COMMITTED TO EXCELLENCE IN EDUCATION

निर्मलस्नेह उत्तम सेवाधर्म

SHRI G.P.M. DEGREE COLLEGE OF SCIENCE & COMMERCE

SHRI G.P.M. DEGREE COLLEGE OF SCIENCE & COMMERCE.

(COMMITED TO EXCELLENCE IN EDUCATION)

CERTIFICATE

This is to certify that Mr/Ms.____

a student of TY.**BSC-CS** Roll no: _______ has completed the required number of practicals in the subject of ____

as prescribed by the UNIVERSITY OF MUMBAI under my supervision during the academic year 2024-2025.

………………..

Prof. Incharge

……………………

Principal

……………………

External Examiner

…...…..…………

Course Co-ordinator

Professor Name: Ms. Jayashree Patade	Class : TY.BSC-CS Semester : Sem V (2024-2025)
Course code : USCSP501	Subject: Artificial intelligence

6		Adaboost Ensemble Learning • Implement the Adaboost algorithm to create an ensemble of weak classifiers.	27-31	
7		Naive Baye’s Classifier • Implement the Naive Baye’s algorithm for classification.	32-36	

Theory -1

Graph Searching Algorithm

Experiment No.—1: Graph Searching Algorithm

Aim: Implement the Breadth first search algorithm to solve a given problem

Tools Used: Anaconda Navigator Jupyter Notebook

Theory: Breadth-First Search (BFS) is a fundamental graph traversal algorithm that explores vertices in the order of their distance from the source vertex. It is particularly useful for finding the shortest path in an unweighted graph.

Breadth first search algorithm is an algorithm used to find the shortest path from one node to another in a graph. It begins by searching through the nodes closest to the starting node, and then expands outward to the more distant nodes.

Breadth-First Search (BFS) is a fundamental algorithm in artificial intelligence and computer science that helps solve problems such as finding the shortest path, checking graph connectivity, and traversing all the vertices of a graph.

The main advantage of the breadth first search algorithm is that it is simple to implement and can be easily understood. Moreover, this algorithm can be used to find the shortest path between two nodes in a graph.

Here's how BFS works:

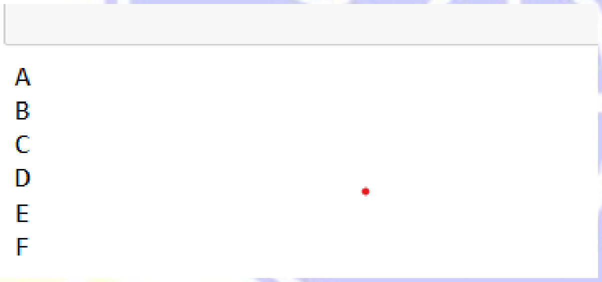
- Select a starting node: Choose a random node to start from, also known as the root or source node.
- Traverse the graph layer-wise: Visit all the nodes and their children nodes in the order they appear in the queue. The queue uses a First In First Out (FIFO) principle.
- Expand outward: Start with the nodes closest to the starting node and then move on to more distant nodes.

BFS is often used in tree traversal and network routing algorithms.

Code:

```
from collections import deque
# BFS function
def bfs(graph, start_node):
    visited = set()  # Set to keep track of visited nodes
    queue = deque([start_node])  # Initialize a queue with the start node
    while queue:
        node = queue.popleft()  # Pop the first node from the queue
                if node not in visited:
            print(node)  # Process the node (you can modify this as needed)
            visited.add(node)  # Mark the node as visited
            # Add all unvisited neighbors to the queue
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)

# Example graph (adjacency list representation)
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
'F': []
}

# Call BFS starting from node 'A'
bfs(graph, 'A')
```

Output:



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

Result and Discussion:

Breadth-First Search (BFS) is an algorithm used for traversing or searching tree or graph data structures. The key idea of BFS is to explore all nodes at the present depth before moving on to nodes at the next depth level. BFS ensures that the shortest path from the source node to a given target node is found in an unweighted graph.

Learning Objectives:

Implementing the Breadth-First Search (BFS) algorithm to solve a given problem involves several learning objectives that can be categorized into different areas such as understanding the algorithm itself, programming skills.

Course Outcomes:

The outcomes of graph searching algorithms are versatile and depend on the specific problem being addressed. BFS is particularly useful for finding the shortest path in unweighted graphs and exploring levels of node while DFS is often used for exploring deeper levels of a graph and applications like topological sorting and cycle detection.

Conclusion:

Viva Questions:

1.What is BFS?

2.What data structure used in BFS?

3.What are some application of BFS?

