**الأكواد البرمجية - الذكاء الاصطناعي (١-١):**

**1-1 : حذف وإضافة عناصر للمكدس : ص 30**

**myStack=[1,21,32,45]**

**print("Initial stack: ", myStack)**

**print(myStack.pop())**

**print(myStack.pop())**

**print("The new stack after pop: ", myStack)**

**myStack.append(78)**

**print("The new stack after push: ", myStack)**

**1-2 : حذف جميع عناصر المكدس : ص 30**

**myStack=[1,21,32,45]**

**print("Initial stack:", myStack)**

**a=len(myStack)**

**print("size of stack",a)**

**# empty the stack**

**for i in range(a):**

 **myStack.pop()**

**print(myStack)**

**myStack.pop()**

**1-3 : إضافة و حذف عناصر من المكدس وفقا لقائمة إجراءات : ص 31**

**def push(stack,element):
 stack.append(element)
def pop(stack):
 return stack.pop()
def isEmpty(stack):
 return len(stack)==0
def createStack():
 return []
newStack=createStack()
while True:
 print("The stack so far is:",newStack)
 print("-----------------------------")
 print("Choose 1 for push")
 print("Choose 2 for pop")
 print("Choose 3 for end")
 print("-----------------------------")
 choice=int(input("Enter your choice: "))
 while choice!=1 and choice!=2 and choice!=3:
 print ("Error")
 choice=int(input("Enter your choice: "))
 if choice==1:
 x=int(input("Enter element for push: "))
 push(newStack,x)
 elif choice==2:
 if not isEmpty(newStack):
 print("The pop element is:",pop(newStack))
 else:
 print("The stack is empty")
 else:
 print("End of program")
 break;**

**myQueue=[1,21,32,45]
print("Initial queue: ", myQueue)
myQueue.pop(0)
myQueue.pop(0)
print("The new queue after pop: ", myQueue)
myQueue.append(78)
print("The new queue after push: ", myQueue)**

**1-4 : حذف و إضافة عناصر للطابور : ص 36**

**1-5 : حذف جميع عناصر الطابور: ص 36**

**myQueue=[1,21,32,45]
print("Initial queue: ", myQueue)
a=len(myQueue)
print("size of queue ",a)
# empty the queue
for i in range(a):
 myQueue.pop(0)
print(myQueue)
myQueue.pop(0)**

**1-6 : انشاء طابور بوظيفة وحدة نمطية: ص 37**

**from queue import\*
myQueue = Queue()
# add the elements in the queue
myQueue.put("a")
myQueue.put("b")
myQueue.put("c")
myQueue.put("d")
myQueue.put("e")
# print the elements of the queue
for element in list(myQueue.queue):
 print(element)**

**from queue import \*
myQueue = Queue()
for i in range(5):
 element=input("enter queue element: ")
 myQueue.put(element)
for element in list(myQueue.queue):
 print(element)
 print ("Queue size is: ",myQueue.qsize())**

**1-7 : انشاء طابور بوظيفة وحدة نمطية لقيم يدخلها المستخدم: ص 38**

**1-8 : التحقق من أن الطابور ممتلئ أو فارغ: ص 38**

**from queue import \*
myQueue = Queue()
myQueue.put("a")
myQueue.put("b")
myQueue.put("c")
myQueue.put("d")
myQueue.put("e")
checkFull=myQueue.full()
print("Is the queue full? ", checkFull)
checkEmpty= myQueue.empty()
print("Is the queue empty? ", checkEmpty)**

