**Practical No: 01**

**Aim: Breadth First Search & Iterative Depth First Search .**

1. Implement the Breadth First Search algorithm to solve a given problem.
2. Implement the Iterative Depth First Search algorithm to solve the same problem.
3. Compare the performance and efficiency of both algorithms.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

1. **Breadth First Search algorithm:**

**Python Code:**

from collections import deque

def bfs\_shortest\_path(graph, start, goal):

queue = deque([[start]])

visited = set()

while queue:

path = queue.popleft()

node = path[-1]

if node == goal:

return path

if node not in visited:

visited.add(node)

for neighbor in graph.get(node, []):

new\_path = list(path)

new\_path.append(neighbor)

queue.append(new\_path)

return None

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

shortest\_path = bfs\_shortest\_path(graph, 'A', 'F')

print("Shortest Path from 'A' to 'F':", shortest\_path)

**Output:**

****

1. **Iterative Depth First Search algorithm:**

**Python Code:**

def iterative\_dfs(graph, start, goal):

stack = [[start]]

visited = set()

while stack:

path = stack.pop()

node = path[-1]

if node == goal:

return path

if node not in visited:

visited.add(node)

for neighbor in reversed(graph.get(node, [])):

new\_path = list(path)

new\_path.append(neighbor)

stack.append(new\_path)

return None

graph = {

'A': ['B', 'C'],

'B': ['D', 'E'],

'C': ['F'],

'D': [],

'E': ['F'],

'F': []

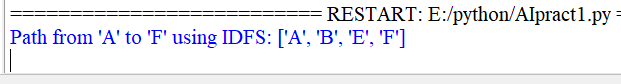
}

# Find path from A to F

dfs\_path = iterative\_dfs(graph, 'A', 'F')

print("Path from 'A' to 'F' using IDFS:", dfs\_path)

**Output:**



1. **Comparing the performance and efficiency of the two algorithms.**

In the comparison of efficiency and performance between Breadth-First Search (BFS) and Iterative Deepening Depth-First Search (IDDFS) algorithms, let's analyze how they perform based on the given example:

**Efficiency:**

* **Time Complexity:**
  + **BFS:** The time complexity of BFS is generally )O(b^d), where b is the branching factor, and d is the depth of the shallowest solution. In the worst-case scenario, BFS may need to explore all nodes up to a certain depth, making the time complexity high, especially in graphs with large branching factors.
  + **IDDFS:** IDDFS performs a series of depth-limited DFS searches, incrementing the depth limit until the goal is found. It can be more efficient than BFS in average-case scenarios as it avoids unnecessary node explorations. However, in the worst case, where the goal node is at the maximum depth, the time complexity can approach O(b^d), similar to BFS.
* **Space Complexity:**
  + **BFS:** BFS has a high space complexity as it needs to store all nodes at a particular depth in memory, which can grow significantly with the branching factor. In the worst case, the space complexity is O(b^d).
  + **IDDFS:** IDDFS is more memory-efficient since it only keeps track of the current path being explored. The space complexity is O(d), where ddd is the maximum depth of the search.

**Performance:**

* **Search Behavior:**
  + **BFS:** BFS explores all nodes level by level, ensuring that the optimal path is found when the edge costs are uniform. However, it can expand a large number of nodes, making it less efficient in terms of memory and time, especially in graphs with a high branching factor.
  + **IDDFS:** IDDFS combines the depth-limited nature of DFS with the completeness of BFS, allowing it to find the optimal solution while maintaining lower memory consumption. However, if the goal node is located deep within the graph or the branching factor is high, IDDFS may still end up exploring a large number of nodes.

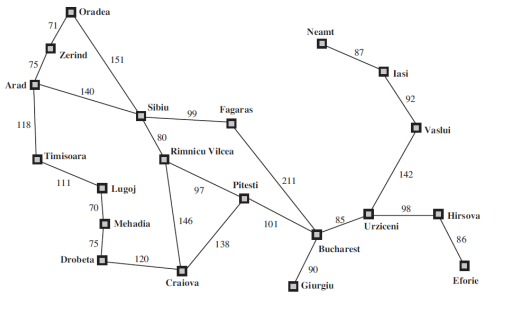
**Practical No: 02**

**Aim: A\* Search and Recursive Best-First Search**

1. Implement the A\* Search algorithm for solving a pathfinding problem.
2. Implement the Recursive Best-First Search algorithm for the same problem.
3. Compare the performance and effectiveness of both algorithms

