R.V.S
COLLEGE OF ENGINEERING
DINDIGUL - 624005

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CS3491-ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING LABORATORY

LABORATORY RECORD

Name :_____

Register No. :_____

Department :_____

R.V.S
COLLEGE OF ENGINEERING
DINDIGUL - 624005

Register No:

Certified that this is a bonafide record of the practical work done byMr./Ms.……In…….Semester…… ……..……..Laboratory in the department of …… during the academic year 20-20.

STAFF-IN-CHARGE **HEADOFTHEDEPARTMENT**

Submitted for the practical examination of Anna University, Chennai, at R.V.S College of Engineering held on_____

INTERNAL EXAMINER **EXTERNAL EXAMINER**

R.V.S COLLEGE OF ENGINEERING

VISION

Our VISION is to produce highly Competent Engineers and Quality Technocrats through continual improvement of the standards of excellence in teaching, training and research for economical and societal development of the nation.

MISSION

RVSCOE Strives:

- To provide high quality education through a dynamic curriculum, state-of-the-art facilities, and well trained faculty to prepare students for successful engineering careers.
- To promote research and innovation by providing opportunities and resources to both faculty and students to contribute to the advancement of engineering and technology.
- To forge strong partnerships with industries to enhance practical learning, internships and employment opportunities for students.
- To instill strong ethical values and social responsibility in students to prepare them to be conscientious engineers and leaders in society.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Vision

To impart quality education for producing highly talented globally recognizable technocrats and entrepreneurs with latest knowledge and innovative ideas in Computer Science and Engineering to meet industrial needs and societal expectations.

Mission

1. To impart high standard value-based technical education in all aspects of Computer Science and Engineering through the state of the art infrastructure and innovative approach.

2. Indoctrinate problem solving and team building skills and promote lifelong learning with a sense of societal and ethical responsibilities.

3. To develop globally competent engineers with strong foundations, capable of "think beyond the usual" so as to adapt to the rapidly changing scenarios.

4. Endow with a conducive environment for faculty to engage in and train students in progressive and convergent research themes by establishing Centre of Excellence.

5. Impart high quality experiential learning to get expertise in modern software tools and to outfit to the real time requirements of the industry.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

1. Apply their technical competence in computer science to solve real world problems, with technical and people leadership.
2. Conduct cutting edge research and develop solutions on problems of social relevance. 3. Work in a business environment, exhibiting team skills, work ethics, adaptability and lifelong learning.

PROGRAM OUTCOMES (POs)

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader
in a team, to manage projects and in multidisciplinary environments.

12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSOs)

The Students will be able to

- Exhibit design and programming skills to build and automate business solutions using cutting edge technologies.
- Strong theoretical foundation leading to excellence and excitement towards research, to provide elegant solutions to complex problems.
- Ability to work effectively with various engineering fields as a team to design, build and develop system applications.

CS3492- ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

LIST OF EXPERIMENTS:

1. Implementation of Uninformed search algorithms (BFS)
2. Implementation of Uninformed search algorithms(DFS)
3. Implementation of informed search algorithm (A*)
4. Momory bounded A* (IDA*)
5. Implementation of Naive Bayes models
6. Build a regression models (simple linear regression)
7. Build a regression models (multiple linear regression)
8. Build a decision tree models
9. Build a random forest models
10. Implement clustering algorithms -K-Means
11. Build a support vector machine
12. Build a simple neural network models

C305 **(PE I-5)**	**CS3491**	**ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING**

Course Outcomes (COs)

C305.1	**CO1**:**Use appropriate search algorithms for problem solving**.
C305.2	**CO2: Apply reasoning under uncertainty**.
C305.3	**CO3: Apply reasoning under uncertainty**.
C305.4	**CO4: Apply reasoning under uncertainty**.
C305.5	**CO5: Build deep learning neural network models**.

CO's-PO's & PSO's MAPPING

CO's	**PO's**												**PSO's**		
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**1**	**2**	**3**
1	3	3	3	3	3	-	-	-	2	2	3	1	1	3	3
2	3	3	2	3	2	-	-	-	2	2	3	3	2	3	2
3	3	3	3	2	3	-	-	-	2	2	1	2	2	3	3
4	2	3	3	3	3	-	-	-	2	2	3	2	3	3	2
5	3	3	3	3	3	-	-	-	3	1	3	2	3	2	3
AVg.	2.8	3	2.8	2.8	2.8	-	-	-	2.2	1.8	2.6	2	2.2	2.8	2.6

1 - low, 2 - medium, 3 - high, '-' - no correlation

Ex.No.	Date	Title	Page. no	Marks	Staff sign.
01.		IMPLEMENTATION OF UNINFORMED SEARCH ALGORITHM (BFS)			
02.		IMPLEMENTATION OF UNINFORMED SEARCH ALGORITHM (DFS)			
03.		IMPLEMENTATION OF INFORMED SEARCH ALGORITHM (A*)			
04.		MEMORY-BOUNDED A* (IDA*)			
05.		IMPLEMENTATION OF NAÏVE BAYES MODELS			
06.		BUILD REGRESSION MODELS (SIMPLE LINEAR REGRESSION)			
07.		BUILD REGRESSION MODELS (MULTIPLE LINEAR REGRESSION)			
08.		BUILD DECISION TREE MODELS			
09.		BUILD RANDOM FOREST MODELS			
10.		IMPLEMENT CLUSTERING ALGORITHM – K- MEANS			
11.		BUILD SUPPORT VECTOR MACHINE (SVM) MODELS			
12.		BUILD A SIMPLE NEURAL NETWORK MODEL			

Implementation of Uninformed Search Algorithms (BFS)

Experiment :01

Date:

AIM:

The aim of implementing BFS is algorithms is to traverse a graph or tree datastructure In a systematic way, visiting all nodes and edges in the structures in a particular order.

Objective:

- To understand and implement the **Breadth-First Search (BFS)** algorithm as an uninformed search technique.
- To learn how to explore all nodes in a graph level by level and understand its applications in AI and problem-solving.

Introduction to BFS:

- **Breadth-First Search (BFS)** is a graph traversal algorithm used to explore all the nodes in a graph (or tree) systematically, visiting all the nodes at one level before moving on to the next.
- **Uninformed search** means no additional information (heuristics) is used to guide the search, and all nodes are explored using a straightforward strategy.
- BFS can be used to find the **shortest path** in an unweighted graph because it explores all possibilities level by level.

BFS Algorithm:

1. Initialize a **queue** and enqueue the starting node.
2. Initialize a **visited set** to track visited nodes.
3. While the queue is not empty: ○ Dequeue a node from the front of the queue. ○ If the node has not been visited, mark it as visited.
   - ○ Enqueue all unvisited neighbors of the current node.
4. Repeat until the queue is empty.

```
from collections import deque
# Graph representation using adjacency list graph
= {
    'A': ['B', 'D'],
    'B': ['A', 'C', 'E'],
    'C': ['B'],
    'D': ['A'],
    'E': ['B']
}
order_of_vi
```

```
sit = [] #
List to store
the order of
visited
nodes

while queue:
node = queue.popleft() # Dequeue the front node if
node not in visited:
visited.add(node) # Mark the node as visited order_of_visit.append(node)
# Record the node's visit
        # Add unvisited neighbors to the queue queue.extend(neighbor for
neighbor in graph[node] if neighbor not in visited)

returnorder_of_visit
# Testing BFS
start_node = 'A'
print(f"BFS Order starting from '{start_node}':", bfs(graph, start_node))
```

Steps:

1. Define the graph in the adjacency list format, where each node points to its neighboring nodes.
2. Implement the **BFS function** that takes the graph and the starting node as parameters.
3. Initialize a **visited set** to keep track of visited nodes.
4. Initialize a **queue** to handle nodes for exploration.
5. Use a **while loop** to traverse the graph, dequeuing nodes and adding unvisited neighbors back into the queue.
6. Store the order of visited nodes in the list `order_of_visit`.
7. Print the order of nodes visited by BFS.