4.How does BFS work?

Theory -2

A* Search Algorithm

Experiment No.—2 : A* Search

Aim: Implement the A* Search algorithm for solving a pathfinding problem.

Tools Used: Anaconda Navigator Jupyter Notebook

Theory: The A* search algorithm is a popular and efficient pathfinding and graph traversal algorithm used in computer science and artificial intelligence.A Algorithm* is an informed search algorithm, meaning it uses additional information (heuristics) to guide its search.

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

It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem.

Another aspect that makes A* so powerful is its implementation of weighted graphs. A weighted graph uses numbers to represent the cost of taking each path or course of action. This means that the algorithms can take the path with the least cost, and find the best route in terms of distance and time.

A major drawback of the algorithm is its space and time complexity. It takes a large amount of space to store all possible paths and a lot of time to find them.

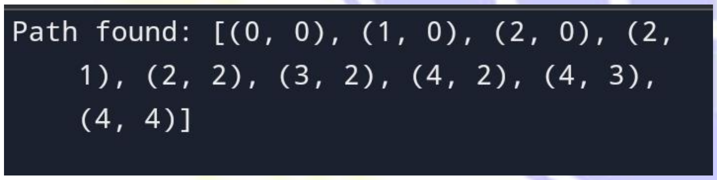
Code:

```
from heapq import heappop, heappush
class Node:
    def _init_(self, position, parent=None, g=0, h=0):
        self.position = position  # Node position (x, y)
        self.parent = parent      # Parent node
        self.g = g               # Cost from start to current node
        self.h = h               # Heuristic (estimated cost from current node to end)
        self.f = g + h           # Total cost (f = g + h)

    def _lt_(self, other):
        return self.f < other.f

def heuristic(a, b):
    """Use Manhattan distance as the heuristic for a grid-based pathfinding."""
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def astar_search(grid, start, end):
    open_list = []
    closed_list = set()

    start_node = Node(position=start, g=0, h=heuristic(start, end))
    heappush(open_list, start_node)
```

```
while open_list:
    current_node = heappop(open_list)
    closed_list.add(current_node.position)

    # Goal check
    if current_node.position == end:
        path = []
        while current_node:
            path.append(current_node.position)
            current_node = current_node.parent
        return path[::-1]  # Return reversed path

    # Explore neighbors (4 directions: up, down, left, right)
    neighbors = [
        (0, -1), (0, 1), (-1, 0), (1, 0)
    ]
    for dx, dy in neighbors:
        neighbor_pos = (current_node.position[0] + dx, current_node.position[1] + dy)
        # Check if neighbor is out of bounds or blocked
        if not (0 <= neighbor_pos[0] < len(grid) and 0 <= neighbor_pos[1] < len(grid[0])):
            continue
        if grid[neighbor_pos[0]][neighbor_pos[1]] == 1:
            continue
```

```
# Skip if neighbor is already in the closed list
            if neighbor_pos in closed_list:
                continue
            g = current_node.g + 1
            h = heuristic(neighbor_pos, end)
            neighbor_node = Node(position=neighbor_pos, parent=current_node, g=g, h=h)
            # Check if this neighbor is already in the open list with a lower f value
            in_open_list = False
            for open_node in open_list:
                if open_node.position == neighbor_pos and open_node.f <= neighbor_node.f:
                    in_open_list = True
                    break
            if not in_open_list:
                heappush(open_list, neighbor_node)

    return None  # No path found

# Example grid (0 = free space, 1 = obstacle)
grid = [
    [0, 1, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [1, 1, 0, 1, 0],
    [0, 0, 0, 0, 0]]

# Starting point and ending point

start = (0, 0)

end = (4, 4)

# Perform A* search

path = astar_search(grid, start, end)

if path:

    print("Path found:", path)

else:

    print("No path found")
```

Output:



```
Path found: [(0, 0), (1, 0), (2, 0), (2,
    1), (2, 2), (3, 2), (4, 2), (4, 3),
    (4, 4)]
```

Result and Discussion:

The *A* Search Algorithm* is a widely used pathfinding and graph traversal algorithm that aims to find the shortest path from a start node to a target node while efficiently using both the graph's structure and heuristic information.

Learning Objectives:

Understanding the fundamentals of A* Search , Apply the A search algorithm , Evaluate heuristic functions

Course Outcomes:

Understand pathfinding and Search Algorithms , Master the A Algorithm* , Implement A in a Programming language* , Design and evaluate Heuristics , Solve Real-World Pathfinding Problems , Optimize and Debug A Implementations* , Integrate A into Larger Systems* , Reflect on Learning and Applications , Collaborate on Problem-solving , Stay Informed on Advancements.