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

1. **Implement the A\* Search algorithm for solving a pathfinding problem.**



**Python Code:**

import heapq

romania\_map = {

'Arad': {'Zerind': 75, 'Timisoara': 118, 'Sibiu': 140},

'Zerind': {'Arad': 75, 'Oradea': 71},

'Timisoara': {'Arad': 118, 'Lugoj': 111},

'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},

'Oradea': {'Zerind': 71, 'Sibiu': 151},

'Lugoj': {'Timisoara': 111, 'Mehadia': 70},

'Fagaras': {'Sibiu': 99, 'Bucharest': 211},

'Rimnicu Vilcea': {'Sibiu': 80, 'Pitesti': 97, 'Craiova': 146},

'Mehadia': {'Lugoj': 70, 'Drobeta': 75},

'Drobeta': {'Mehadia': 75, 'Craiova': 120},

'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},

'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},

'Bucharest': {'Fagaras': 211, 'Pitesti': 101, 'Giurgiu': 90, 'Urziceni': 85},

'Giurgiu': {'Bucharest': 90},

'Urziceni': {'Bucharest': 85, 'Hirsova': 98, 'Vaslui': 142},

'Hirsova': {'Urziceni': 98, 'Eforie': 86},

'Eforie': {'Hirsova': 86},

'Vaslui': {'Urziceni': 142, 'Iasi': 92},

'Iasi': {'Vaslui': 92, 'Neamt': 87},

'Neamt': {'Iasi': 87}

}

class Node:

def \_\_init\_\_(self, city, cost, parent=None):

self.city = city

self.cost = cost

self.parent = parent

def \_\_lt\_\_(self, other):

return self.cost < other.cost

def astar\_search(graph, start, goal):

open\_list = []

closed\_set = set()

heapq.heappush(open\_list, start)

while open\_list:

current\_node = heapq.heappop(open\_list)

if current\_node.city == goal.city:

return construct\_path(current\_node)

closed\_set.add(current\_node.city)

for neighbor, distance in graph[current\_node.city].items():

if neighbor not in closed\_set:

new\_node = Node(neighbor, current\_node.cost + distance, current\_node)

heapq.heappush(open\_list, new\_node)

return None

def construct\_path(node):

path = []

while node:

path.append(node.city)

node = node.parent

return path[::-1]

start\_node = Node('Arad', 0)

goal\_node = Node('Bucharest', 0)

path = astar\_search(romania\_map, start\_node, goal\_node)

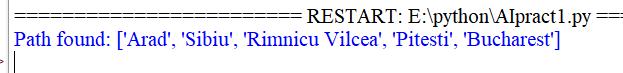
if path:

print("Path found:", path)

else:

print("No path found")

**Output:**



1. **Python code for Recursive Best-First Search algorithm**

**Python Code:**

# Define the graph and heuristic

graph = {

'A': {'B': 1, 'C': 3},

'B': {'A': 1, 'D': 1, 'E': 5},

'C': {'A': 3, 'F': 4},

'D': {'B': 1, 'E': 2, 'G': 7},

'E': {'B': 5, 'D': 2, 'G': 2},

'F': {'C': 4, 'G': 1},

'G': {'D': 7, 'E': 2, 'F': 1}

}

heuristic = {'A': 7, 'B': 6, 'C': 2, 'D': 3, 'E': 1, 'F': 1, 'G': 0}

# RBFS Algorithm

def rbfs(graph, node, goal, heuristic, path, cost\_so\_far, f\_limit):

if node == goal:

return path, cost\_so\_far

neighbors = [(cost\_so\_far + graph[node][neighbor] + heuristic[neighbor], neighbor)

for neighbor in graph.get(node, {})]

if not neighbors:

return None, float('inf')

while neighbors:

neighbors.sort()

best\_f, best\_node = neighbors[0]

if best\_f > f\_limit:

return None, best\_f

alternative = neighbors[1][0] if len(neighbors) > 1 else float('inf')

result, best\_f = rbfs(graph, best\_node, goal, heuristic, path + [best\_node],

cost\_so\_far + graph[node][best\_node], min(f\_limit, alternative))

if result:

return result, best\_f

neighbors[0] = (best\_f, best\_node)

return None, float('inf')

def recursive\_best\_first\_search(graph, start, goal, heuristic):

