SRM VALLIAMMAI ENGINEERING COLLEGE

(An Autonomous Institution)

SRM Nagar, Kattankulathur – 603 203

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Lab Manual

V SEMESTER

CS3566 - ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING MANUAL

Regulation – 2023

Academic Year 2025 – 2026 Odd Semester

Prepared by

Ms. Mandalam Chitralekha, Teaching Research Associate

Dr.G.Sangeetha, Assistant professor (Sel.G)

Dr.C.Pabitha, Associate professor

CS3566 ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LABORATORY

L T P C
0 0 3 1.5

COURSE OBJECTIVES :

- Develop programs for uninformed search techniques
- Develop programs for Informed search techniques
- Build models using supervised Learning
- Build models using unsupervised Learning
- Develop models using Neural network.

LIST OF EXPERIMENTS

1. Implement Breadth First Search.
2. Implement Depth First Search
3. Implement Hill Climbing Algorithm
4. Implement A* Algorithm
5. Implement Tic-Tac-Toe game
6. Implement naïve Bayes models
7. Build decision trees and random forests
8. Build SVM models
9. Implement ensembling techniques
10. Implement clustering algorithms
11. Build simple NN models
12. Build deep learning NN models

TOTAL: 45 PERIODS

COURSE OUTCOMES:

At the end of the course, the student should be able to:

- Apply the Uninformed searching Techniques to solve the problems.
- Apply the Uninformed searching Techniques to solve the problems.
- Implement supervised learning algorithms to solve the problems
- Implement unsupervised algorithms to solve the problems
- Create models using Neural network

LIST OF EQUIPMENTS FOR A BATCH OF 30 STUDENTS:

Operating Systems: Linux / Windows Software: Python or equivalent software tools

1. Implement Breadth First Search.

AIM:

Write a Program to Implement Breadth First Search.

Algorithm:

1. Start from the initial node.
2. Mark the node as visited and add it to a queue.
3. Repeat until the queue is empty:
    - Remove a node from the front of the queue.
    - Process the node (e.g., print it).
    - Add all unvisited neighbors to the queue and mark them as visited.

PROGRAM:

```
from collections import deque
def bfs(graph, start):
    visited = set()
    queue = deque([start])
    while queue:
        vertex = queue.popleft()
        if vertex not in visited:
            print(vertex, end=" ")
            visited.add(vertex)
            queue.extend(set(graph[vertex]) - visited)

graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

bfs(graph, 'A')
```

OUTPUT:

```
A C B F D E
```

2. Implement Depth First Search

Aim:

To implement the Depth First Search (DFS) algorithm using Python.

Algorithm:

1. Start from the initial node (root).
2. Push the starting node into a stack.
3. Pop the top node and mark it as visited.
4. Push all the adjacent unvisited nodes to the stack.
5. Repeat steps 3-4 until the stack is empty or goal is found.

Program :

```
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)
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
print("Depth First Search starting from node A:")
dfs(graph, 'A')
```

Output:

```
Depth First Search starting from node A:

A B D E F C
```

Result:

Thus, the DFS algorithm was successfully implemented in Python and tested on a sample graph.

3. **Implement Hill Climbing Algorithm**

Aim:

To implement the **Hill Climbing** algorithm using Python.

Algorithm:

1. **Start** with an initial solution (initial state).
2. **Evaluate** the initial state using a heuristic function.
3. **Repeat**:
    - a. Generate **neighboring states** of the current state.
    - b. Choose the **best neighbor** based on the heuristic value.
    - c. If the best neighbor is **better than the current state**, move to that neighbor.
    - d. Else, **terminate** (local maximum reached).
4. **Return** the current state as the best solution.