Sample Output

BFS Order starting from 'A': ['A', 'B', 'D', 'C', 'E']

Applications of BFS:

- **Shortest Path**: BFS is useful for finding the shortest path in an unweighted graph.
- **Web Crawling**: BFS is used in web crawlers to explore websites level by level.
- **Social Networks**: BFS can help find the shortest path between two people in a social network.
- **Game AI**: BFS can be used in games for pathfinding algorithms where all moves are of equal cost.

Conclusion:

- The **BFS algorithm** was successfully implemented to traverse a graph.
- BFS is an important algorithm in AI and problem-solving, particularly in scenarios where we need to explore a graph systematically.

- Through this experiment, students gain hands-on experience in implementing BFS and understanding its applications in AI.

Course Outcomes:

CO1: Use appropriate search algorithms for problem solving:
CO2: Apply reasoning under uncertainty:
CO3: Build supervised learning models:
CO4: Build ensembling and unsupervised models:

Result:

Thus the program for BFS is executed successfully and output is verified

Experiment: 02 Implementation of Uninformed Search Algorithms (DFS)

Date:

AIM:

The aim of implementing DFS is algorithms is to traverse a graph or tree data structure In a systematic way, visiting all nodes and edges in the structures in a particular order, without re visiting any node twice.

Objective:

- To understand and implement the **Depth-First Search (DFS)** algorithm as an uninformed search technique.
- To explore how DFS traverses a graph by going deep into a branch before backtracking.

Introduction to DFS:

- **Depth-First Search (DFS)** is a graph traversal algorithm that starts at a source node and explores as far as possible along each branch before backtracking.
- Unlike **BFS**, which explores all nodes level by level, **DFS** goes deep into the graph before visiting siblings, making it useful in scenarios such as solving puzzles, finding paths in mazes, or traversing tree structures.
- **DFS** can be implemented using either recursion or an explicit stack.

Pre-requisites:

- Basic understanding of graphs and trees.
- Understanding of stacks and recursive programming.

DFS Algorithm:

1. Initialize a **stack** with the start node (or use recursion).
2. Initialize a **visited set** to keep track of visited nodes.
3. While the stack is not empty:
    - Pop a node from the stack.
    - If the node has not been visited:
        - Mark it as visited.
        - Push all unvisited neighbors of the node onto the stack.
4. Repeat until the stack is empty or the goal is reached.

Code

```
graph = {
   'A': ['B', 'D'],
```

```
    'B': ['A', 'C', 'E'],
    'C': ['B'],
    'D': ['A'],
if visited is None: visited = set() # Initialize visited
set if not provided visited.add(node) # Mark the
node as visited print(node, end=" ") # Print the
node as it's visited for neighbor in graph[node]: if
neighbor not in visited: dfs(graph, neighbor,
visited)

# Testing DFS start_node = 'A' print(f"DFS Traversal starting
from '{start_node}':", end=" ") dfs(graph, start_node)
```

Steps:

1. Define the graph using an adjacency list, where each node points to its neighboring nodes.
2. Implement the **DFS function** using recursion:
    - If the node has not been visited, mark it as visited and print it.
    - Recursively call the DFS function for all unvisited neighbors.
3. Test the DFS function by passing a starting node.
4. The **visited set** ensures nodes are not revisited, preventing infinite loops.

Sample output

DFS Traversal starting from 'A': A B C E D

Applications of DFS:

- **Maze Solving · Pathfinding in Graphs**
- **Topological Sorting**.
- **Cycle Detection**

Conclusion:

- The **DFS algorithm** was successfully implemented to traverse a graph using recursion.
- DFS explores as deep as possible into the graph before backtracking, making it different from BFS.
- This experiment helps students understand the depth-first approach to graph traversal and its applications in real-world problems.

Course Outcomes:

CO1: Use appropriate search algorithms for problem solving:
CO2: Apply reasoning under uncertainty CO3: Build supervised learning models:

CO4: Build ensembling and unsupervised models:
CO5: Build deep learning neural network models:

Result:

Thus the program for DFS is executed successfully and output is verified

Experiment :03 Implementation of Informed Search Algorithms (A*)

Date:

AIM:

The aim of a c program for implementing informed search algorithms like A* efficiently find The shortest path between two points in a graph or network.

Objective:

- To understand and implement the **A*** algorithm, which combines the advantages of both **BFS** and heuristic methods.
- To explore how A* uses a heuristic function to find the shortest path more efficiently than uninformed search algorithms.

Introduction to A* Algorithm:

- **A*** is a search algorithm that uses both the **cost to reach the current node** and an **estimated cost to reach the goal** to determine the best path.
- It is guaranteed to find the shortest path in a weighted graph if an admissible heuristic is used.
- The A* algorithm works by maintaining a **priority queue** where each node has a **cost function**f(n) that is the sum of:
  - g(n): The actual cost to reach node nnn. ○ h(n): The heuristic estimate of the cost to reach the goal from node nnn.

  f(n)=g(n)+h(n)

- The search explores the node with the lowest f(n) value first.

Pre-requisites:

- Basic understanding of graphs, trees, and search algorithms.
- Understanding of **heuristics** and their role in search algorithms.

A* Algorithm:

1. Initialize two sets: ○ **Open Set**: A priority queue containing nodes to be evaluated, starting with the initial node.
   - **Closed Set**: A set containing nodes already evaluated.
2. Initialize the **cost function**f(n)=g(n)+h(n) for all nodes.
3. While the open set is not empty: ○ Get the node n with the lowest f(n). ○ If n is the goal node, reconstruct the path and return the result. ○ Otherwise, move node n from the open set to the closed set.
   - For each neighbor of node n:
     - Calculate its g and h values.