**1-9 : انشاء مكدس و حذف عناصره بوظيفة وحدة نمطية : ص 39**

**from queue import \***

**myStack = LifoQueue()**

**myStack.put("a")**

**myStack.put("b")**

**myStack.put("c")**

**myStack.put("d")**

**myStack.put("e")**

**for i in range(5):**

 **k=myStack.get()**

 **print(k)**

**checkEmpty= myStack.empty()**

**print("Is the stack empty?", checkEmpty)**

**from queue import \*
import time
printDocument = " "
printQueueSize = 0
printQueueMaxSize = 7
printQueue = Queue(printQueueMaxSize)
# add a document to print the queue
def addDocument(document):
 printQueueSize = printQueue.qsize()
 if printQueueSize == printQueueMaxSize:
 print("!! ", document, " was not sent to print queue.")
 print("The print queue is full.")
 print()
 return
 printQueue.put(document)
 time.sleep(0.5) #Wait 5.0 seconds
 print(document, " sent to print queue.")
 printQueueSizeMessage()
 # print a document from the print queue
def printDocument():
 printQueueSize = printQueue.qsize()
 if printQueueSize == 0:
 print("!! The print queue is empty.")
 print()
 return
 printDocument = printQueue.get()
 time.sleep(1) # wait one second
 print ("OK - ", printDocument, " is printed.")
 printQueueSizeMessage()
 # print a message with the size of the queue
def printQueueSizeMessage():
 printQueueSize = printQueue.qsize()
 if printQueueSize == 0:
 print ("There are no documents waiting for printing.")
 elif printQueueSize == 1:
 print ("There is 1 document waiting for printing.")
 else:
 print ("There are ", printQueueSize, " documents waiting for printing.")
 print()
addDocument("Document A")
addDocument("Document B")
addDocument("Document C")
addDocument("Document D")
addDocument("Document E")
addDocument("Document F")
addDocument("Document G")
printDocument()
addDocument("Document H")
printDocument()
addDocument("Document I")
printDocument()
addDocument("Document J")
addDocument("Document K")
printDocument()
printDocument()
printDocument()
printDocument()
printDocument()
printDocument()
printDocument()
printDocument()**

**1-10 : محاكاة طابور طباعة الملفات : ص 40-41**

**1-11 : انشاء عقدة باستخدام الفئة : ص 46**

**class Node:
 def \_\_init\_\_(self, data, next=None):
 self.data = data # node data
 self.next = next # Pointer to the next node
 # Create a single node
first = Node("Monday")
print(first.data)**

**1-12 : انشاء قائمة مترابطة بعقدة واحدة : ص 46**

**# single node
class Node:
 def \_\_init\_\_(self, data = None, next=None):
 self.data = data
 self.next = next
 # linked list with one head node
class LinkedList:
 def \_\_init\_\_(self):
 self.head = None
 # list linked with a single node
Linkedlist1 = LinkedList()
Linkedlist1.head = Node("Monday")
print(Linkedlist1.head.data)**

**1-13 : انشاء قائمة مترابطة تحتوي عدة عقد:** **ص47**

**class Node:
 def \_\_init\_\_(self, data = None, next=None):
 self.data = data
 self.next = next
 # an empty linked list with a head node.
class LinkedList:
 def \_\_init\_\_(self):
 self.head = None
# the main program
linked\_list = LinkedList()
# the first node
linked\_list.head = Node("Monday")
# the second node
linked\_list.head.next = Node("Tuesday")
# the third node
linked\_list.head.next.next = Node("Wednesday")
# print the linked list
node = linked\_list.head
while node:
 print (node.data)
 node = node.next**

**# single node**

**1-14 : إضافة عقدة إلى القائمة المترابطة : ص 48**

**class Node:**

 **def \_\_init\_\_(self, data = None, next=None):**

 **self.data = data**

 **self.next = next**

**# linked list with one head node**

**class LinkedList:**

 **def \_init\_(self):**

 **self.head = None**

**def insertAfter(new, prev):**

 **# create the new node**

 **new\_node = Node(new)**

 **# make the next of the new node the same as the next of the previous node**

 **new\_node.next = prev.next**

 **# make the next of the previous node the new node**

 **prev.next = new\_node**

**# create the linked list**

**L\_list = LinkedList()**

**# add the first two nodes**

**L\_list.head = Node(12)**

**second = Node(99)**

**L\_list.head.next = second**

**# insert the new node after node 12 (the head of the list)**

**insertAfter(37, L\_list.head)**

**# print the linked list**

**node = L\_list.head**

**while node:**

 **print (node.data)**

**class Node:**

**1-15 : حذف عقدة من القائمة المترابطة :** **ص 49**

**def \_\_init\_\_(self, data=None, next\_node=None):**

**self.data = data**

**self.next = next\_node**

**class LinkedList:**

**def \_\_init\_\_(self):**

**self.head = None**

**def deleteNode(key, linked\_list):**

**# Store the head node**

**temp = linked\_list.head**

**# If the head node itself holds the key to be deleted**

**if temp is not None and temp.data == key:**

**linked\_list.head = temp.next**

**temp = None**

**return**

**# Search for the key to be deleted, keep track of the previous node**

**while temp is not None:**

**if temp.data == key:**

**break**

**prev = temp**

**temp = temp.next**

**# If the key was not found in the linked list**

**if temp is None:**

**return**

**# Unlink the node from the linked list**

**prev.next = temp.next**

**temp = None**

**# Create the linked list**

**L\_list = LinkedList()**

**# Add the first three nodes**

**L\_list.head = Node(12)**

**second = Node(37)**

**third = Node(99)**

**L\_list.head.next = second**

**second.next = third**

**# Delete node with data 37**

**deleteNode(37, L\_list)**

**# Print the linked list**

**node = L\_list.head**

**while node:**

**print(node.data)**

**node = node.next**

**1-16 : إنشاء شجرة باستخدام قاموس البايثون : ص 56**

**myTree = {
 "a": ["b", "c"], # node
 "b": ["d", "e"],
 "c": [None, "f"],
 "d": [None, None],
 "e": [None, None],
 "f": [None, None],
 }
print(myTree)**