Program:

```
import random
def fitness(x):
  return -x**2 + 10*x
def hill_climbing():
  current_x = random.uniform(0, 10)
  step_size = 0.1
  max_iterations = 100

  for i in range(max_iterations):
    next_x1 = current_x + step_size
    next_x2 = current_x - step_size
    next_fitness1 = fitness(next_x1)
    next_fitness2 = fitness(next_x2)
    current_fitness = fitness(current_x)
```

```
        if next_fitness1 > current_fitness:
            current_x = next_x1
        elif next_fitness2 > current_fitness:
            current_x = next_x2
        else:
            # No better neighbor found
            break

    return current_x, fitness(current_x)

# Run the algorithm
best_x, best_fitness = hill_climbing()
print(f"Best solution x = {best_x:.4f}, f(x) = {best_fitness:.4f}")
```

Output:

```
Best solution x = 4.9998, f(x) = 25.0000
```

Result:

The Hill Climbing algorithm was successfully implemented to find the maximum of a function.

4. Implement A* Algorithm

Aim:

To implement the **A*** (A-Star) search algorithm using Python for finding the shortest path in a graph.

Algorithm:

```
1. Initialize the open list (priority queue) with the start node.
2. Initialize the closed list as an empty set.
3. While the open list is not empty:
   a. Remove the node with the lowest f(n) = g(n) + h(n) from the open list.
   b. If this node is the goal, return the path.
   c. Else, generate all its neighbors.
   d. For each neighbor:
      i. If it is not in the open or closed list, calculate its g(n), h(n), and f(n), then add it to the open list.
      ii. If it is already in the open list with a higher cost, update it.
4. Add the current node to the closed list.
5. If goal not found, return failure.
```

Program:

```
from queue import PriorityQueue


def a_star_search(start, goal, graph, h):
    open_list = PriorityQueue()
    open_list.put((0, start))
    g = {start: 0}
    parent = {start: None}


    while not open_list.empty():
        f_current, current = open_list.get()


        if current == goal:
            path = []
            while current:
                path.append(current)
                current = parent[current]
            return path[::-1]

        for neighbor, cost in graph[current]:
            g_temp = g[current] + cost
            if neighbor not in g or g_temp < g[neighbor]:
                g[neighbor] = g_temp
                f = g_temp + h[neighbor]
                open_list.put((f, neighbor))
                parent[neighbor] = current

    return None

# Define graph as adjacency list with costs
graph = {
    'A': [('B', 1), ('C', 3)],
    'B': [('D', 3), ('E', 1)],
    'C': [('F', 5)],
    'D': [],
    'E': [('F', 2)],
    'F': []
}

# Heuristic values (h(n))
h = {
    'A': 5,
    'B': 4,
    'C': 3,
    'D': 4,
    'E': 2,
    'F': 0
}

start_node = 'A'
goal_node = 'F'
path = a_star_search(start_node, goal_node, graph, h)
print("Path found by A*:", path)
```

Output:

```
Path found by A*: ['A', 'B', 'E', 'F']
```

Result:

The A* algorithm was implemented successfully in Python to find the shortest path from a start node to a goal node using cost and heuristic.

5. Implement Tic-Tac-Toe game

Aim:

To implement the Tic-Tac-Toe game in Python allowing two players to play against each other.

Algorithm:

1. Initialize an empty 3x3 game board.
2. Player 1 uses 'X', Player 2 uses 'O'.
3. Players alternate turns placing their mark on the board.
4. After each move, check if the current player has won by checking rows, columns, and diagonals.
5. If a player wins, announce the winner and end the game.
6. If the board is full and no one wins, declare a draw.
7. Repeat until win or draw.