Conclusion:

Viva Questions:

1.What is A* search Algorithm?

2.How does algorithm work?

3.What are application of A* search algorithm?

4.What are the component of A* search Algorithm?

Theory-3

Decision Tree Learning

Experiment No.—3 : Decision Tree Learning

Aim: Implement the Decision Tree Learning algorithm to build a decision tree for a given dataset

Tools Used: Anaconda Navigator Jupyter Notebook

Theory: The theory behind Decision Tree Learning revolves around constructing a predictive model in the form of a tree structure that recursively partitions the input space into regions, where each region corresponds to a specific class label (in classification tasks) or a predicted value (in regression tasks).

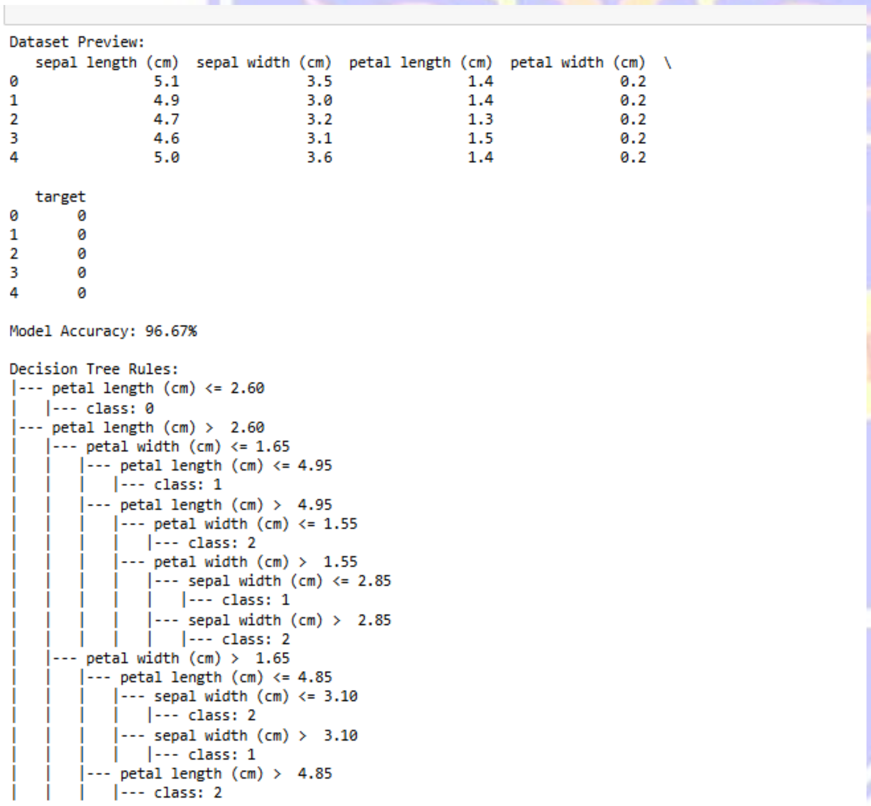
- Decision Tree is a Supervised learning technique that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems. It is a tree-structured classifier, where internal nodes represent the features of a dataset, branches represent the decision rules and each leaf node represents the outcome. o In a Decision tree, there are two nodes, which are the Decision Node and Leaf Node. Decision nodes are used to make any decision and have multiple branches, whereas Leaf nodes are the output of those decisions and do not contain any further branches.
- The decisions or the test are performed on the basis of features of the given dataset.
- It is a graphical representation for getting all the possible solutions to a problem/decision based on given conditions.
- It is called a decision tree because, similar to a tree, it starts with the root node, which expands on further branches and constructs a tree-like structure. o In order to build a tree, we use the CART algorithm, which stands for Classification and Regression Tree algorithm.
- A decision tree simply asks a question, and based on the answer (Yes/No).