Code

importheapq

```
# Graph representation with weights (Adjacency list format) graph
= {
   'A': {'B': 1, 'C': 4},
   'B': {'A': 1, 'C': 2, 'D': 5},
   'C': {'A': 4, 'B': 2, 'D': 1},
   'D': {'B': 5, 'C': 1}
}
heuristic = {
   'A': 7,
   'B': 6,
   'C': 2,
   'D': 0 # Goal node (D)
}

defa_star(graph, start, goal, heuristic):
   # Open set (priority queue), stores nodes to be evaluated with their f values open_set
= []
heapq.heappush(open_set, (0 + heuristic[start], start)) # Push initial node into the queue with f-
value (g + h)

   # Maps to track the g and f values of each node g_values
= {start: 0}
f_values = {start: heuristic[start]}

   # Parent map to reconstruct the path came_from
= {}

whileopen_set:
      # Get the node with the lowest f-value current_f,
current = heapq.heappop(open_set)

      # If we have reached the goal, reconstruct the path
if current == goal: path = [] while current in
came_from: path.append(current) current =
came_from[current]
path.append(start)
return path[::-1] # Reverse to get path from start to goal

      # For each neighbor of the current node for
neighbor, weight in graph[current].items():
tentative_g = g_values[current] + weight # Cost
to reach the neighbor

         # If this path is better, update the f, g, and parent values
if neighbor not in g_values or tentative_g<g_values[neighbor]:
came_from[neighbor] = current g_values[neighbor] =
tentative_g f_values[neighbor] = tentative_g +
```

```
heuristic[neighbor]                    heapq.heappush(open_set,
(f_values[neighbor], neighbor))

return None # If no path is found

# Testing A* algorithm
start_node     =     'A'
goal_node = 'D'
path = a_star(graph, start_node, goal_node, heuristic)
print(f"Path found by A* from '{start_node}' to '{goal_node}': {path}")
```

Steps:

1. **Graph Representation**: Represent the graph using an adjacency list, where each node points to its neighboring nodes with associated edge weights.
2. **Heuristic**: Define a heuristic function that estimates the cost from each node to the goal.
3. **A* Function**:
    - Initialize the open set as a priority queue. ○ Define **g-values** (cost to reach a node) and **f-values** (total estimated cost).
    - While the open set is not empty, evaluate nodes by picking the one with the lowest **f-value** and explore its neighbors.
    - Track the path using a **came_from** map to reconstruct the solution once the goal is found.
4. **Output**: If a path is found, return it; otherwise, return `None`.

Sample output

Path found by A* from 'A' to 'D': ['A', 'C', 'D']

Applications of A*:

- **Pathfinding in Games**: A* is commonly used in game development for pathfinding algorithms to navigate characters or agents on maps.
- **Robotics**: In robot motion planning, A* is used to find the optimal path to a target while avoiding obstacles.
- **Route Planning**: A* is widely used in GPS and route navigation systems to find the fastest or shortest route.

Conclusion:

- The **A*** algorithm was successfully implemented to find the shortest path in a weighted graph using a heuristic.
- A* improves upon **BFS** by considering both actual cost and heuristic estimates, making it much more efficient for pathfinding problems.
- Through this experiment, students learn the power of informed search and the importance of heuristics in AI problem-solving.

Course Outcomes:

CO1: Use appropriate search algorithms for problem solving:
CO2: Apply reasoning under uncertainty:
CO3: Build supervised learning models:
CO4: Build ensembling and unsupervised models:.
CO5: Build deep learning neural network models:

Result:

Thus the program for implementing informed search algorithms like A* has verified successfully

Experiment :04 **Memory-Bounded A* (IDA*)**

Date:

AIM:

The aim of a c program for implementing memory -bounded A* efficiently find The shortest path between two points in a graph .

Objective

To implement the **Iterative Deepening A*** (IDA*) algorithm, which combines the space efficiency of depth-first search with the optimality of A* search, for finding paths in a graph using limited memory.

Introduction to the Algorithm

The Iterative Deepening A* (IDA*) algorithm is a memory-bounded heuristic search technique. Unlike A*, which maintains open and closed lists in memory, IDA* performs a series of depth-first searches with increasing cost thresholds. This approach helps in reducing memory consumption while still leveraging heuristic information. It is ideal for large graphs where A* may run out of memory.

Pre-requisite Knowledge

- Graph representations (adjacency list)
- Depth-First Search (DFS)
- Heuristic search: f(n) = g(n) + h(n)
- Understanding of cost, path, and nodes in a graph

Algorithm: Iterative Deepening A*

1. Set the initial threshold as the heuristic of the start node.
2. Perform a depth-first search (DFS) with the threshold as a cost limit.
3. For each node:
    - Calculate f(n) = g(n) + h(n) ○ If f(n) exceeds the threshold, return that f-value.
    - If the goal is found, return the path.
4. After each DFS iteration: ○ If the goal was not found, update the threshold to the minimum f-value that exceeded the threshold.
5. Repeat until the goal is found or no path exists.

Code

```
graph = {
   'A': [('B', 1), ('C', 4)],
```

```
    'B': [('C', 2), ('D', 5)],
    'C': [('D', 1)],
    'D': []
}

heuristic = {
    'A': 7,
    'B': 6, 'C':
    2,
    'D': 0 }

defida_star(start, goal):
defdfs(node, g, threshold, path):
        f = g + heuristic[node]
if f > threshold:
return  f  if  node
== goal:
return "FOUND", path + [node] min_cost = float('inf')
for neighbor, cost in graph.get(node, []): if neighbor not
in path: result = dfs(neighbor, g + cost, threshold, path +
[node]) ifisinstance(result, tuple) and result[0] ==
"FOUND":
return result elifisinstance(result,
(int,     float)):     min_cost     =
min(min_cost,                result)
returnmin_cost

threshold = heuristic[start] while
True:
result = dfs(start, 0, threshold, []) ifisinstance(result,
tuple) and result[0] == "FOUND": return result[1] elif
result == float('inf'):
return None
threshold = result

#          Execute
start_node = 'A'
goal_node = 'D'
path = ida_star(start_node, goal_node)
print(f"Path found by IDA* from '{start_node}' to '{goal_node}': {path}")
```

Sample Output

Path found by IDA* from 'A' to 'D': ['A', 'B', 'C', 'D']

Steps to Execute

1. Define the graph using an adjacency list with edge weights.
2. Define a heuristic function for each node.
3. Implement the IDA* search with recursive DFS bounded by a threshold.
4. Start the search from the start node to the goal node.
5. Print the final path returned by the IDA* algorithm.

Applications

- Game AI (memory-limited environments)
- Robotics (pathfinding in large maps)
- Puzzle solvers (like 15-puzzle, sliding tiles)
- Navigation systems with limited memory

Conclusion

- The IDA* (Iterative Deepening A*) algorithm was successfully implemented to find the shortest path in a weighted graph using a heuristic with limited memory usage.
- IDA* combines the benefits of A*'s informed search with depth-first search's low memory footprint, making it suitable for memory-constrained environments.
- Through this experiment, students understand how iterative deepening and heuristics can be integrated to solve complex pathfinding problems efficiently with minimal memory consumption.

Result:

Thus the program for implementing memory-bounded search algorithms like A* has verified successfully

Experiment :05 **Implementation of Naïve Bayes Models**

Date:

AIM:

The aim of the naïve bayes algorithms is to classify a given set of data points into different class Based on the probability of each data point belonging to particular class.

Objective

To implement a Naïve Bayes classifier to predict whether a person will purchase a product based on their **age** and estimated salary, using the **Social Network Ads** dataset.

Introduction to Naïve Bayes

Naïve Bayes is a simple yet powerful classification algorithm based on **Bayes' Theorem**, a fundamental concept in probability theory. It belongs to the family of **supervised learning** algorithms and is especially suited for **classification problems**.

The algorithm is called "naïve" because it assumes that all features (attributes) are **independent** of each other — an assumption rarely true in real-world data but still produces excellent results in many scenarios.

The **Gaussian Naïve Bayes** variant is typically used when the features are **continuous-valued and follow a normal distribution**, as is the case with Age and EstimatedSalary.