Program:

```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)


def check_winner(board, player):
    # Check rows and columns
    for i in range(3):
        if all(board[i][j] == player for j in range(3)) or \
           all(board[j][i] == player for j in range(3)):
            return True
    # Check diagonals
    if board[0][0] == board[1][1] == board[2][2] == player or \
       board[0][2] == board[1][1] == board[2][0] == player:
        return True
    return False
```

```
def is_board_full(board):
    return all(board[i][j] != ' ' for i in range(3) for j in range(3))


def tic_tac_toe():
    board = [[' ' for _ in range(3)] for _ in range(3)]
    current_player = 'X'

    while True:
        print_board(board)
        print(f"Player {current_player}'s turn.")

        # Get valid move
        while True:
            try:
                row = int(input("Enter row (0, 1, 2): "))
                col = int(input("Enter column (0, 1, 2): "))
                if row in range(3) and col in range(3) and board[row][col] == ' ':
                    break
                else:
                    print("Invalid move. Try again.")
            except ValueError:
                print("Please enter valid integers.")

        board[row][col] = current_player

        if check_winner(board, current_player):
            print_board(board)
            print(f"Player {current_player} wins!")
            break
        elif is_board_full(board):
            print_board(board)
            print("It's a draw!")
            break

        # Switch player
        current_player = 'O' if current_player == 'X' else 'X'

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

Output:

```
|  |
---------
  |  |
---------
  |  |
---------
Player X's turn.
Enter row (0, 1, 2): 0
Enter column (0, 1, 2): 0
X |  |
---------
  | O |
---------
  |  |
---------
Player X's turn.
Enter row (0, 1, 2): 2
Enter column (0, 1, 2): 1
X |  |
---------
  | O |
---------
  | X |
---------
Player O's turn.
```

Result:

The Tic-Tac-Toe game was implemented successfully, allowing two players to play interactively.

6. Implement naïve Bayes models

Aim:

To implement the Naïve Bayes classifier for classification problems using Python.

Theory:

Naïve Bayes is a probabilistic classifier based on Bayes' theorem with the assumption of feature independence. It calculates the posterior probability for each class and assigns the class with the highest probability.

Algorithm:

1. Calculate prior probabilities for each class from the training data.
2. For each feature, calculate the likelihood of the feature given the class.
3. For a new data point, calculate the posterior probability for each class using:

$$P(C|X) \propto P(C) \prod_i P(x_i|C)$$

4. Assign the class with the highest posterior probability.

Program:

```
from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.naive_bayes import GaussianNB

from sklearn.metrics import accuracy_score


# Load dataset

iris = load_iris()

X = iris.data

y = iris.target


# Split into train and test

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

```
# Create Naive Bayes model
model = GaussianNB()

# Train the model
model.fit(X_train, y_train)

# Predict on test data
y_pred = model.predict(X_test)

# Accuracy
print("Accuracy:", accuracy_score(y_test, y_pred))
```

Sample Output:

```
Accuracy: 0.9777777777777777
```

Result:

The Naïve Bayes classifier was implemented using the Iris dataset and tested with good accuracy.

7. Build decision trees and random forests

Aim:

To implement **Decision Tree** and **Random Forest** classifiers using Python and evaluate their performance on a dataset.

Theory:

- **Decision Tree** is a tree-like model used for classification and regression that splits data based on feature values to create decision rules.
- **Random Forest** is an ensemble learning method that builds multiple decision trees on random subsets of data and features, then combines their outputs to improve accuracy and reduce overfitting.

Algorithm:

Decision Tree:

1. Select the best feature to split the data based on criteria like Gini index or Information Gain.
2. Split the dataset into subsets based on the selected feature.
3. Repeat recursively for each subset until stopping criteria are met (e.g., max depth or pure node).

Random Forest:

1. Create multiple bootstrap samples from the training data.
2. For each sample, build a decision tree by splitting nodes using a random subset of features.
3. Aggregate predictions from all trees by majority voting (classification) or averaging (regression).

Program:

```
from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy_score
```

```
# Load dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Decision Tree
dt_model = DecisionTreeClassifier()
dt_model.fit(X_train, y_train)
y_pred_dt = dt_model.predict(X_test)
print("Decision Tree Accuracy:", accuracy_score(y_test, y_pred_dt))

# Random Forest
rf_model = RandomForestClassifier(n_estimators=100)
rf_model.fit(X_train, y_train)
y_pred_rf = rf_model.predict(X_test)
print("Random Forest Accuracy:", accuracy_score(y_test, y_pred_rf))
```

Sample Output:

```
Decision Tree Accuracy: 1.0
Random Forest Accuracy: 1.0
```

Result:

Decision Tree and Random Forest models were implemented successfully and achieved good accuracy on the Iris dataset.