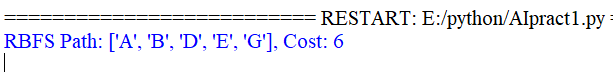
return rbfs(graph, start, goal, heuristic, [start], 0, float('inf'))

# Execute RBFS

path, cost = recursive\_best\_first\_search(graph, 'A', 'G', heuristic)

print(f"RBFS Path: {path}, Cost: {cost}")

**output:**



1. **Comparing the performance and efficiency of the two algorithms.**

In the comparison of efficiency and performance between the A\* and RBFS algorithms, we can consider factors such as time complexity, space complexity, and search behavior as follows

**Efficiency:**

**Time Complexity:**

* **A\***: The time complexity of A\* is **O(b^d)**, where b is the branching factor and d is the depth of the shallowest solution. A\* expands nodes based on their **f(n)=g(n)+h(n)**, where g(n) is the cost to reach node n and h(n) is the heuristic estimate of the cost from n to the goal. When the heuristic is admissible (never overestimates), A\* efficiently explores fewer nodes than uninformed search algorithms. In the worst case, its time complexity is still exponential, especially if the heuristic is weak or inconsistent.
* **RBFS**: Recursive Best-First Search (RBFS) has similar time complexity to A\*, but it backtracks whenever necessary to conserve memory. The time complexity depends on how often it revisits nodes, and in the worst case, it may end up re-exploring many nodes. This can result in a time complexity of O(b^d), comparable to A\*, but with more potential node revisits due to its recursive, memory-conserving nature.

**Space Complexity:**

* **A\***: A\* maintains both an open list (nodes to be explored) and a closed list (nodes already explored), which results in a high space complexity of O(b^d) in the worst case. The memory requirement grows as the search space increases, especially in problems with high branching factors or deep solutions.
* **RBFS**: RBFS is much more space-efficient compared to A\*. It only keeps track of the current path and the best alternative path, resulting in a space complexity of O(d), where d is the depth of the solution. This makes RBFS suitable for memory-constrained environments, but at the cost of potentially re-exploring paths.

**Performance:**

**Search Behavior:**

* **A\***: A\* is complete and optimal when using an admissible and consistent heuristic, meaning it guarantees finding the optimal solution. It expands nodes based on their f-value and prioritizes the most promising paths. However, A\* requires significant memory to store its open and closed lists, which can make it inefficient in large search spaces despite its optimality.
* **RBFS**: RBFS is designed to conserve memory by exploring paths recursively and backtracking when necessary. While it performs well in terms of space efficiency, it may revisit and re-explore nodes, especially in cases where the heuristic is weak or inconsistent. RBFS is not guaranteed to find the optimal path, but it can handle larger problems with limited memory.

**Practical No: 03**

**Aim: Decision Tree Learning**

• Implement the Decision Tree Learning algorithm to build a decision tree for a given dataset.

• Evaluate the accuracy and effectiveness of the decision tree on test data.

• Visualize and interpret the generated decision tree.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |
| --- | --- |
| **DataSet** | We download the Iris data set and use it for the present case. As we are performing the practical on Google Colab, we upload the dataset as Iris.csv |

First install the requirements

**pip install numpy pandas scikit-learn matplotlib**

**Python Code:**

import pandas as pd

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.tree import DecisionTreeClassifier, export\_text

from sklearn.metrics import accuracy\_score

import matplotlib.pyplot as plt

from sklearn.tree import plot\_tree

data = load\_iris() # Reading the Iris.csv file

X = data.data # Extracting Attributes / Features

y = data.target # Extracting Target / Class Labels

df = pd.DataFrame(data=X, columns=data.feature\_names)

df['species'] = y

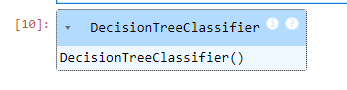
# Creating Train and Test datasets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

clf = DecisionTreeClassifier() # Initialize the model

clf.fit(X\_train, y\_train) # Train the model

**Output:**



# Predict Accuracy Score

y\_pred = clf.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Accuracy of the Decision Tree: {accuracy:.2f}")

