```
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)
```