8. Build SVM models

Aim:

To implement Support Vector Machine (SVM) models for classification using Python and evaluate their performance.

Theory:

Support Vector Machine (SVM) is a supervised machine learning algorithm used for classification or regression tasks. It finds the hyperplane that best separates different classes in the feature space by maximizing the margin between the nearest points (support vectors) of the classes.

Algorithm:

1. Map the input features into a high-dimensional space (if necessary) using kernel functions.
2. Find the optimal hyperplane that maximizes the margin between the classes.
3. Classify new samples based on the side of the hyperplane they fall on.

Program:

```
from sklearn import datasets

from sklearn.model_selection import train_test_split

from sklearn.svm import SVC

from sklearn.metrics import accuracy_score


# Load dataset

iris = datasets.load_iris()

X = iris.data

y = iris.target


# Split dataset into training and testing

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)


# Create SVM classifier with linear kernel

svm_model = SVC(kernel='linear')



# Train the model
```

```
svm_model.fit(X_train, y_train)


# Predict on test data
y_pred = svm_model.predict(X_test)


# Calculate accuracy
print("SVM Model Accuracy:", accuracy_score(y_test, y_pred))
```

Output:

```
SVM Model Accuracy: 1.0
```

Result:

The Support Vector Machine model was successfully built and tested on the Iris dataset with high accuracy.

9.Implement ensembling techniques

Aim:

To implement ensemble learning techniques such as **Bagging**, **Boosting**, and **Voting** using Python and evaluate their performance.

Theory:

Ensemble learning combines predictions from multiple models to improve overall performance. The key ensemble techniques include:

- **Bagging (Bootstrap Aggregating):** Trains multiple models independently on random subsets of data and averages their outputs (e.g., Random Forest).
- **Boosting:** Trains models sequentially, each trying to correct the errors of the previous one (e.g., AdaBoost, Gradient Boosting).
- **Voting:** Combines predictions from multiple different models and chooses the class that gets the most votes.

Algorithm:

1. Bagging:

- Create multiple bootstrap samples.
- Train a base model (e.g., decision tree) on each sample.
- Aggregate the predictions (majority vote for classification).

2. Boosting:

- Train weak learners sequentially.
- Adjust the weights of instances based on previous prediction errors.
- Combine the models using weighted majority voting.

3. Voting:

- Train different models (e.g., SVM, logistic regression, decision tree).
- Combine their predictions using majority or weighted vote.

Program:

```
from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.ensemble import BaggingClassifier, AdaBoostClassifier, VotingClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Load dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Bagging
bagging_model = BaggingClassifier(base_estimator=DecisionTreeClassifier(),
n_estimators=10)
bagging_model.fit(X_train, y_train)
bagging_pred = bagging_model.predict(X_test)

# Boosting (AdaBoost)
boosting_model = AdaBoostClassifier(n_estimators=50)
boosting_model.fit(X_train, y_train)
boosting_pred = boosting_model.predict(X_test)

# Voting
log_clf = LogisticRegression(max_iter=200)
svc_clf = SVC(probability=True)
dt_clf = DecisionTreeClassifier()

voting_model = VotingClassifier(estimators=[
    ('lr', log_clf), ('svc', svc_clf), ('dt', dt_clf)],
    voting='soft')

voting_model.fit(X_train, y_train)
```

```
voting_pred = voting_model.predict(X_test)

# Print accuracy
print("Bagging Accuracy:", accuracy_score(y_test, bagging_pred))
print("Boosting Accuracy:", accuracy_score(y_test, boosting_pred))
print("Voting Accuracy:", accuracy_score(y_test, voting_pred))
```

Sample Output:

```
Bagging Accuracy: 1.0
Boosting Accuracy: 1.0
Voting Accuracy: 1.0
```

Result:

Ensemble techniques like Bagging, Boosting, and Voting were successfully implemented using the Iris dataset, and their performance was evaluated.