Code:

```
# Import necessary libraries
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn import metrics
from sklearn.tree import export_text

# Load the Iris dataset
iris = load_iris()

# Create a DataFrame for the features
df = pd.DataFrame(iris.data, columns=iris.feature_names)

# Add the target variable to the DataFrame
df['target'] = iris.target

# Display the first 5 rows of the dataset
print("Dataset Preview:")
print(df.head())

# Define the features (X) and the target variable (y)
```

```
X = df.drop('target', axis=1)  # Features (all columns except 'target')

y = df['target']  # Target variable (the 'target' column)
# Split the dataset into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)

# Initialize the Decision Tree Classifier
clf = DecisionTreeClassifier()

# Train the classifier on the training data
clf = clf.fit(X_train, y_train)

# Predict the target variable on the testing data
y_pred = clf.predict(X_test)

# Evaluate the accuracy of the model
accuracy = metrics.accuracy_score(y_test, y_pred)
print("\nModel Accuracy: {:.2f}%".format(accuracy * 100))

# Print the decision tree rules
tree_rules = export_text(clf, feature_names=list(X.columns))
print("\nDecision Tree Rules:")
print(tree_rules)
```

Output:



```
Dataset Preview:
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

   target
0       0
1       0
2       0
3       0
4       0

Model Accuracy: 96.67%

Decision Tree Rules:
|--- petal length (cm) <= 2.60
|   |--- class: 0
|--- petal length (cm) >  2.60
|   |--- petal width (cm) <= 1.65
|   |   |--- petal length (cm) <= 4.95
|   |   |   |--- class: 1
|   |   |--- petal length (cm) >  4.95
|   |   |   |--- petal width (cm) <= 1.55
|   |   |   |   |--- class: 2
|   |   |   |--- petal width (cm) >  1.55
|   |   |   |   |--- sepal width (cm) <= 2.85
|   |   |   |   |   |--- class: 1
|   |   |   |   |--- sepal width (cm) >  2.85
|   |   |   |   |   |--- class: 2
|   |--- petal width (cm) >  1.65
|   |   |--- petal length (cm) <= 4.85
|   |   |   |--- sepal width (cm) <= 3.10
|   |   |   |   |--- class: 2
|   |   |   |--- sepal width (cm) >  3.10
|   |   |   |   |--- class: 1
|   |   |--- petal length (cm) >  4.85
|   |   |   |--- class: 2
```

Results and Discussion:

Decision Tree Learning is a popular machine learning algorithm used for both classification and regression tasks. The model represents decisions and their possible consequences in a tree-like structure. It splits the data into subsets based on the most significant attribute, aiming to create pure subsets

Learning Objectives:

Decision Tree Learning aims to achieve several objectives, primarily revolving around the creation of an effective decision tree model that accurately predicts outcomes based on input data.

Course Outcomes:

Understanding Foundations, Algorithmic Knowledge, Data Handling and Preprocessing, Model Evaluation, Interpretability and Visualization, Advanced Topics, Application and Case Studies, Ethical and Practical Considerations, Hands-on Experience, Problem Solving and Critical Thinking.

Conclusion:

Viva Questions:

1.What is decision tree learning?

2.What are the advantages of using decision tree learning?

3. What are the disadvantages of using decision tree learning?

4.How does decision tree learning work?

Theory-4

Feed Forward Backpropagation Neural Network

Experiment No.—4 : Feed Forward Backpropagation Neural Network

Aim: Implement the Feed Forward Backpropagation algorithm to train a neural network.

Tools Used: Anaconda Navigator Jupyter Notebook

Theory: Feedforward backpropagation neural networks consist of an input layer, hidden layers, and an output layer. During forward propagation, inputs are transformed through weighted sums and activation functions to produce outputs. The performance is evaluated using a loss function. Backpropagation calculates gradients to update weights efficiently, optimizing the network using methods like Stochastic Gradient Descent or Adam. Regularization techniques, such as dropout, help prevent overfitting, while hyperparameter tuning ensures optimal performance. This framework enables the network to learn complex patterns and make accurate predictions across various applications

A Feedforward Neural Network (FNN) is a type of artificial neural network where connections between the nodes do not form cycles. This characteristic differentiates it from recurrent neural networks (RNNs). The network consists of an input layer, one or more hidden layers, and an output layer. Information flows in one direction—from input to output—hence the name "feedforward."

Structure of a Feedforward Neural Network

1. Input Layer: The input layer consists of neurons that receive the input data. Each neuron in the input layer represents a feature of the input data.

2. Hidden Layers: One or more hidden layers are placed between the input and output layers. These layers are responsible for learning the complex patterns in the data. Each neuron in a hidden layer applies a weighted sum of inputs followed by a non-linear activation function.

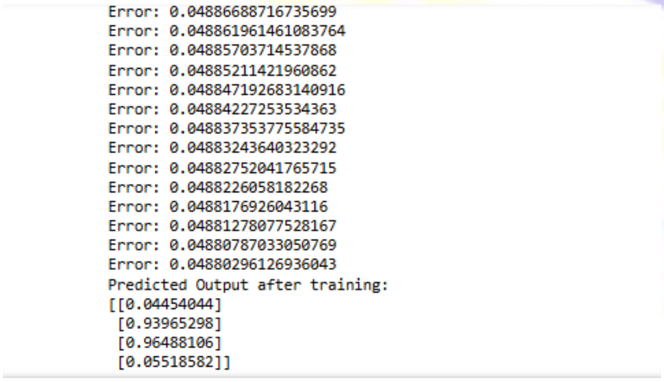
3. Output Layer: The output layer provides the final output of the network. The number of neurons in this layer corresponds to the number of classes in a classification problem or the number of outputs in a regression problem.