Bayes' Theorem:

$$P(C|X)=P(X|C)\cdot P(C)/P(X)$$

P(C|X): Posterior probability (probability of class C given data X)

P(X|C): Likelihood (probability of data X given class C)

P(C): Prior probability of class C

P(X): Probability of data X

Pre-requisite Knowledge

- Basic understanding of probability
- Concept of conditional probability and Bayes' Theorem
- Supervised learning (training a model on labeled data)
- Familiarity with Python and libraries like pandas and scikit-learn

Algorithm Steps

1. **Import** necessary libraries and load the dataset.
2. **Preprocess** the dataset (handle missing values if any, extract required columns).
3. **Split** the dataset into training and test sets.
4. **Train** the Gaussian Naïve Bayes classifier on the training data.
5. **Make predictions** on the test set.
6. **Evaluate** the model using metrics like accuracy and confusion matrix.

Code

```
import pandas as pd fromsklearn.model_selection
import train_test_split fromsklearn.naive_bayes
import GaussianNB
fromsklearn.metrics import accuracy_score, confusion_matrix

# Step 1: Load the dataset
df = pd.read_csv("Social_Network_Ads.csv")

# Step 2: Feature selection
X = df[['Age', 'EstimatedSalary']] # Independent variables y
= df['Purchased'] # Dependent variable

# Step 3: Split into training and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)

# Step 4: Train the model model
= GaussianNB()
model.fit(X_train, y_train)

# Step 5: Predict the test results y_pred
= model.predict(X_test)

# Step 6: Evaluate the model accuracy =
accuracy_score(y_test, y_pred) conf_matrix =
confusion_matrix(y_test, y_pred)

# Step 7: Display results
print("Accuracy of Naïve Bayes model:", accuracy) print("Confusion
Matrix:\n", conf_matrix)
```

sample output:

Accuracy of Naïve Bayes model: 0.89

Confusion Matrix:

[[63 5]

[3 29]]

Applications of Naïve Bayes

- Email spam detection
- Sentiment analysis
- Credit scoring
- Medical diagnosis
- Real-time ad targeting and personalization

Conclusion

- The Naïve Bayes classifier was successfully implemented using real-world data.
- This experiment demonstrates how probabilistic reasoning and statistical learning are applied to solve classification problems.
- Students gained hands-on experience with feature selection, model training, and evaluation using performance metrics.

Course Outcome

CO3: Build supervised learning models

Result:

Thus the program for naive bayes is verified successfully and output is verified

Experiment :06 Build Regression Models (Simple Linear Regression)
Date:

AIM:

The aim to building a simple linear regression model to understand and quantify the relationship Between one independent variable and one dependent variable.

Objective

To implement a simple linear regression model that predicts salary based on years of experience using a given dataset.

Introduction

Simple Linear Regression is a fundamental machine learning technique used for predicting a continuous output variable based on a single input variable. It assumes a linear relationship between the input and output variables. In this experiment, we aim to predict an employee's salary based on their years of experience.

The mathematical equation for simple linear regression is:

$$Salary=b0+b1\times(Years\_of\_Experience)$$

b0 is the y-intercept b1 is the slope

(regression coefficient)

The model tries to minimize the error between predicted and actual salary using least squares method.

Pre-requisites

- Understanding of supervised learning
- Basics of Python, pandas, matplotlib, and scikit-learn
- Concept of regression and evaluation metrics (MSE, R^2)

Algorithm

1. Import the required libraries and load the dataset.
2. Select 'Years of Experience' as the independent variable and 'Salary' as the dependent variable.
3. Split the data into training and testing sets.
4. Train the linear regression model on the training data.

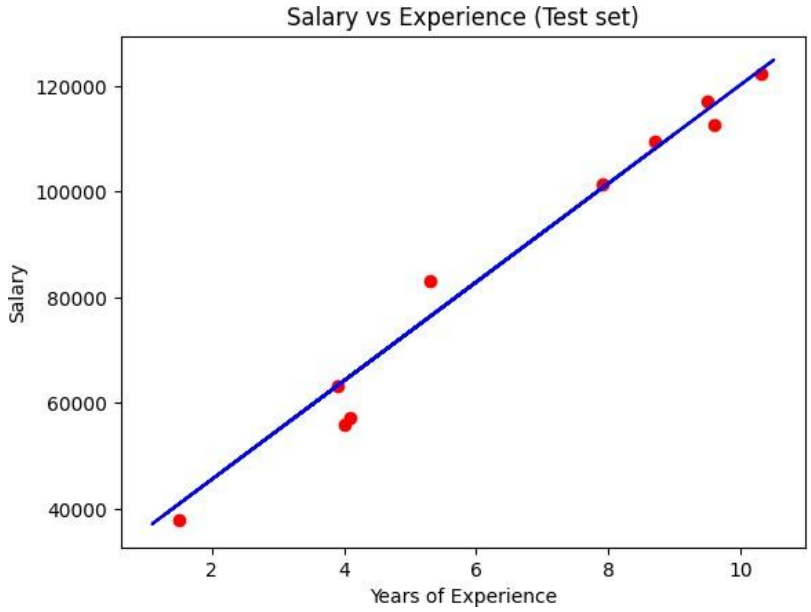
5. Predict salary values on the test data.

Code

```
# Simple Linear Regression

# Importing the libraries
importnumpy as np
importmatplotlib.pyplot as
plt import pandas as pd

# Importing the dataset
dataset = pd.read_csv('Salary_Data.csv')
X = dataset.iloc[:, :-1].values y
= dataset.iloc[:, -1].values

# Splitting the dataset into the Training set and Test set fromsklearn.model_selection
import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 1/3, random_state = 0)

# Training the Simple Linear Regression model on the Training set
fromsklearn.linear_model import LinearRegression regressor =
LinearRegression() regressor.fit(X_train, y_train)

# Predicting the Test set results y_pred
= regressor.predict(X_test)

# Visualising the Training set results plt.scatter(X_train,
y_train, color = 'red')
plt.plot(X_train, regressor.predict(X_train), color = 'blue')
plt.title('Salary vs Experience (Training set)')
plt.xlabel('Years of Experience')
plt.ylabel('Salary')
plt.show()

# Visualising the Test set results plt.scatter(X_test,
y_test, color = 'red')
plt.plot(X_train, regressor.predict(X_train), color = 'blue')
plt.title('Salary vs Experience (Test set)') plt.xlabel('Years
of Experience')
plt.ylabel('Salary')
plt.show()
r2 = r2_score(y_test, y_pred) print("R²
Score:", r2
```

Sample Output

```
R² Score: 0.9749154407708353
```



Applications

- Predicting salary based on work experience
- Forecasting production based on working hours
- Sales prediction based on marketing spend
- Cost estimation in engineering based on project size

Conclusion

- A simple linear regression model was successfully implemented and trained to predict salary using years of experience.
- The model performed well, demonstrating a strong linear relationship and providing a solid understanding of regression fundamentals.
- This experiment provided insights into model training, testing, evaluation, and visualization in supervised learning.

Course Outcome

CO3: Build supervised learning models

Result:

Thus the program for building a simple linear regression model has been successfully verified and the Output is verified.

Experiment :07 Build Regression Models(multiple linear regression)

Date:

Aim:

The aim of building a multiple linear regression model is to establish a predictive that uses multiple independent variables to forecast a single dependent variable.