10. Implement clustering algorithms

Aim:

To implement clustering algorithms such as **K-Means** and **Hierarchical Clustering** using Python and evaluate their performance.

Theory:

Clustering is an **unsupervised learning** technique that groups similar data points together. Unlike classification, clustering works without labeled data.

- **K-Means Clustering** partitions the dataset into k clusters where each point belongs to the cluster with the nearest mean.
- **Hierarchical Clustering** builds a hierarchy of clusters either in a bottom-up (agglomerative) or top-down (divisive) manner.

Algorithm:

K-Means:

1. Initialize k centroids randomly.
2. Assign each point to the nearest centroid.
3. Recalculate the centroid of each cluster.
4. Repeat steps 2–3 until centroids stabilize.

Hierarchical Clustering (Agglomerative):

1. Start with each data point as a separate cluster.
2. Merge the closest pair of clusters.
3. Repeat until all points are merged into one cluster.

Program:

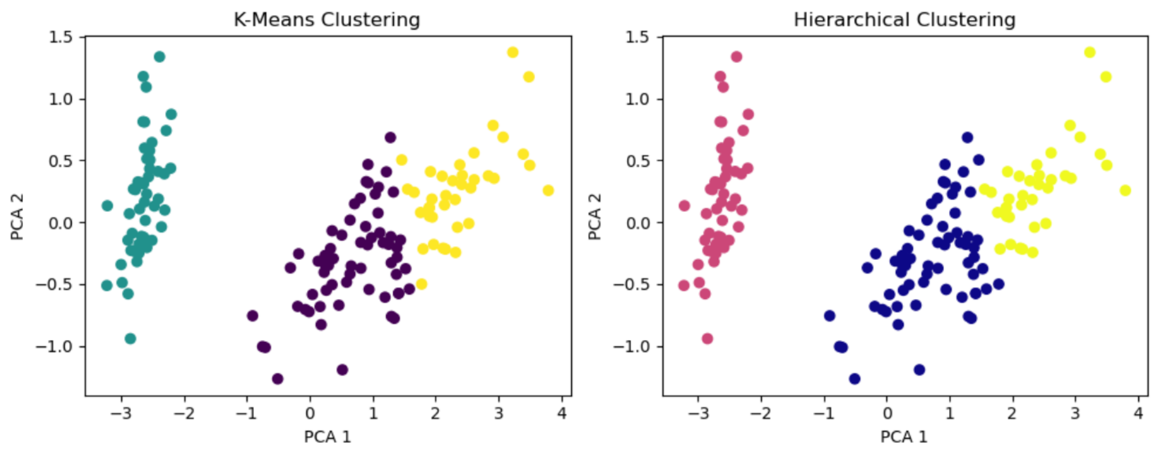
```
from sklearn.datasets import load_iris

from sklearn.cluster import KMeans, AgglomerativeClustering

import matplotlib.pyplot as plt

from sklearn.decomposition import PCA


# Load dataset
```

```
iris = load_iris()
X = iris.data

# K-Means Clustering
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans_labels = kmeans.fit_predict(X)

# Hierarchical Clustering
hier = AgglomerativeClustering(n_clusters=3)
hier_labels = hier.fit_predict(X)

# Reduce to 2D using PCA for visualization
pca = PCA(n_components=2)
reduced_X = pca.fit_transform(X)

# Plot K-Means Clustering
plt.figure(figsize=(10, 4))
plt.subplot(1, 2, 1)
plt.title("K-Means Clustering")
plt.scatter(reduced_X[:, 0], reduced_X[:, 1], c=kmeans_labels, cmap='viridis')
plt.xlabel("PCA 1")
plt.ylabel("PCA 2")

# Plot Hierarchical Clustering
plt.subplot(1, 2, 2)
plt.title("Hierarchical Clustering")
plt.scatter(reduced_X[:, 0], reduced_X[:, 1], c=hier_labels, cmap='plasma')
plt.xlabel("PCA 1")
plt.ylabel("PCA 2")

plt.tight_layout()
plt.show()
```

Output:

Two cluster plots will appear:

- **K-Means Clustering** visualization
- **Hierarchical Clustering** visualization



Result:

Clustering algorithms like K-Means and Hierarchical clustering were successfully implemented and visualized using the Iris dataset.