Code:

```
import numpy as np


# Activation function: Sigmoid and its derivative

def sigmoid(x):

    return 1 / (1 + np.exp(-x))


def sigmoid_derivative(x):

    return x * (1 - x)


# Neural Network class

class NeuralNetwork:

    def __init__(self, input_size, hidden_size, output_size):

        # Initialize weights and biases randomly

        self.input_size = input_size

        self.hidden_size = hidden_size

        self.output_size = output_size


        self.weights_input_hidden = np.random.rand(input_size, hidden_size)

        self.weights_hidden_output = np.random.rand(hidden_size, output_size)


        self.bias_hidden = np.random.rand(1, hidden_size)

        self.bias_output = np.random.rand(1, output_size)
```

```
# Feedforward function
    def feedforward(self, X):
        # Input to hidden layer
        self.hidden_layer_input = np.dot(X, self.weights_input_hidden) + self.bias_hidden
        self.hidden_layer_output = sigmoid(self.hidden_layer_input)

        # Hidden layer to output layer
        self.output_layer_input = np.dot(self.hidden_layer_output, self.weights_hidden_output) + self.bias_output
        output = sigmoid(self.output_layer_input)
        return output

    # Backpropagation function
    def backpropagate(self, X, y, output, learning_rate):
        # Compute the error
        error = y - output
        print(f"Error: {np.mean(np.abs(error))}")

        # Compute output layer delta
        output_delta = error * sigmoid_derivative(output)

        # Compute hidden layer delta
        hidden_layer_error = output_delta.dot(self.weights_hidden_output.T)
        hidden_delta = hidden_layer_error * sigmoid_derivative(self.hidden_layer_output)
        # Update weights and biases using the deltas
        self.weights_hidden_output += self.hidden_layer_output.T.dot(output_delta) * learning_rate
        self.weights_input_hidden += X.T.dot(hidden_delta) * learning_rate
        self.bias_output += np.sum(output_delta, axis=0, keepdims=True) * learning_rate
        self.bias_hidden += np.sum(hidden_delta, axis=0, keepdims=True) * learning_rate

    # Training function
    def train(self, X, y, epochs=10000, learning_rate=0.1):
        for epoch in range(epochs):
          # Step 1: Feedforward
          output = self.feedforward(X)

          # Step 2: Backpropagation
          self.backpropagate(X, y, output, learning_rate)

          # Print error every 1000 epochs
          if epoch % 1000 == 0:
            print(f"Epoch {epoch}, Error: {np.mean(np.abs(y - output))}")

# Example usage
if __name__ == "__main__":
    # Input dataset (X) - 4 samples, 3 features
    X = np.array([[0, 0, 1],
            [0, 1, 1],
            [1, 0, 1],
            [1, 1, 1]])

    # Output labels (y) - 4 samples, 1 target value
    y = np.array([[0], [1], [1], [0]])
  # Initialize the Neural Network
    nn = NeuralNetwork(input_size=3, hidden_size=4, output_size=1)
    # Train the neural network
    nn.train(X, y, epochs=10000, learning_rate=0.1)
    # Test the trained network with the same input
    output = nn.feedforward(X)
    print("Predicted Output after training:")
    print(output)
```

Output:



```
Error: 0.04886688716735699
Error: 0.048861961461083764
Error: 0.04885703714537868
Error: 0.04885211421960862
Error: 0.048847192683140916
Error: 0.04884227253534363
Error: 0.048837353775584735
Error: 0.04883243640323292
Error: 0.04882752041765715
Error: 0.0488226058182268
Error: 0.0488176926043116
Error: 0.04881278077528167
Error: 0.04880787033050769
Error: 0.04880296126936043
Predicted Output after training:
[[0.04454044]
 [0.93965298]
 [0.96488106]
 [0.05518582]]
```

Results and Discussion:

The Feed Forward Backpropagation Neural Network is a supervised learning algorithm used for tasks such as classification and regression. It consists of an input layer, one or more hidden layers, and an output layer.

Learning Objectives:

Learn the structure of feedforward neural networks, including the roles of input, hidden, and output layers.Gain skills in computing outputs through forward propagation, applying activation functions to neurons.