Objective

To build a multiple linear regression model that predicts the profit of a startup based on its expenditures and location using the 50_Startups dataset.

Introduction

Multiple Linear Regression is an extension of Simple Linear Regression where the dependent variable is predicted using two or more independent variables. It assumes a linear relationship between dependent and independent variables.

The model equation is:

$$Profit=b0+b1\times(R\&D\ Spend)+b2\times(Administration)+b3\times(Marketing\ Spend)$$

This technique helps identify how multiple factors influence a single outcome (here, the startup's profit).

Pre-requisites

- Basics of Linear Algebra and Statistics
- Python programming with pandas and scikit-learn
- Understanding of one-hot encoding for categorical data

Algorithm

1. Import the dataset and necessary libraries.
2. Encode categorical variables (e.g., State).
3. Separate features and target variable (Profit).
4. Split the dataset into training and test sets.
5. Train the multiple linear regression model.
6. Predict the results on the test set.
7. Evaluate the model performance.
8. Interpret the coefficients and output.

```
# Importing the libraries
importnumpy as np
```

```
importmatplotlib.pyplot as
plt import pandas as pd
# Importing the dataset
dataset = pd.read_csv('50_Startups.csv')
X = dataset.iloc[:, :-
1].values y = dataset.iloc[:, -
1].values print(X)

# Encoding categorical data fromsklearn.compose import
ColumnTransformer fromsklearn.preprocessing import OneHotEncoder ct
= ColumnTransformer(transformers=[('encoder', OneHotEncoder(), [3])],
remainder='passthrough') X = np.array(ct.fit_transform(X)) print(X)

# Splitting the dataset into the Training set and Test set fromsklearn.model_selection
import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)

# Training the Multiple Linear Regression model on the Training set
fromsklearn.linear_model import LinearRegression regressor =
LinearRegression() regressor.fit(X_train, y_train)

# Predicting the Test set results
y_pred =
regressor.predict(X_test)
np.set_printoptions(precision=2)
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))

r2 = r2_score(y_test, y_pred)
print("R² Score:", r2)
print("Intercept:",
model.intercept_)
print("Coefficients:", model.coef_)
```

Sample output

```
R² Score: 0.9347068473282546
Intercept: 25478.12743600038
Coefficients: [9371.02]
```

Applications

- Financial forecasting in startups and enterprises
- Real estate price prediction
- Business decision support systems
- Cost prediction in manufacturing based on multiple factors

Conclusion

- Multiple Linear Regression was successfully implemented on the 50 Startups dataset.

- The model provided insights into how various expenditures and the state location affect a startup's profitability.
- Students learned how to preprocess data, perform feature encoding, and build a regression model using real-world datasets.

Course Outcome

CO3: Build supervised learning models

Result:

Thus the program of building a multiple linear regression model has been successfully Executed and the output is verified.

Experiment :08 **Build Decision Tree Models**

Date:

Aim:

The aim for building a decision tree models the dataset into increasingly smaller according to the most crucial attribute creating a tree structure with decision nodes feature tests and leaf nodes representing outcomes.

Objective

To implement a Decision Tree Classifier using the Social Network Ads dataset to predict whether a user will purchase a product based on Age and Estimated Salary.

Introduction

A Decision Tree is a supervised machine learning algorithm used for classification and regression tasks. It mimics human decision-making by learning decision rules inferred from the data features. Each internal node represents a decision on a feature, each branch represents the outcome, and each leaf node represents a class label (decision).

In classification, Decision Trees split data recursively based on attribute values that maximize information gain or minimize impurity (e.g., Gini Index or Entropy).

Pre-requisites

- Understanding of supervised learning
- Python programming basics
- Familiarity with pandas, matplotlib, and scikit-learn
- Knowledge of classification metrics (accuracy, confusion matrix, etc.)

Algorithm

1. Import the dataset and necessary libraries.
2. Preprocess the data (select features, handle labels).
3. Split the data into training and test sets.
4. Train a Decision Tree classifier on the training data.
5. Predict outcomes on the test data.
6. Evaluate the model using accuracy, confusion matrix, etc.
7. Visualize the decision boundary and tree.

Code

```
importnumpy as np
importmatplotlib.pyplot as
plt import pandas as pd
X = dataset.iloc[:, :-1].values y
= dataset.iloc[:, -1].values

# Splitting the dataset into the Training set and Test set fromsklearn.model_selection
import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, random_state =
0) print(X_train) print(y_train) print(X_test) print(y_test)

# Feature Scaling
fromsklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train =
sc.fit_transform(X_train) X_test =
sc.transform(X_test)
print(X_train)
print(X_test)

# Training the Decision Tree Classification model on the Training set fromsklearn.tree
import DecisionTreeClassifier
classifier = DecisionTreeClassifier(criterion = 'entropy', random_state = 0) classifier.fit(X_train,
y_train)

# Predicting a new result
print(classifier.predict(sc.transform([[30,87000]])))

# Predicting the Test set results y_pred
= classifier.predict(X_test)
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))

# Making the Confusion Matrix
fromsklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, y_pred) print(cm)
print(accuracy_score(y_test, y_pred))
```

Sample Output

Confusion Matrix

```
[[62 6]
[ 3 29]]
```

Accuracy

0.91

Applications

- Customer behavior prediction
- Medical diagnosis systems

- Loan approval decisions
- Fraud detection

Conclusion

- The Decision Tree Classifier was implemented and evaluated on a real-world dataset.
- It successfully predicted purchasing behavior based on demographic features.
- Students learned how tree-based models split data, how entropy works, and how to visualize decision-making.

Course Outcome

CO3: Build supervised learning models

Result:

Thus the program of building decision tree model is executed successfully and output is Verified.

Experiment:09 **Build Random Forest Models**
Date:

Aim:

The aim for building random forest models

Objective

To implement a Random Forest Classifier using the Social Network Ads dataset to predict whether a user will purchase a product based on Age and Estimated Salary.

Introduction

Random Forest is an ensemble learning method that builds multiple decision trees and merges their results to improve accuracy and control overfitting. It operates by training each tree on a random subset of the data and features (bagging). The final classification is typically determined by a majority vote among the individual trees.

This method increases model robustness and generalization compared to a single decision tree, which may overfit the training data.

Pre-requisites

- Knowledge of Decision Trees
- Understanding of supervised learning
- Python programming and scikit-learn
- Concepts of overfitting and ensemble methods

Algorithm

1. Import required libraries and the dataset.
2. Preprocess the dataset: extract features and target variable.
3. Scale the features using standardization.
4. Split the dataset into training and testing sets.
5. Fit a Random Forest Classifier to the training data.
6. Predict test results.
7. Evaluate the model using confusion matrix and accuracy.

