import random

def print\_board(board):

for row in board:

print(" | ".join(row))

print("-" \* 9)

def check\_winner(board, player):

for row in board + list(zip(\*board)) + [board[i][i] for i in range(3)] + [board[i][2 - i] for i in range(3)]:

if all(cell == player for cell in row):

return True

return False

def is\_full(board):

return all(cell != " " for row in board for cell in row)

def get\_empty\_cells(board):

return [(row, col) for row in range(3) for col in range(3) if board[row][col] == " "]

def ai\_move(board):

return random.choice(get\_empty\_cells(board))

def main():

board = [[" " for \_ in range(3)] for \_ in range(3)]

player = "X"

print("Welcome to Tic Tac Toe!")

while True:

print\_board(board)

if player == "X":

row, col = map(int, input("Enter row and column (0, 1, 2) for your move: ").split())

else:

print("AI's turn:")

row, col = ai\_move(board)

if board[row][col] == " ":

board[row][col] = player

if check\_winner(board, player):

print\_board(board)

print(f"{player} wins!")

break

elif is\_full(board):

print\_board(board)

print("It's a draw!")

break

player = "O" if player == "X" else "X"

else:

print("Cell already occupied. Try again.")

main()

from collections import deque

graph = {

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

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

'C': ['A', 'F'],

'D': ['B'],

'E': ['B', 'F'],

'F': ['C', 'E']

}

def bfs(graph, start):

visited = set()

queue = deque([start])

while queue:

node = queue.popleft()

if node not in visited:

print(node, end=' ') # Print the current node

visited.add(node)

queue.extend(neighbor for neighbor in graph[node] if neighbor not in visited)

# Call the BFS function starting from 'A'

bfs(graph, 'A')

# Define a graph as an adjacency list

graph = {

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

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

'C': ['A', 'F'],

'D': ['B'],

'E': ['B', 'F'],

'F': ['C', 'E']

}

def dfs(graph, node, visited):

if node not in visited:

print(node, end=' ')# Print the current node

visited.add(node)# Mark the node as visited

for neighbor in graph[node]:

dfs(graph, neighbor, visited)# Recursively visit neighbors

visited = set()

dfs(graph, 'A', visited)

from sklearn.neighbors import KNeighborsClassifier

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

from sklearn.datasets import load\_iris

import numpy as np

import matplotlib.pyplot as plt

irisData = load\_iris()

X = irisData.data

y = irisData.target

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

neighbors = np.arange(1, 9)

train\_accuracy = np.empty(len(neighbors))

test\_accuracy = np.empty(len(neighbors))

for i, k in enumerate(neighbors):

knn = KNeighborsClassifier(n\_neighbors=k)

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

train\_accuracy[i] = knn.score(X\_train, y\_train)

test\_accuracy[i] = knn.score(X\_test, y\_test)

plt.plot(neighbors, test\_accuracy, label='Testing dataset Accuracy')

plt.plot(neighbors, train\_accuracy, label='Training dataset Accuracy')

plt.legend()

plt.xlabel('n\_neighbors')

plt.ylabel('Accuracy')

plt.show()

import numpy as np

import matplotlib.pyplot as plt

def estimate\_coef(x, y):

# number of observations/points

n = np.size(x)

# mean of x and y vector

m\_x = np.mean(x)

m\_y = np.mean(y)

SS\_xy = np.sum(y\*x) - n\*m\_y\*m\_x

SS\_xx = np.sum(x\*x) - n\*m\_x\*m\_x

b\_1 = SS\_xy / SS\_xx

b\_0 = m\_y - b\_1\*m\_x

return (b\_0, b\_1)

def plot\_regression\_line(x, y, b):

plt.scatter(x, y, color="m", marker="o", s=30)

y\_pred = b[0] + b[1]\*x

plt.plot(x, y\_pred, color="g")

plt.xlabel('x')

plt.ylabel('y')

plt.show()

def main():

x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

y = np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])

b = estimate\_coef(x, y)

print("Estimated coefficients:\nb\_0 = {}\nb\_1 = {}".format(b[0], b[1]))

plot\_regression\_line(x, y, b)

if \_\_name\_\_ == "\_\_main":

main()

import numpy as np

import pandas as pd

from sklearn.datasets import load\_iris

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

import matplotlib.pyplot as plt

data = load\_iris()

X = data.data

y = data.target

y = pd.get\_dummies(y).values

y[:3]

# Split data into train and test data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=4)

# Initialize variables

learning\_rate = 0.1

iterations = 5000

N = y\_train.shape[0]

input\_size = 4

hidden\_size = 2

output\_size = 3

results = pd.DataFrame(columns=["mse", "accuracy"])

np.random.seed(10)

W1 = np.random.normal(scale=0.5, size=(input\_size, hidden\_size))

W2 = np.random.normal(scale=0.5, size=(hidden\_size, output\_size))

def sigmoid(x):

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

def mean\_squared\_error(y\_pred, y\_true):

return ((y\_pred - y\_true) \*\* 2).sum() / (2 \* y\_pred.shape[0])

def accuracy(y\_pred, y\_true):

acc = y\_pred.argmax(axis=1) == y\_true.argmax(axis=1)

return acc.mean()

for itr in range(iterations):

Z1 = np.dot(X\_train, W1)

A1 = sigmoid(Z1)

Z2 = np.dot(A1, W2)

A2 = sigmoid(Z2)

mse = mean\_squared\_error(A2, y\_train)

acc = accuracy(A2, y\_train)

results = results.append({"mse": mse, "accuracy": acc}, ignore\_index=True)

E1 = A2 - y\_train

dW1 = E1 \* A2 \* (1 - A2)

E2 = np.dot(dW1, W2.T)

dW2 = E2 \* A1 \* (1 - A1)

W2\_update = np.dot(A1.T, dW1) / N

W1\_update = np.dot(X\_train.T, dW2) / N

W2 = W2 - learning\_rate \* W2\_update

W1 = W1 - learning\_rate \* W1\_update

results.mse.plot(title="Mean Squared Error")

results.accuracy.plot(title="Accuracy")

plt.show()

# Feedforward

Z1 = np.dot(X\_test, W1)

A1 = sigmoid(Z1)

Z2 = np.dot(A1, W2)

A2 = sigmoid(Z2)

acc = accuracy(A2, y\_test)

print("Test Accuracy:", acc)