Course Outcomes:

Understand Network Architecture, Implement Forward Propagation, Calculate Loss Functions, Apply Backpropagation Algorithm, Optimize Weights Using Techniques, Evaluate Model Performance, Implement Regularization Methods, Tune Hyperparameters Effectively, Develop Real-World Applications.

Conclusion:

Viva Questions:

1.What is the neural network?

2.What are the maim component of neural network?

3.What is back propagations?

4.What are the common applications of neural network?

Theory-5

Support vector machine (SVM)

Experiment No.—5: Support vector machine(SVM)

Aim: Implement the SVM algorithm for binary Classification

Tools Used: Anaconda navigator ,Jupiter notebook ,

Theory: Support Vector Machine (SVM) is a supervised machine learning algorithm used for classification and regression tasks. The primary objective of SVM is to find the optimal hyperplane that separates the data points of different classes with the maximum margin. This hyperplane acts as a decision boundary, and the goal is to maximize the distance between the hyperplane and the closest data points from each class, known as support vectors. By maximizing this margin, SVM aims to enhance the model's ability to generalize to unseen data, thereby reducing overfitting.

SVM is a supervised learning algorithm primarily used for classification tasks. It aims to find a hyperplane in an N-dimensional space (N = number of features) that distinctly classifies the data points into two classes.

Key Concepts in SVM:

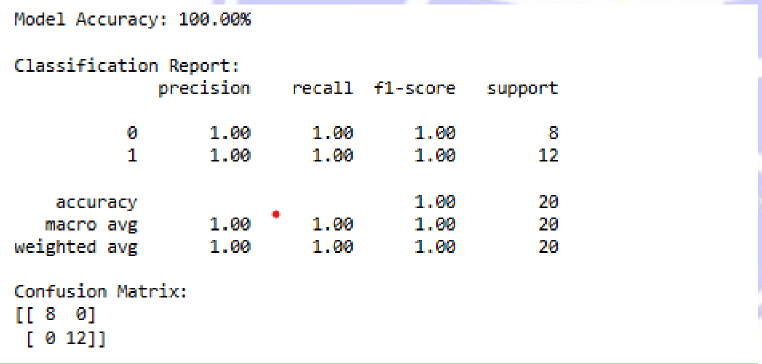
1. **Support Vectors**:
   - Support vectors are the data points that are closest to the hyperplane. They are the most critical elements of the dataset because they define the position and orientation of the hyperplane.
   - The model's decision boundary depends on these support vectors, and removing or changing them would affect the classification outcome.
2. **Margin**:
   - The margin is the distance between the hyperplane and the nearest support vectors. A larger margin indicates better separation between the classes, which generally leads to better generalization on unseen data.
3. **Linear SVM**:
   - In the case of a linearly separable dataset, SVM finds a linear hyperplane to classify the data point

Code:

```
# Import necessary libraries
import numpy as np
import pandas as pd
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Load the Iris dataset
iris = datasets.load_iris()

# Create a DataFrame for the features
df = pd.DataFrame(iris.data, columns=iris.feature_names)

# Add the target variable to the DataFrame
df['target'] = iris.target

df = df[df['target'] != 2]  # Remove class 2

# Define the features (X) and the target variable (y)
```

```
X = df.drop('target', axis=1)  # Features (all columns except 'target')
y = df['target']  # Target variable (the 'target' column)

# Split the dataset into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)

# Standardize the features (scaling them to have zero mean and unit variance)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Initialize the Support Vector Machine classifier
svm = SVC(kernel='linear')  # Using a linear kernel for binary classification

# Train the SVM classifier on the training data
svm.fit(X_train, y_train)

# Predict the target variable on the testing data
y_pred = svm.predict(X_test)

# Evaluate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print("Model Accuracy: {:.2f}%".format(accuracy * 100)
# Print the classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

# Print the confusion matrix
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

Output:



```
Model Accuracy: 100.00%

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         8
           1       1.00      1.00      1.00        12

    accuracy                           1.00        20
   macro avg       1.00      1.00      1.00        20
weighted avg       1.00      1.00      1.00        20