Code

```
importnumpy as np
import pandas as pd

# Importing the dataset
dataset = pd.read_csv('Social_Network_Ads.csv')
```

```
X = dataset.iloc[:, :-1].values y
= dataset.iloc[:, -1].values

# Splitting the dataset into the Training set and Test set fromsklearn.model_selection
import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, random_state =
0) print(X_train) print(y_train) print(X_test)
print(y_test)

# Feature Scaling
fromsklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train                              =
sc.fit_transform(X_train)  X_test =
sc.transform(X_test)
print(X_train) print(X_test)

# Training the Random Forest Classification model on the Training set fromsklearn.ensemble
import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators = 10, criterion = 'entropy', random_state = 0)
classifier.fit(X_train, y_train)

# Predicting a new result
print(classifier.predict(sc.transform([[30,87000]])))

# Predicting the Test set results y_pred
= classifier.predict(X_test)
print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))

# Making the Confusion Matrix
fromsklearn.metrics import confusion_matrix, accuracy_score
cm     =     confusion_matrix(y_test,     y_pred)     print(cm)
accuracy_score(y_test, y_pred)
```

Sample output

Confusion Matrix

```
[[63 5] [
4 28]]
```

Accuracy 0.91
Analysis

- By combining multiple trees, it reduces variance and the risk of overfitting.
- With only 10 trees (n_estimators=10), it already performs well. Accuracy may increase with more trees.
- It handles feature importance effectively and is robust to noise.

Applications

- Credit scoring and risk assessment
- Medical diagnostics
- Customer segmentation and behavior analysis
- Industrial fault detection

Conclusion

- A Random Forest Classifier was successfully built and tested on a real-world dataset.
- It provided high accuracy and demonstrated the power of ensemble methods over individual models.
- This experiment helps students understand how combining models can lead to better performance and stability.

Course Outcome

CO4: Build ensembling and unsupervised models

Result:

Thus the program of building random forest models is successfully executed and output is Verified.

Implement Clustering Algorithm – K-Means

Experiment: 10
Date:

Aim:

The aim of implementing clustering algorithm--k-means to minimize the total Distance between points and their assigned cluster centroid.

Objective

To implement K-Means Clustering to segment customers based on annual income and spending score using the Mall Customers dataset.

Introduction

K-Means is an unsupervised learning algorithm used for clustering. It partitions a dataset into *K* distinct, non-overlapping clusters. The algorithm iteratively assigns data points to the nearest cluster centroid and updates the centroids based on the mean of assigned points until convergence.

This technique is widely used in market segmentation, image compression, pattern recognition, and more.

Pre-requisites

- Understanding of unsupervised learning
- Basic knowledge of Euclidean distance
- Familiarity with Python, NumPy, Pandas, Matplotlib, and scikit-learn

Algorithm Steps

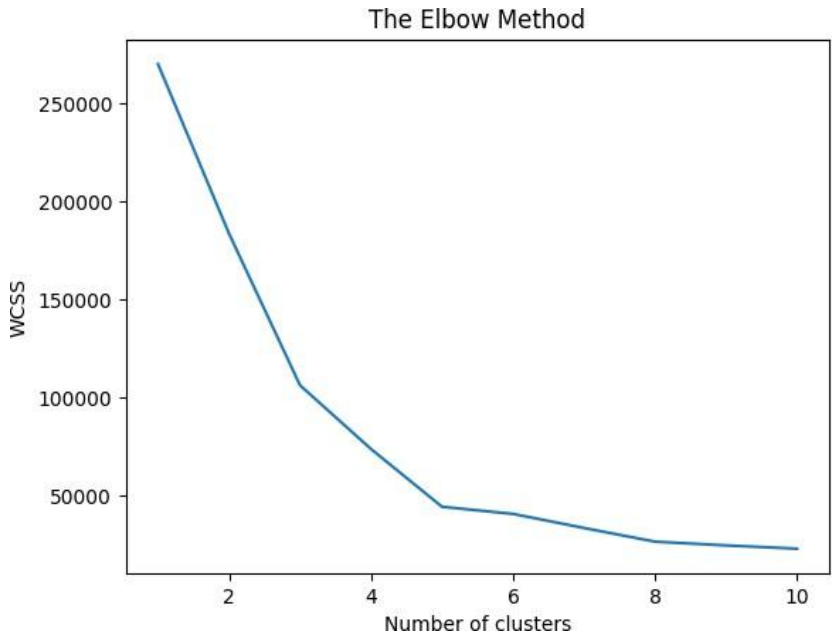
1. Load and explore the dataset.
2. Select relevant features (Annual Income, Spending Score).
3. Use the Elbow method to determine the optimal number of clusters.
4. Fit the K-Means algorithm with the chosen number of clusters.
5. Visualize the clusters.

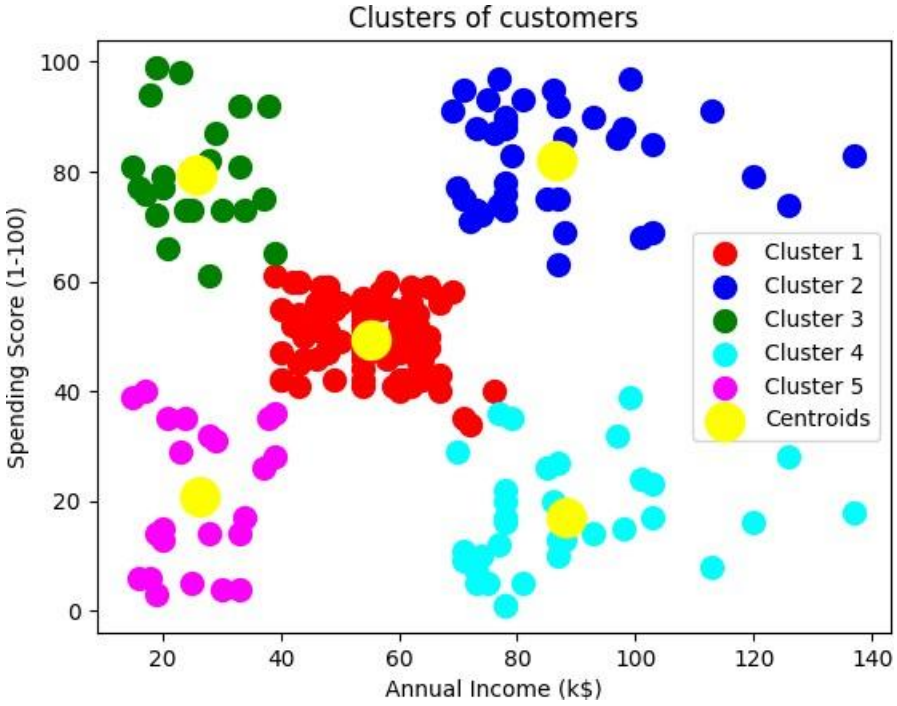
Code

```
importnumpy as np
importmatplotlib.pyplot as
plt import pandas as pd

# Importing the dataset
dataset = pd.read_csv('Mall_Customers.csv')
X = dataset.iloc[:, [3, 4]].values
```

```
# Using the elbow method to find the optimal number of clusters fromsklearn.cluster
import KMeans
wcss = [] for i in
range(1, 11):
kmeans = KMeans(n_clusters = i, init = 'k-means++', random_state = 42)
kmeans.fit(X) wcss.append(kmeans.inertia_) plt.plot(range(1, 11), wcss)
plt.title('The Elbow Method') plt.xlabel('Number of clusters')
plt.ylabel('WCSS') plt.show()

# Training the K-Means model on the dataset
kmeans = KMeans(n_clusters = 5, init = 'k-means++', random_state = 42) y_kmeans
= kmeans.fit_predict(X)

# Visualising the clusters plt.scatter(X[y_kmeans == 0, 0], X[y_kmeans == 0, 1], s = 100, c =
'red', label = 'Cluster 1') plt.scatter(X[y_kmeans == 1, 0], X[y_kmeans == 1, 1], s = 100, c = 'blue',
label = 'Cluster 2') plt.scatter(X[y_kmeans == 2, 0], X[y_kmeans == 2, 1], s = 100, c = 'green',
label = 'Cluster 3') plt.scatter(X[y_kmeans == 3, 0], X[y_kmeans == 3, 1], s = 100, c = 'cyan',
label = 'Cluster 4') plt.scatter(X[y_kmeans == 4, 0], X[y_kmeans == 4, 1], s = 100, c = 'magenta',
label = 'Cluster 5') plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s =
300, c = 'yellow', label = 'Centroids') plt.title('Clusters of customers') plt.xlabel('Annual Income
(k$)') plt.ylabel('Spending Score (1-100)') plt.legend() plt.show()
```

Sample output:





Analysis:

- The algorithm grouped customers into 5 segments based on income and spending score.
- Helps understand customer behavior, such as identifying high-income, low-spending or low-income, high-spending groups.
- Useful for targeted marketing and customer retention strategies.