**Output:**



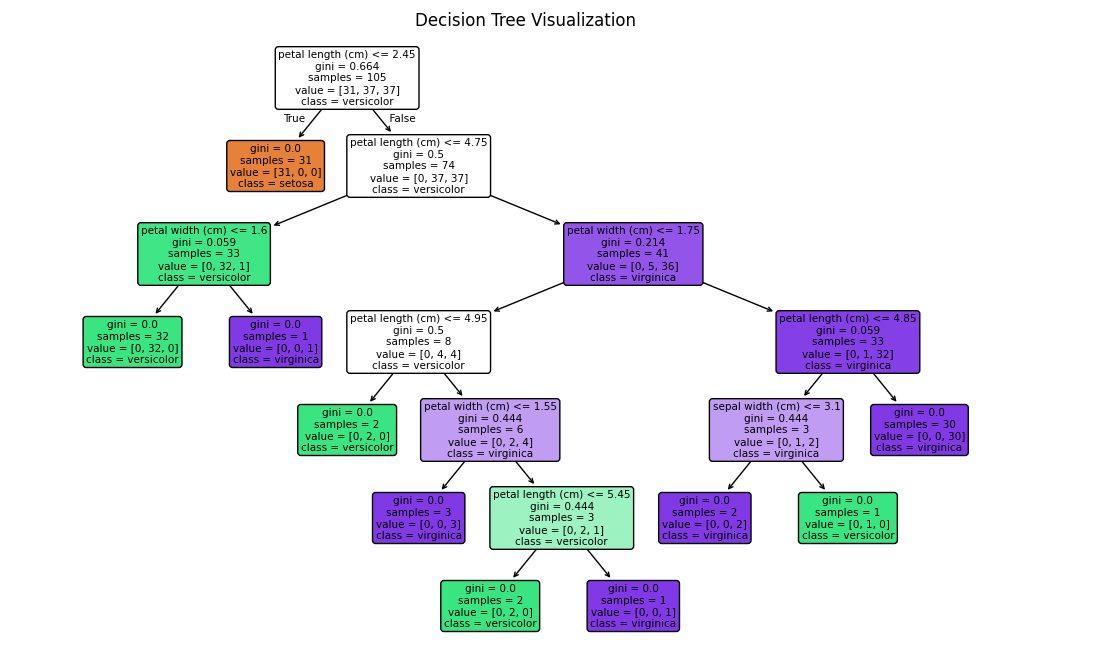
plt.figure(figsize=(12, 8))

plot\_tree(clf, filled=True, feature\_names=data.feature\_names, class\_names=data.target\_names, rounded=True)

plt.title("Decision Tree Visualization")

plt.show()

**Output:**



**Practical No : 04**

**Aim:** **Feed Forward Backpropagation Neural Network**

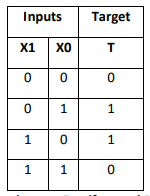
1. Implement the Feed Forward Backpropagation algorithm to train a neural network.

2. Use a given dataset to train the neural network for a specific task.

3. Evaluate the performance of the trained network on test data

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

For the present case we implement the XOR- operation using Feedforward Backpropagation Neural Network. The XOR-operation for a 2-input variable is as follows



**Python Code:**

import numpy as np

def sigmoid(x):

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

def sigmoid\_derivative(x):

return x \* (1 - x)

class NeuralNetwork:

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

# Initialize weights with random values

self.weights\_input\_hidden = np.random.uniform(-1, 1, (input\_size, hidden\_size))

self.weights\_hidden\_output = np.random.uniform(-1, 1, (hidden\_size, output\_size))

def forward(self, inputs):

# Forward propagation

self.hidden\_input = np.dot(inputs, self.weights\_input\_hidden)

self.hidden\_output = sigmoid(self.hidden\_input)

self.output\_input = np.dot(self.hidden\_output, self.weights\_hidden\_output)

self.predicted\_output = sigmoid(self.output\_input)

return self.predicted\_output

def backward(self, inputs, target, learning\_rate):

# Backpropagation

error = target - self.predicted\_output

delta\_output = error \* sigmoid\_derivative(self.predicted\_output)

error\_hidden = delta\_output.dot(self.weights\_hidden\_output.T)

delta\_hidden = error\_hidden \* sigmoid\_derivative(self.hidden\_output)

# Update weights

self.weights\_hidden\_output += np.outer(self.hidden\_output, delta\_output) \* learning\_rate

self.weights\_input\_hidden += np.outer(inputs, delta\_hidden) \* learning\_rate

def train(self, training\_data, targets, epochs, learning\_rate):

for epoch in range(epochs):

for i in range(len(training\_data)):

inputs = training\_data[i]

target = targets[i]

self.forward(inputs)

self.backward(inputs, target, learning\_rate)

def predict(self, inputs):

return self.forward(inputs)

training\_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

targets = np.array([[0], [1], [1], [0]])