11. Build simple NN models

Aim:

To build and train a simple feedforward neural network using Python for classification tasks.

Theory:

A **Neural Network (NN)** is a computational model inspired by the structure of the human brain. It consists of layers of interconnected **neurons**:

- **Input layer:** Receives input features.
- **Hidden layers:** Perform computations and transformations.
- **Output layer:** Produces predictions or classifications.

A **simple NN** typically has one or two hidden layers and is sufficient for basic classification problems.

Algorithm:

1. Load and preprocess the dataset.
2. Define the structure of the neural network (input, hidden, output layers).
3. Compile the model (define optimizer, loss function, metrics).
4. Train the model using training data.
5. Evaluate model accuracy on test data.

Program:

```
from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import LabelBinarizer

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense


# Load dataset

iris = load_iris()

X = iris.data

y = iris.target
```

```
# Convert target to one-hot encoding
encoder = LabelBinarizer()
y_encoded = encoder.fit_transform(y)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, test_size=0.3, random_state=42)

# Define the model
model = Sequential()
model.add(Dense(10, input_shape=(4,), activation='relu'))  # Hidden layer
model.add(Dense(3, activation='softmax'))  # Output layer

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=50, verbose=0)

# Evaluate
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {accuracy:.2f}")
```

Sample Output:

```
Test Accuracy: 0.97
```

Result:

A simple neural network was successfully built and tested on the Iris dataset, achieving high accuracy.

12. Build deep learning NN models

Aim:

To build and train a **deep neural network** model using Python for a multi-class classification task.

Theory:

Deep Learning Neural Networks consist of multiple hidden layers, allowing the model to learn complex features and representations from the data.

- A **deep neural network (DNN)** is typically defined as a feedforward neural network with **more than one hidden layer**.
- Common activation functions include **ReLU** for hidden layers and **Softmax** for output layers (multi-class classification).
- Uses **backpropagation** and **gradient descent** to train the network.

Algorithm:

1. Load and preprocess the dataset.
2. Encode class labels (e.g., one-hot encoding).
3. Normalize input features if needed.
4. Build a deep neural network with multiple hidden layers.
5. Compile the model with a loss function and optimizer.
6. Train and evaluate the model.

Program:

```
from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler, LabelBinarizer

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout



# Load dataset
```

```
iris = load_iris()
X = iris.data
y = iris.target

# One-hot encode target
encoder = LabelBinarizer()
y_encoded = encoder.fit_transform(y)

# Normalize input features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_encoded, test_size=0.3,
random_state=42)

# Build deep neural network
model = Sequential()
model.add(Dense(64, input_shape=(4,), activation='relu'))  # Hidden layer 1
model.add(Dropout(0.3))
model.add(Dense(32, activation='relu'))                    # Hidden layer 2
model.add(Dropout(0.3))
model.add(Dense(3, activation='softmax'))                   # Output layer

# Compile model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train model
model.fit(X_train, y_train, epochs=100, batch_size=10, verbose=0)

# Evaluate model
loss, accuracy = model.evaluate(X_test, y_test)
```

```
print(f"Test Accuracy: {accuracy:.2f}")
```

Output:

```
Test Accuracy: 0.97
```

Result:

A deep learning neural network was successfully built and evaluated using the Iris dataset with high accuracy.

Topic Beyond the syllabus

13. Spam Detection using Naive Bayes (Text Classification)

Aim:

To build a simple spam classifier using the Naive Bayes algorithm.