Applications

- Customer segmentation in retail and banking
- Image compression
- Social network analysis

Conclusion

- K-Means clustering was successfully implemented on a real-world dataset.
- The Elbow method helped determine the optimal cluster number.
- Visualizing the clusters provided insights into different customer groups.

Course Outcome

CO4: Build ensembling and unsupervised models

Result:

Thus the program of implementing clustering algorithms k-means has been successfully Executed and output is verified.

Experiment 11: Random Forest Classifier

Date:

Aim:

The aim of random forest classifier that uses many decision trees to make Predictions it takes different random parts of the dataset to train each tree and then it Combines the results by averaging them.

Objective

To implement and evaluate a **Random Forest classifier** to predict whether a user purchases a product based on their age and estimated salary using the Social Network Ads dataset.

Introduction to Random Forest

Random Forest is an ensemble learning algorithm that builds multiple decision trees during training and outputs the **mode of the classes** (classification) or **mean prediction** (regression) of the individual trees. It is robust, reduces overfitting, and increases accuracy by averaging out individual tree errors.

Unlike a single decision tree, which might overfit the training data, Random Forest reduces variance by training multiple trees on different samples and features.

Pre-requisites

- Understanding of supervised learning (classification)
- Knowledge of decision trees
- Familiarity with Python, pandas, matplotlib, seaborn, and scikit-learn

Algorithm

1. Import the dataset and required libraries.
2. Preprocess the data: select features, encode if needed, split into training and testing sets.
3. Train a Random Forest Classifier on the training set.
4. Predict results on the test set.
5. Evaluate the model using confusion matrix and accuracy score.
6. Visualize the decision boundaries.

Code

```
import pandas as pd importnumpy as np
importmatplotlib.pyplot as plt
fromsklearn.model_selection import train_test_split
fromsklearn.preprocessing import StandardScaler
fromsklearn.ensemble import RandomForestClassifier fromsklearn
```

```
# Load the dataset dataset =
pd.read_csv('Social_Network_Ads.csv') X =
dataset[['Age', 'EstimatedSalary']].values y =
dataset['Purchased'].values

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)

# Feature scaling sc =
StandardScaler()
X_train = sc.fit_transform(X_train) X_test
= sc.transform(X_test)

# Training the Random Forest model
classifier = RandomForestClassifier(n_estimators=100, criterion='entropy', random_state=0)
classifier.fit(X_train, y_train) y_pred = classifier.predict(X_test)

# Evaluation
cm = confusion_matrix(y_test, y_pred) accuracy
= accuracy_score(y_test, y_pred)

print("Confusion Matrix:\n", cm)
print("Accuracy: {:.2f}%".format(accuracy * 100))

# Optional Visualization (for 2D data)
frommatplotlib.colors import ListedColormap

X_set, y_set = X_test, y_test X1, X2 = np.meshgrid(
np.arange(start=X_set[:, 0].min()-1, stop=X_set[:, 0].max()+1, step=0.01),
np.arange(start=X_set[:, 1].min()-1, stop=X_set[:, 1].max()+1, step=0.01)
)
plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
alpha=0.75, cmap=ListedColormap(('red', 'green')))
plt.scatter(X_set[:, 0], X_set[:, 1], c=y_set,
cmap=ListedColormap(('red', 'green')))
plt.title('Random Forest Classification (Test set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.show()
```

Result:

Thus the program of random forest classifier has been executed successfully and output is Verified.

Build Support Vector Machine (SVM) Models

Experiment :11
Date:

Aim:

The aim of building support vector Machine (SVM) Models is to find the best Possible line or decision boundary that separates the data points of different data.

Objective

To implement and evaluate a **Support Vector Machine (SVM)** classifier to predict whether a user purchases a product based on their age and estimated salary using the **Social Network Ads** dataset.

Introduction to SVM

Support Vector Machine is a powerful supervised learning algorithm used for classification and regression tasks. In classification, it finds the optimal **hyperplane** that separates data points of different classes with the **maximum margin**.

SVM can also handle **non-linear classification** by applying the **kernel trick**, which transforms the input space into a higher-dimensional space to make data linearly separable.

Pre-requisites

- Basics of classification
- Understanding of decision boundaries and margin
- Familiarity with Python, pandas, matplotlib, seaborn, and scikit-learn

Algorithm

1. Load and preprocess the dataset.
2. Scale the features.
3. Split the data into training and test sets.
4. Train an SVM classifier on the training set.
5. Predict the test set results.
6. Evaluate using confusion matrix and accuracy score.
7. Visualize the decision boundaries (for 2D data).