# Create and train the neural network

input\_size = 2

hidden\_size = 4

output\_size = 1

learning\_rate = 0.1

epochs = 10000

nn = NeuralNetwork(input\_size, hidden\_size, output\_size)

nn.train(training\_data, targets, epochs, learning\_rate)

# Test the trained network

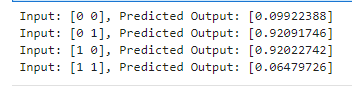
for i in range(len(training\_data)):

inputs = training\_data[i]

prediction = nn.predict(inputs)

print(f"Input: {inputs}, Predicted Output: {prediction}")

**Output:**



**Practical No : 05**

**Aim:** Support Vector Machines (SVM)

• Implement the SVM algorithm for binary classification.

• Train an SVM model using a given dataset and optimize its parameters.

• Evaluate the performance of the SVM model on test data and analyze the results.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |
| --- | --- |
| **DataSet** | We download the Iris data set and use it for the present case. As we are performing the practical on Google Colab, we upload the dataset as Iris.csv |

**Python Code:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.svm import SVC

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

# Load the Iris dataset from scikit-learn

from sklearn.datasets import load\_iris

# Load dataset

data = pd.read\_csv('Iris.csv')

X = data.drop(['Id', 'Species'], axis=1) # Drop 'Id' and 'Species' columns

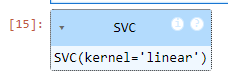
y = data['Species']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

svm\_classifier = SVC(kernel='linear')

svm\_classifier.fit(X\_train, y\_train)

**Output:**



y\_pred = svm\_classifier.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

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

**Output:**



**Practical No: 06**

**Aim:** Naive Bayes' Classifier

• Implement the Naive Bayes' algorithm for classification.

• Train a Naive Bayes' model using a given dataset and calculate class probabilities.

• Evaluate the accuracy of the model on test data and analyze the results.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |
| --- | --- |
| **DataSet** | We download the Iris data set and use it for the present case. As we are performing the practical on Google Colab, we upload the dataset as Iris.csv |

**Python Code:**

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score

# Load the Iris dataset

iris = load\_iris()

X = iris.data

y = iris.target

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

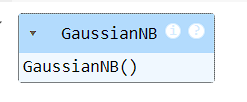
# Create a Naïve Bayes classifier (Gaussian Naïve Bayes for continuous features)

clf = GaussianNB()

# Train the classifier on the training data

clf.fit(X\_train, y\_train)

**Output:**



# Make predictions on the test data

y\_pred = clf.predict(X\_test)

# Calculate and print the accuracy

accuracy = accuracy\_score(y\_test, y\_pred)

print("Accuracy:", accuracy)

**Output:**



**Practical No : 07**

**Aim:** K-Nearest Neighbors (K-NN)

• Implement the K-NN algorithm for classification or regression.

• Apply the K-NN algorithm to a given dataset and predict the class or value for test data.

• Evaluate the accuracy or error of the predictions and analyze the results.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

|  |  |
| --- | --- |
| **DataSet** | We download the Iris data set and use it for the present case. As we are performing the practical on Google Colab, we upload the dataset as Iris.csv |

**Python Code:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor

from sklearn.metrics import accuracy\_score, mean\_squared\_error

# Load the dataset (use the Iris dataset for classification as an example)

data = pd.read\_csv('/content/Iris.csv')

# For classification, assume the target variable is 'Species'

X = data.drop('Species', axis=1)  # Features are all columns except 'Species'

y = data['Species']  # Target is the 'Species' column

# Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

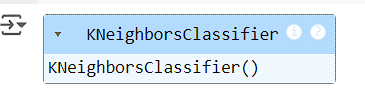
k = 5  # You can experiment with different values of k

knn\_classifier = KNeighborsClassifier(n\_neighbors=k)

# Fit the classifier to the training data

knn\_classifier.fit(X\_train, y\_train)

**Output:**



# Make predictions on the test data

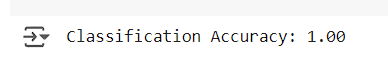
y\_pred = knn\_classifier.predict(X\_test)

# Evaluate the accuracy of the classifier

accuracy = accuracy\_score(y\_test, y\_pred)

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

**Output:**



**Practical No: 08**

**Aim:** **Demo of TensorFlow Tools**

• Explore and experiment with TensorFlow tools and libraries.