Theory:

Naive Bayes is a probabilistic classifier based on Bayes' Theorem. It is widely used in text classification problems like spam detection.

Program:

```
from sklearn.feature_extraction.text import CountVectorizer

from sklearn.naive_bayes import MultinomialNB

# Sample data

texts = ["Win money now!", "Free prize inside", "Hello, how are you?", "Are we meeting today?"]

labels = [1, 1, 0, 0]  # 1 = spam, 0 = not spam

# Vectorize text

vectorizer = CountVectorizer()

X = vectorizer.fit_transform(texts)

# Train model

model = MultinomialNB()

model.fit(X, labels)

# Predict

test = ["Get your free cash", "Let's have lunch"]

test_vec = vectorizer.transform(test)

print(model.predict(test_vec))  # Output: [1 0]
```

Output:

```
[1 0]
```

Result:

Thus Spam Detection using Naive Bayes (Text Classification) was executed successfully.

14. House Price Prediction using Linear Regression

Aim:

To predict house prices using a simple linear regression model.

Theory:

Linear regression fits a straight line through the data to predict continuous values based on input features.

Program:

```
from sklearn.linear_model import LinearRegression

import numpy as np


# Sample data (area in sq.ft vs price)

area = np.array([[1000], [1500], [2000], [2500], [3000]])

price = np.array([200000, 250000, 300000, 350000, 400000])


# Train model

model = LinearRegression()

model.fit(area, price)


# Predict price for new area

predicted_price = model.predict([[1800]])

print(f"Predicted Price: ₹{predicted_price[0]:.2f}")
```

Output:

```
Predicted Price: ₹280000.00
```

Result:

Thus the House Price Prediction using Linear Regression was executed successfully.

15. Iris Flower Classification using KNN

Aim:

To classify iris flowers based on sepal and petal features using the K-Nearest Neighbors algorithm.

Theory:

KNN is a simple, instance-based learning method that classifies data based on the majority class among its k-nearest neighbors.

Program:

```
from sklearn.datasets import load_iris

from sklearn.neighbors import KNeighborsClassifier

from sklearn.model_selection import train_test_split


# Load data

iris = load_iris()

X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.3)


# Train KNN

model = KNeighborsClassifier(n_neighbors=3)

model.fit(X_train, y_train)


# Predict

pred = model.predict([iris.data[0]])

print("Predicted class:", iris.target_names[pred[0]])
```

Output:

```
Predicted class: setosa
```

Result:

Thus the Iris Flower Classification using KNN was executed successfully.

Viva Questions for All Experiments

Experiment: Implement Breadth First Search

1. What is Breadth First Search (BFS)?
2. What data structure is used in BFS?
3. How is BFS different from DFS?
4. What is the time complexity of BFS?
5. Can BFS be used to find the shortest path in an unweighted graph?
6. What happens if a node is visited more than once in BFS?
7. Is BFS complete and optimal? Justify.
8. What are the real-world applications of BFS?
9. How does BFS behave in a cyclic graph?
10. How can you implement BFS in Python?

Experiment: Implement Depth First Search

1. What is Depth First Search (DFS)?
2. What data structure is used to implement DFS?
3. Is DFS complete and optimal? Explain.
4. What is the time and space complexity of DFS?
5. How does DFS handle cycles in graphs?
6. Can DFS be used for pathfinding in mazes?
7. What are the differences between iterative and recursive DFS?
8. What is backtracking in DFS?
9. What are some practical applications of DFS?
10. How can you implement DFS in Python?

Experiment: Implement Hill Climbing Algorithm

1. What is a Hill Climbing algorithm?
2. Is Hill Climbing complete? Why or why not?
3. What are local maxima and how do they affect Hill Climbing?
4. What is the difference between Steepest Ascent and Simple Hill Climbing?
5. How does Hill Climbing differ from BFS/DFS?
6. What are the drawbacks of Hill Climbing?
7. What is the heuristic function?