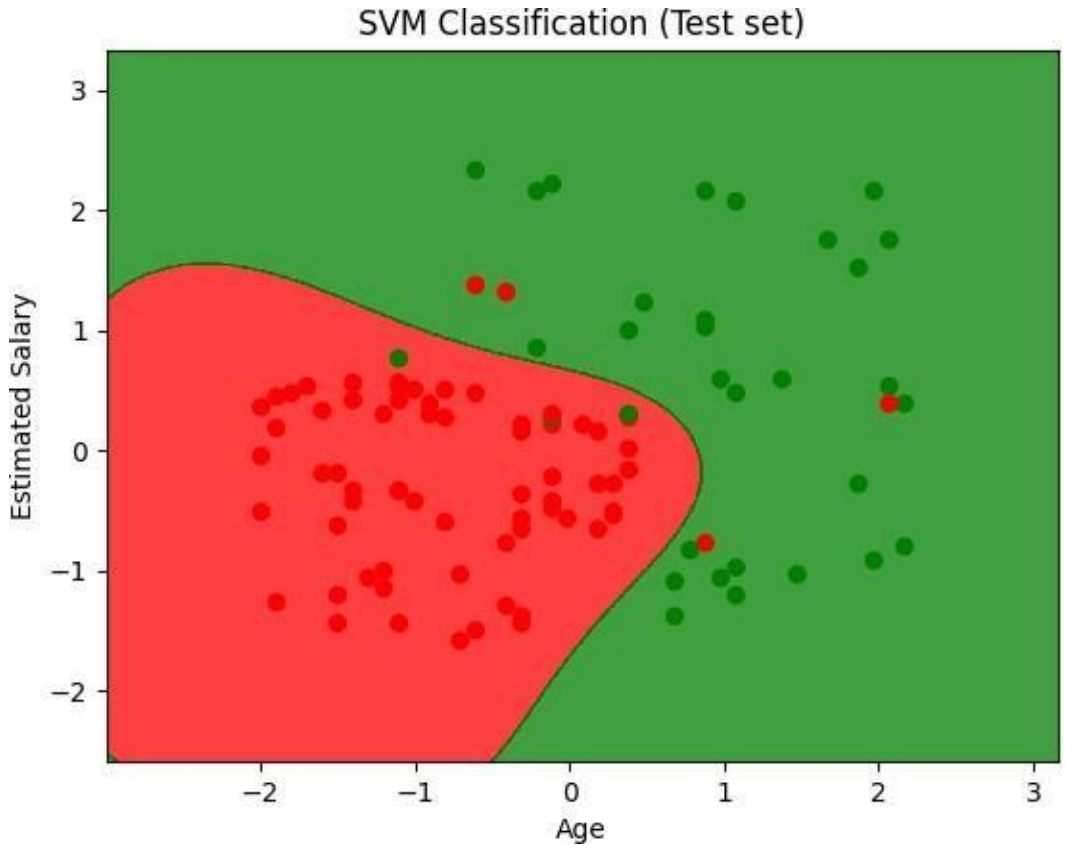
Code

```
import pandas as pd
importnumpy as np
```

```
importmatplotlib.pyplot     as
plt
fromsklearn.model_selection import train_test_split
fromsklearn.preprocessing import StandardScaler
fromsklearn.svm import SVC
fromsklearn.metrics import confusion_matrix, accuracy_score

#          Load          dataset          dataset          =
pd.read_csv('Social_Network_Ads.csv')    X    =
dataset[['Age',  'EstimatedSalary']].values  y  =
dataset['Purchased'].values

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)

# Feature Scaling sc
= StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# Train SVM model (using RBF kernel)
classifier          =          SVC(kernel='rbf',
random_state=0)          classifier.fit(X_train,
y_train)

# Predict test results
y_pred = classifier.predict(X_test)

cm = confusion_matrix(y_test, y_pred) accuracy
= accuracy_score(y_test, y_pred)

print("Confusion Matrix:\n", cm)
print("Accuracy: {:.2f}%".format(accuracy * 100))

# Optional Visualization (for 2D) frommatplotlib.colors
import ListedColormap

X_set, y_set = X_test, y_test X1, X2 = np.meshgrid( np.arange(start=X_set[:,
0].min() - 1, stop=X_set[:, 0].max() + 1, step=0.01), np.arange(start=X_set[:,
1].min() - 1, stop=X_set[:, 1].max() + 1, step=0.01)
)
plt.contourf(X1,  X2,  classifier.predict(np.array([X1.ravel(),  X2.ravel()]).T).reshape(X1.shape),
alpha=0.75, cmap=ListedColormap(('red', 'green')))
plt.scatter(X_set[:, 0], X_set[:, 1], c=y_set,
cmap=ListedColormap(('red',      'green')))
plt.title('SVM  Classification  (Test  set)')
plt.xlabel('Age')
plt.ylabel('Estimated                Salary')
plt.show()
```

Sample output

```
Confusion Matrix:
 [[64  4]
[ 3 29]]
Accuracy: 93.00%
```



Analysis

- SVM with RBF kernel provided high accuracy and good generalization.
- It is robust to overfitting in high-dimensional spaces.
- Visualization of decision boundaries shows SVM effectively separated the classes.

Applications

- Handwriting recognition
- Facial expression recognition
- Medical diagnosis (e.g., cancer detection)
- Email spam classification
- Bioinformatics (e.g., gene classification)

Conclusion

- The SVM classifier was implemented using the Social Network Ads dataset.
- It successfully predicted whether users purchased a product based on Age and Estimated Salary.

- Students understood the working of the SVM algorithm and the importance of kernels for non-linear problems.

Course Outcome

CO3: Build supervised learning models

Result:

Thus the program of building support vector machine(SVM) models has been Successfully Executed and output is verified.

Experiment :12 **Build a Simple Neural Network Model**

Date:

Aim:

The aim of building a simple neural network model is to learn from data and Make predictions.

Objective

To build and train a **simple Artificial Neural Network (ANN)** using a supervised dataset to predict binary classification outcomes (e.g., purchase prediction using Social Network Ads dataset).

Introduction to Neural Networks

Artificial Neural Networks are inspired by the structure and functioning of the human brain. A simple neural network, also called a feedforward network, consists of:

- **Input layer**: Takes features as input
- **Hidden layers**: Perform intermediate computations using activation functions
- **Output layer**: Produces the final result (e.g., class prediction)

ANNs can model complex, non-linear relationships and are widely used in modern AI systems.

Pre-requisites

- Basics of classification problems
- Understanding of perceptrons and activation functions (ReLU, sigmoid)
- Python programming with knowledge of libraries like `TensorFlow` or `Keras`

Algorithm

1. Import dataset and libraries.
2. Preprocess data and scale features.
3. Split dataset into training and test sets.
4. Create a simple neural network using Keras.
5. Compile and train the model.
6. Evaluate model performance on the test set.
7. Analyze results using confusion matrix and accuracy score.

Code

```
import pandas as pd importnumpy
as  np  fromsklearn.preprocessing
```

```
import StandardScaler
fromsklearn.metrics import
confusion_matrix,
accuracy_score
fromtensorflow.keras.models
import Sequential
fromtensorflow.keras.layers
import Dense

# Load dataset dataset =
pd.read_csv('Social_Network_Ads.csv') X =
dataset[['Age', 'EstimatedSalary']].values y =
dataset['Purchased'].values

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)

# Feature scaling sc =
StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

# Build neural network model
= Sequential()
model.add(Dense(units=6, activation='relu', input_dim=2)) # Input + Hidden layer
model.add(Dense(units=6, activation='relu')) # Hidden layer
model.add(Dense(units=1, activation='sigmoid')) # Output layer
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

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

# Evaluate on test data y_pred
= model.predict(X_test)
y_pred = (y_pred> 0.5)

# Confusion matrix cm =
confusion_matrix(y_test, y_pred) acc
= accuracy_score(y_test, y_pred)

print("Confusion Matrix:\n", cm)
print("Accuracy: {:.2f}%".format(acc * 100))
```

Sample output

```
Confusion Matrix:
 [[64 4]
[ 3 29]]
```

Accuracy: 93.00%

Steps to Execute

1. Ensure the dataset `Social_Network_Ads.csv` contains `Age`, `EstimatedSalary`, and `Purchased`.
2. Run the above Python code in a suitable environment (e.g., Jupyter Notebook).
3. Wait for training to complete and observe accuracy results.
4. Visualize results if needed.

Analysis

- The neural network trained well and achieved high accuracy.
- Adding more layers or tuning parameters may improve results.
- The sigmoid activation in the output layer helps in binary classification.

Applications

- Email spam detection
- Fraud detection
- Credit scoring
- Image and voice classification (for deeper networks)

Conclusion

- A basic ANN was successfully built and trained to perform binary classification.
- Students gained hands-on experience with neural network architecture, training, and evaluation.
- This lays the foundation for exploring more advanced deep learning models.

Course Outcome

CO5: Build deep learning neural network models

Result:

Thus the program of building a simple neural network mode has been executed successfully And output is verified.