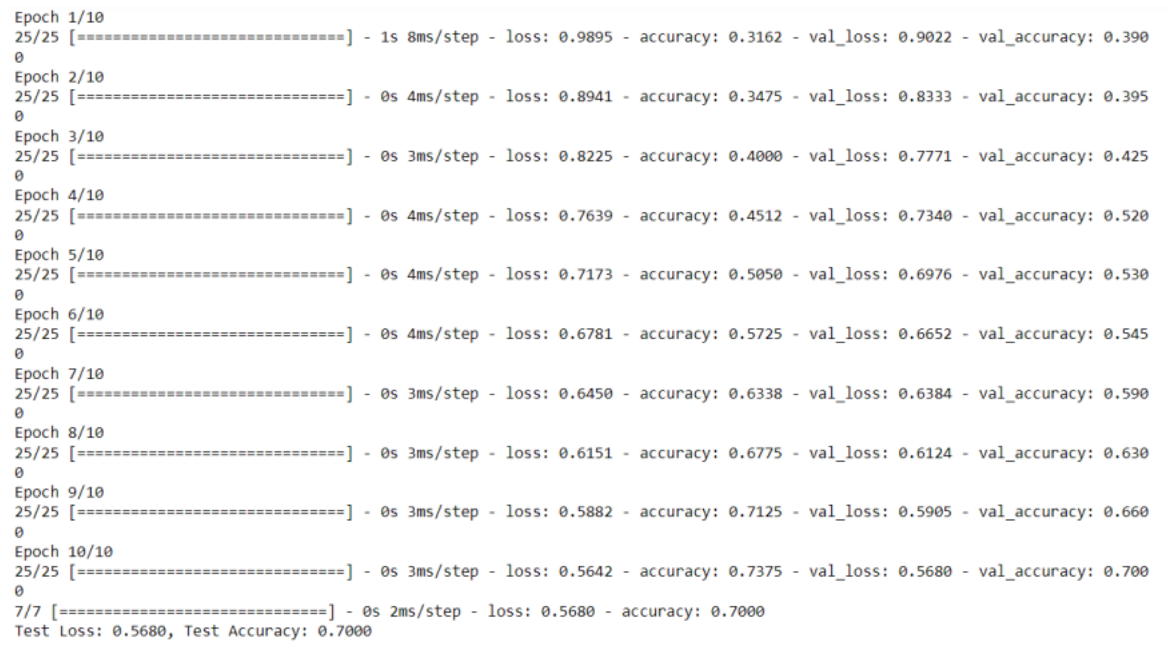
THEORY

A simple feedforward neural network for binary classification consists of input, hidden, and output layers. The input layer receives features, each node representing a feature. Hidden layers process these features through weighted connections, applying activation functions to introduce non-linearity. The output layer, often using a sigmoid activation, produces a probability indicating class membership. During training, the network adjusts weights using backpropagation and gradient descent to minimize a loss function. The network learns to differentiate between classes, making it capable of binary classification tasks.

MAIN.PY

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification

X, y = make_classification(n_samples=1000, n_features=20, n_classes=2,
random_state=42)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
model = Sequential()
model.add(Dense(10, input_dim=20, activation="relu"))
model.add(Dense(1, activation="sigmoid"))
model.compile(
    loss="binary_crossentropy", optimizer=Adam(lr=0.001), metrics=["accuracy"]
)
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test,
y_test))
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Loss: {loss:.4f}, Test Accuracy: {accuracy:.4f}")
```

OUTPUT