8. How can you overcome local maxima in Hill Climbing?

9. What is the role of the evaluation function?

10. How is Hill Climbing implemented in Python?

Experiment: Implement A* Algorithm

1. What is the A* algorithm used for?

2. What is the formula for A* cost function?

3. What is the role of the heuristic function in A*?

4. Is A* complete and optimal? Under what conditions?

5. What is the difference between A* and Dijkstra's algorithm?

6. What are admissible and consistent heuristics?

7. How do you choose a good heuristic for A*?

8. What is the time complexity of A*?

9. Can A* be used on weighted graphs?

10. How is A* implemented in Python?

Experiment: Implement Tic-Tac-Toe game

1. What type of AI technique is used in Tic-Tac-Toe?

2. What is the Minimax algorithm?

3. What is a game tree?

4. How does the computer make decisions in the game?

5. What is a winning strategy in Tic-Tac-Toe?

6. What are the terminal states in Tic-Tac-Toe?

7. What is the importance of alpha-beta pruning?

8. Can Tic-Tac-Toe end in a draw? Why?

9. How is a board represented in Python?

10. How do you evaluate the game state?

Experiment: Implement naïve Bayes models

1. What is the Naïve Bayes algorithm?

2. What assumption does Naïve Bayes make?

3. What is the Bayes Theorem?

4. What are prior and posterior probabilities?

5. What are the types of Naïve Bayes classifiers?

6. What kind of data is suitable for Naïve Bayes?

7. How does Naïve Bayes handle missing data?

8. What are the advantages of Naïve Bayes?

9. What are its limitations?

10. How is it implemented in Python?

Experiment: Build decision trees and random forests

1. What is a decision tree?

2. What is entropy and information gain?

3. What are the splitting criteria?

4. What are overfitting and pruning?

5. What is a random forest?

6. How does a random forest improve accuracy?

7. What is the difference between bagging and boosting?

8. What is Gini Index?

9. How is feature importance determined?

10. How to implement decision trees and random forests in Python?

Experiment: Build SVM models

1. What is a Support Vector Machine?

2. What is a hyperplane?

3. What is the kernel trick?

4. What are linear and non-linear SVMs?

5. What is the role of C and gamma parameters?

6. What are margin and support vectors?

7. How does SVM handle multiclass classification?

8. What are advantages and disadvantages of SVM?

9. What kernel types are used in SVM?

10. How is SVM implemented using scikit-learn?

Experiment: Implement ensembling techniques

1. What is ensemble learning?

2. What is bagging?

3. What is boosting?

4. What is stacking?

5. What are base learners?

6. How do ensemble methods reduce variance?

7. What is the difference between hard and soft voting?

8. What is an example of boosting algorithm?

9. What is the advantage of using ensembles?

10. How are ensemble models implemented in Python?

Experiment: Implement clustering algorithms

1. What is clustering?

2. What is the difference between supervised and unsupervised learning?

3. What is K-means clustering?

4. How does the K-means algorithm work?

5. What is hierarchical clustering?

6. How is the number of clusters chosen?

7. What is inertia in K-means?

8. What are dendrograms?

9. What are real-life applications of clustering?

10. How is clustering implemented in Python?

Experiment: Build simple NN models

1. What is a neural network?

2. What are input, hidden, and output layers?

3. What is an activation function?

4. What is forward propagation?

5. What is backpropagation?

6. What is the loss function?

7. What is the role of the optimizer?

8. What is overfitting and how to avoid it?

9. How do you split data for training and testing?

10. How do you implement a simple NN in Keras?

Experiment: Build deep learning NN models

1. What makes a neural network deep?

2. What is the difference between shallow and deep networks?
3. What is dropout?
4. What are epochs and batch size?
5. What is the vanishing gradient problem?
6. What is ReLU and why is it used?
7. What is a softmax layer?
8. What is early stopping?
9. How do you evaluate model performance?
10. How is a deep learning model implemented in Python?