• Perform a demonstration or mini-project showcasing the capabilities of the tools.

• Discuss and present the findings and potential applications.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Before running the code we create two files having emails, one is spam mail and other is non-spam email and save them as Spam.txt and NSpam.txt

|  |  |
| --- | --- |
| **Spam.txt:** | Subject: Congratulations! You've Won $100,000 Cash Prize  Dear George,  I am thrilled to inform you that you are the lucky winner of our recent contest and have been awarded a cash prize of $100,000! Your participation and enthusiasm are truly appreciated, and we couldn't be happier to share this exciting news with you.  Regards  idontSmile |
| **NSpam.txt:** | Subject: Invitation for Dinner  Dear Friend,  I hope this email finds you well. I wanted to extend a warm invitation to you for a dinner party on 18/8/2023 at my home next Friday, and it would be wonderful to have you join us.  Best regards,  iSmile |

**Python Code:**

import tensorflow as tf

import numpy as np

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

# Step 1: Prepare a dataset of labeled emails (spam and non-spam)

emails = [

"Buy cheap watches! Free shipping!",

"Meeting for lunch today?",

"Claim your prize! You've won $1,000,000!",

"Important meeting at 3 PM."

]

labels = [1, 0, 1, 0]

# Step 2: Tokenize and pad the email text data

max\_words = 1000 # Maximum number of words to consider

max\_len = 50 # Maximum length of each email (in terms of words)

tokenizer = Tokenizer(num\_words=max\_words, oov\_token="<OOV>") # Out of Vocabulary token

tokenizer.fit\_on\_texts(emails)

sequences = tokenizer.texts\_to\_sequences(emails)

X\_padded = pad\_sequences(sequences, maxlen=max\_len, padding="post", truncating="post")

# Step 3: Define the neural network model

model = tf.keras.Sequential([

tf.keras.layers.Embedding(input\_dim=max\_words, output\_dim=16, input\_length=max\_len),

tf.keras.layers.Flatten(),

tf.keras.layers.Dense(16, activation='relu'),

tf.keras.layers.Dense(1, activation='sigmoid')

])

# Compile the model

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

# Step 4: Define training data and labels as NumPy arrays

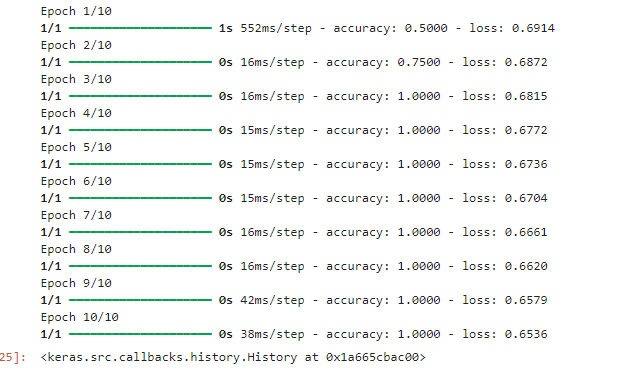
training\_data = np.array(X\_padded)

training\_labels = np.array(labels)

# Step 5: Train the model

model.fit(training\_data, training\_labels, epochs=10)

**Output:**



# Step 6: Test if 'Spam.txt' is spam or not

file\_path = "Spam.txt"

try:

with open(file\_path, "r", encoding="utf-8") as file:

sample\_email\_text = file.read()

sequences\_sample = tokenizer.texts\_to\_sequences([sample\_email\_text])

sample\_email\_padded = pad\_sequences(sequences\_sample, maxlen=50, padding="post", truncating="post")

prediction = model.predict(sample\_email\_padded)

if prediction > 0.5:

print(f"Sample Email ('{file\_path}'): SPAM")

else:

print(f"Sample Email ('{file\_path}'): NOT SPAM")

except FileNotFoundError:

print(f"Error: The file {file\_path} was not found.")

except Exception as e:

print(f"An error occurred: {e}")

**We run the above code with the text file NSpam.txt (file\_path = "NSpam.txt")and get the following**

|  |  |
| --- | --- |
| **Output:** |  |

**We run the above code with the text file Spam.txt (file\_path = "Spam.txt") and get the following**

|  |  |
| --- | --- |
| **Output:** |  |