```
Epoch 1/10
25/25 [==============================] - 1s 8ms/step - loss: 0.9895 - accuracy: 0.3162 - val_loss: 0.9022 - val_accuracy: 0.390
0
Epoch 2/10
25/25 [==============================] - 0s 4ms/step - loss: 0.8941 - accuracy: 0.3475 - val_loss: 0.8333 - val_accuracy: 0.395
0
Epoch 3/10
25/25 [==============================] - 0s 3ms/step - loss: 0.8225 - accuracy: 0.4000 - val_loss: 0.7771 - val_accuracy: 0.425
0
Epoch 4/10
25/25 [==============================] - 0s 4ms/step - loss: 0.7639 - accuracy: 0.4512 - val_loss: 0.7340 - val_accuracy: 0.520
0
Epoch 5/10
25/25 [==============================] - 0s 4ms/step - loss: 0.7173 - accuracy: 0.5050 - val_loss: 0.6976 - val_accuracy: 0.530
0
Epoch 6/10
25/25 [==============================] - 0s 4ms/step - loss: 0.6781 - accuracy: 0.5725 - val_loss: 0.6652 - val_accuracy: 0.545
0
Epoch 7/10
25/25 [==============================] - 0s 3ms/step - loss: 0.6450 - accuracy: 0.6338 - val_loss: 0.6384 - val_accuracy: 0.590
0
Epoch 8/10
25/25 [==============================] - 0s 3ms/step - loss: 0.6151 - accuracy: 0.6775 - val_loss: 0.6124 - val_accuracy: 0.630
0
Epoch 9/10
25/25 [==============================] - 0s 3ms/step - loss: 0.5882 - accuracy: 0.7125 - val_loss: 0.5905 - val_accuracy: 0.660
0
Epoch 10/10
25/25 [==============================] - 0s 3ms/step - loss: 0.5642 - accuracy: 0.7375 - val_loss: 0.5680 - val_accuracy: 0.700
0
7/7 [==============================] - 0s 2ms/step - loss: 0.5680 - accuracy: 0.7000
Test Loss: 0.5680, Test Accuracy: 0.7000
```

EXPERIMENT 16: Write a program to evaluate the performance of any classification model using metrics like accuracy, precision, recall, F1-score, etc.

THEORY

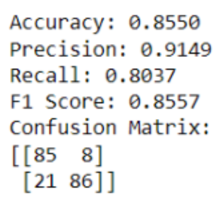
The performance of a classification model is assessed using various metrics. **Accuracy** measures the overall correctness of predictions. **Precision** gauges the accuracy of positive predictions, emphasizing their reliability. **Recall** assesses the model's ability to capture all positive instances. **F1-score** combines precision and recall, offering a balanced metric. **Confusion matrix** provides a detailed view of true positives, true negatives, false positives, and false negatives. **ROC-AUC** evaluates the model's ability to distinguish between classes across different thresholds. These metrics collectively offer a comprehensive evaluation of a classification model's effectiveness in correctly identifying and discriminating between classes.

MAIN.PY

```
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    confusion_matrix,
)
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification
from sklearn.linear_model import LogisticRegression

X, y = make_classification(n_samples=1000, n_features=20, n_classes=2,
random_state=42)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
model = LogisticRegression()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
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)
conf_matrix = confusion_matrix(y_test, y_pred)
print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
```

```python
print(f"F1 Score: {f1:.4f}")
print("Confusion Matrix:")
print(conf_matrix)
```

OUTPUT



```
Accuracy: 0.8550
Precision: 0.9149
Recall: 0.8037
F1 Score: 0.8557
Confusion Matrix:
[[85  8]
 [21 86]]
```

EXPERIMENT 17: Write a program to construct a simple Bayesian network.

THEORY

A Bayesian network is a directed acyclic graph in which each edge corresponds to a conditional dependency, and each node corresponds to a unique random variable. Bayesian network consists of two major parts: a directed acyclic graph and a set of conditional probability distributions

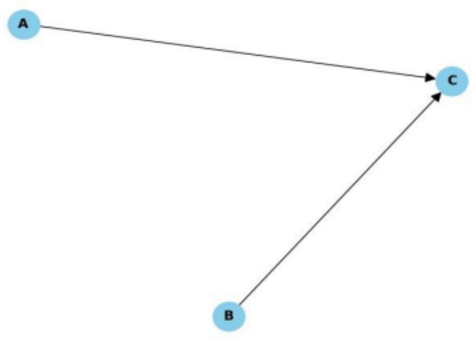
• The directed acyclic graph is a set of random variables represented by nodes.

• The conditional probability distribution of a node (random variable) is defined for every possible outcome of the preceding causal node(s).

MAIN.PY

```python
import numpy as np
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt

# Generate a simple Bayesian network structure
structure = [("A", "C"), ("B", "C")]
# Create a directed graph using NetworkX
bayesian_network = nx.DiGraph(structure)
# Plot the Bayesian network
pos = nx.spring_layout(bayesian_network)
nx.draw(
    bayesian_network,
    pos,
    with_labels=True,
    font_weight="bold",
    node_color="skyblue",
    node_size=800,
    arrowsize=20,
)
plt.title("Simple Bayesian Network")
plt.show()
```

OUTPUT



EXPERIMENT 18: Write a program to implement a deep learning model for image classification.

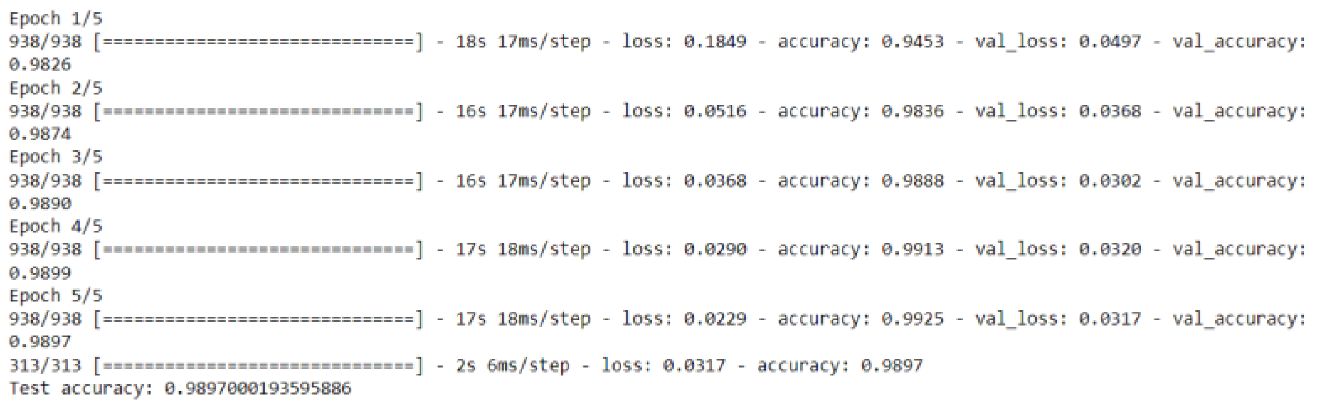
THEORY

A deep learning model for image classification involves a convolutional neural network (CNN). The CNN extracts hierarchical features from the input image through convolutional layers, utilizing filters to detect patterns. Pooling layers reduce spatial dimensions while retaining essential information. Fully connected layers integrate features for classification, and activation functions introduce non-linearity. During training, the model adjusts weights using backpropagation and optimization algorithms to minimize a defined loss function. Transfer learning, using pre-trained models, enhances performance on smaller datasets. The model learns hierarchical representations, enabling accurate classification of diverse image categories.

MAIN.PY

```python
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import (
    mnist,
)  # Example dataset, you can replace it with your dataset

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28, 28, 1)).astype("float32") / 255
test_images = test_images.reshape((10000, 28, 28, 1)).astype("float32") / 255
train_labels = tf.keras.utils.to_categorical(train_labels)
test_labels = tf.keras.utils.to_categorical(test_labels)
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation="relu", input_shape=(28, 28,
1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation="relu"))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation="relu"))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation="relu"))
model.add(layers.Dense(10, activation="softmax"))
model.compile(optimizer="adam", loss="categorical_crossentropy",
metrics=["accuracy"])
model.fit(
    train_images,
    train_labels,
    epochs=5,
    batch_size=64,
    validation_data=(test_images, test_labels),
)
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"Test accuracy: {test_acc}")
```

OUTPUT



```
Epoch 1/5
938/938 [==============================] - 18s 17ms/step - loss: 0.1849 - accuracy: 0.9453 - val_loss: 0.0497 - val_accuracy:
0.9826
Epoch 2/5
938/938 [==============================] - 16s 17ms/step - loss: 0.0516 - accuracy: 0.9836 - val_loss: 0.0368 - val_accuracy:
0.9874
Epoch 3/5
938/938 [==============================] - 16s 17ms/step - loss: 0.0368 - accuracy: 0.9888 - val_loss: 0.0302 - val_accuracy:
0.9890
Epoch 4/5
938/938 [==============================] - 17s 18ms/step - loss: 0.0290 - accuracy: 0.9913 - val_loss: 0.0320 - val_accuracy:
0.9899
Epoch 5/5
938/938 [==============================] - 17s 18ms/step - loss: 0.0229 - accuracy: 0.9925 - val_loss: 0.0317 - val_accuracy:
0.9897
313/313 [==============================] - 2s 6ms/step - loss: 0.0317 - accuracy: 0.9897
Test accuracy: 0.9897000193595886
```