Confusion Matrix:
[[ 8  0]
 [ 0 12]]
```

Results and Discussion:

The Support Vector Machine (SVM) algorithm is a powerful supervised learning model used for classification and regression tasks. It works by finding the hyperplane that best separates the data points of different classes in a high-dimensional space. SVM is particularly effective in high-dimensional spaces and when the number of dimensions exceeds the number of samples.

Learning Objectives:

Maximizing the Margin: SVM aims to find the hyperplane that best separates the data into different classes with the maximum margin. This means that the distance between the hyperplane and the nearest data points of any class (called support vectors) is maximized, which helps in improving the model's generalization capabilities.

Course Outcomes:

The main objective of the SVM algorithm is to find the optimal hyperplane in an N-dimensional space that can separate the data points in different classes in the feature space. The hyperplane tries that the margin between the closest points of different classes should be as maximum as possible.

Conclusion:

Viva Question:

1.What is the support vector machine(SVM)?

2.What are support vector?

3.What are the advantages of SVM?

Theory-6

Adaboost Ensemble Learning

Experiment No.—6: Adaboost Ensemble Learning

Aim: Implement the Adaboost algorithm to create an ensemble of weak classifiers

Tools Used: Anaconda Navigator Jupyter Notebook

Theory: Adaboost's success stems from its ability to leverage the wisdom of multiple weak learners, each specializing in different aspects of the data.

The algorithm ensures that each subsequent weak learner focuses more on instances that are difficult to classify correctly, thus iteratively improving the overall performance.

In summary, Adaboost is a powerful ensemble learning method that enhances the predictive accuracy of weak learners by iteratively adjusting their focus on misclassified instances and combining their predictions using a weighted voting scheme. Its theoretical foundation lies in leveraging multiple weak learners to create a strong ensemble that performs well on complex classification tasks.

Weak Learners: Adaboost works by combining multiple weak learners (typically decision trees, also known as stumps or shallow trees). A weak learner is a model that performs slightly better than random guessing on a classification problem.

Course Outcomes: Adaboost tends to reduce bias and variance compared to individual weak learners, leading to improved generalization performance.

It is less prone to overfitting because it focuses training on instances that are harder to classify correctly.

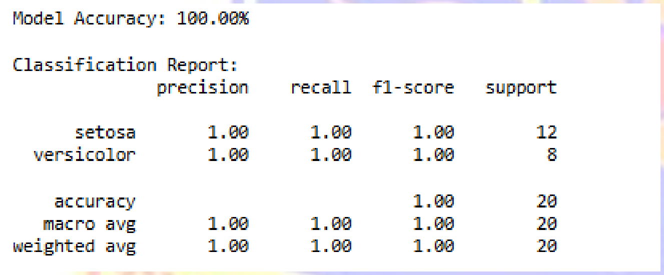
It can adapt to complex datasets and capture intricate patterns by combining multiple models.

Code:

```
# Import necessary libraries
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.metrics import accuracy_score, classification_report

# Load the Iris dataset
iris = load_iris()

# Create a DataFrame for the features
df = pd.DataFrame(iris.data, columns=iris.feature_names)

# Add the target variable to the DataFrame
df['target'] = iris.target

# We will filter the dataset for binary classification (e.g., classes 0 and 1)
df_binary = df[df['target'].isin([0, 1])]
```

```
# Define the features (X) and target (y)

X = df_binary.drop('target', axis=1)  # Features (all columns except 'target')

y = df_binary['target']  # Target variable (the 'target' column)

# Split the dataset into training and testing sets (80% training, 20% testing)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize a Decision Tree classifier with a max depth of 1 (weak classifier)

weak_classifier = DecisionTreeClassifier(max_depth=1)

# Initialize the AdaBoostClassifier with the weak classifier

ada_boost = AdaBoostClassifier(base_estimator=weak_classifier, n_estimators=50, learning_rate=1.0,
random_state=42)

# Train the AdaBoost classifier on the training data

ada_boost.fit(X_train, y_train)

# Predict the target variable on the test data

y_pred = ada_boost.predict(X_test)


# Evaluate the accuracy of the model

accuracy = accuracy_score(y_test, y_pred)

print("Model Accuracy: {:.2f}%".format(accuracy * 100))


# Detailed classification report

print("\nClassification Report:")

print(classification_report(y_test, y_pred, target_names=iris.target_names[:2]))
```

Output:



```
Model Accuracy: 100.00%

Classification Report:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        12
  versicolor       1.00      1.00      1.00         8

    accuracy                           1.00        20
   macro avg       1.00      1.00      1.00        20
weighted avg       1.00      1.00      1.00        20
```

Result and Discussion:

AdaBoost (Adaptive Boosting) is a powerful ensemble learning technique that combines multiple weak classifiers to create a strong classifier. It works by adjusting the weights of incorrectly classified instances, focusing more on the hard-to-classify samples during subsequent iterations.

Learning Objectives:

Adaboost is a boosting algorithm, which means it focuses on sequentially training weak learners.Adaboost aims to create an ensemble model by combining multiple weak learners.

Course Outcomes:

Explain the concept of ensemble methods and how they improve model performance.Discuss the differences between boosting, bagging, and stacking. Use libraries like scikit-learn to implement AdaBoost in Python, configuring the algorithm's parameters and selecting appropriate base learners.Analyze model performance using metrics such as accuracy, precision, recall, and F1-score.

Conclusion:

Viva Questions:

1.What is adaboost algorithm?

2.How does adaboost algorithm works?

3.What is weak classifier?

4. What are the advantages of using adaboost?

Theory-7

Naive Baye's Classifier

Experiment No.—7: Naïve Baye's Classifier

Aim: Implement the Naive Bayes' algorithm for classification

Tools Used: Anaconda Navigator Jupyter Notebook

Theory:

Naïve Bayes algorithm is a supervised learning algorithm, which is based on Bayes theorem and used for solving classification problems. It is mainly used in text classification that includes a high-dimensional training dataset. o Naïve Bayes Classifier is one of the simple and most effective Classification algorithms which helps in building the fast machine learning models that can make quick predictions. It is a probabilistic classifier, which means it predicts on the basis of the probability of an object. Some popular examples of Naïve Bayes Algorithm are spam filtration, Sentimental analysis, and classifying articles.

Why is it called Naïve Bayes?

The Naïve Bayes algorithm is comprised of two words Naïve and Bayes, Which can be described as:

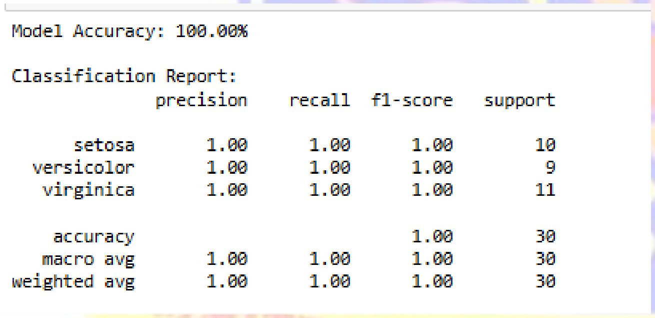
- Naïve: It is called Naïve because it assumes that the occurrence of a certain feature is independent of the occurrence of other features. Such as if the fruit is identified on the bases of color, shape, and taste, then red, spherical, and sweet fruit is recognized as an apple. Hence each feature individually contributes to identify that it is an apple without depending on each other.
- Bayes: It is called Bayes because it depends on the principle of Bayes' Theorem.

Code:

```
# Import necessary libraries
import numpy as np
import pandas as pd
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, classification_report

# Load the Iris dataset
iris = load_iris()

# Create a DataFrame for the features
df = pd.DataFrame(iris.data, columns=iris.feature_names)

# Add the target variable to the DataFrame
df['target'] = iris.target

# Define the features (X) and target (y)
X = df.drop('target', axis=1)  # Features (all columns except 'target')
y = df['target']  # Target variable (the 'target' column)
```

```
# Split the dataset into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize the Gaussian Naive Bayes classifier
gnb = GaussianNB()

# Train the classifier on the training data
gnb.fit(X_train, y_train)

# Predict the target variable on the test data
y_pred = gnb.predict(X_test)

# Evaluate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print("Model Accuracy: {:.2f}%".format(accuracy * 100))

# Detailed classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=iris.target_names))
```

Output:



```
Model Accuracy: 100.00%

Classification Report:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        10
  versicolor       1.00      1.00      1.00         9
   virginica       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

Result and Discussion:

The Naive Bayes Classifier is a probabilistic classifier based on applying Bayes' theorem with strong (naive) independence assumptions between the features. It is particularly effective for classification tasks involving text data, spam detection, and other categorical data applications.

Learning Objectives:

Naive Bayes algorithm is rooted in Bayes' theorem, which describes the probability of an event based on prior knowledge of conditions that might be related to the event.Learn the assumptions underlying the Naive Bayes algorithm, particularly the "naive" assumption of independence between features.

Course Outcomes:

Upon completing a course focused on the Naive Bayes classifier, students will acquire a comprehensive understanding of its underlying principles, including Bayes' theorem and the independence assumption between features. They will gain practical skills in implementing the Naive Bayes algorithm using popular programming languages and libraries, enabling them to preprocess datasets

Conclusion:

Viva Questions:

1.What is Naïve Baye’s Classifier?

2.How does Naïve Baye’s Classifier works?

3.What is Naïve Baye’s classifier?

4. What are the advantages of using Naïve Baye’s Classifier?