**1-17 : إنشاء شجرة بطباعة عدد العقد المنبثقة و عناصرها : ص 56**

**myTree = {"Data Structures":["Linear","Non-linear"],
 "Linear":["Stack","Queue","Linked List"],
 "Non-linear":["Tree", "Graph"]}
for parent in myTree:
 print(parent, "has",len(myTree[parent]),"nodes" )
 for children in myTree[parent]:
 print(" ",children)**

**myGraph = { "a" : ["b","c"],**

**1-18 : إنشاء مخطط موجه : ص 61**

**"b" : ["c", "d"],**

**"c" : ["d", "e"],**

**"d" : [],**

**"e" : [],**

**}**

**print(myGraph)**

**1-19 : إنشاء مخطط غير موجه و إضافة حواف جديدة : ص 61-62**

**myGraph = {"0": ["2", "6"],
 "1": ["9", "0"],
 "2": ["7"],
 "3": ["5", "0"],
 "4": ["2", "1"],
 "5": ["4"],
 "6": ["4"],
 "7": ["1", "8"],
 "8": ["5"],
 "9": ["6"],

 }
print(myGraph)**

**def mySumGrade (gradesList):
 sumGrade=0
 l=len(gradesList)
 for i in range(l):
 sumGrade=sumGrade+gradesList[i]
 return sumGrade
def avgFunc (gradesList):
 s=mySumGrade(gradesList)
 l=len(gradesList)
 avg=s/l
 return avg
# program section
grades=[89,88,98,95]
averageGrade=avgFunc(grades)
print ("The average grade is: ",averageGrade)**

**2-1 : حساب المتوسط : ص 72**

**2-2 : حساب مضروب رقم باستخدام حلقة for : ص 73**

**def factorial(x):
 if x == 0:
 return 1
 else:
 return (x \* factorial(x-1))
# main program
num = int(input("Type a number: "))
f=factorial(num)
print("The factorial of ", num, " is: ", f)**

**2-3 : حساب مضروب رقم بدالة المضروب : ص 74**

**def factorial(x):**

 **if x == 0:**

 **return 1**

 **else:**

 **return (x \* factorial(x-1))**

**# main program**

**num = int(input("Type a number: "))**

**f=factorial(num)**

**print("The factorial of ", num, " is: ", f)**

**2-4 استخراج أكبر عنصر في قائمة : ص 75**

**def findMaxRecursion(A,n):
 if n==1:
 m = A[n-1]
 else:
 m = max(A[n-1],findMaxRecursion(A,n-1))
 return m
def findMaxIteration(A,n):
 m = A[0]
 for i in range(1,n):
 m = max(m,A[i])
 return m
# main program
myList = [3,73,-5,42]
l = len(myList)
myMaxRecursion = findMaxRecursion(myList,l)
print("Max with recursion is: ", myMaxRecursion)
myMaxIteration = findMaxIteration(myList,l)
print("Max with iteration is: ", myMaxIteration)**

**2-٥: تطبيق خوارزمية العمق DFS على مخطط : ص 84**

**graph = {
 "A" : ["B","C"],
 "B" : ["D","E"],
 "C" : ["F"],
 "D" : [],
 "E" : [],
 "F" : []
}
visitedBFS = [] # List to keep track of visited nodes
queue = [] # Initialize a queue
# bfs function
def bfs(visited, graph, node):
 visited.append(node)
 queue.append(node)
 while queue:
 n = queue.pop(0)
 print (n, end = " ")
 for neighbor in graph[n]:
 if neighbor not in visited:
 visited.append(neighbor)
 queue.append(neighbor)
# main program
bfs(visitedBFS, graph, "A")**

**2-٦: تطبيق خوارزمية العمق DFS على مخطط : ص 84**

**graph = {
 "A": ["B", "C"],
 "B": ["D", "E"],
 "C": ["F"],
 "D": [],
 "E": [],
 "F": []
}
visitedDFS = [] # list to keep track of visited nodes

# dfs function
def dfs(visited, graph, node):
 if node not in visited:
 print(node, end=" ")
 visited.append(node)
 for neighbor in graph[node]:
 dfs(visited, graph, neighbor)

dfs(visitedDFS, graph, "A")**

**import json**

**2-٧: التشخيص الطبي الاصدار الأول: ص 90 الى92**

**def load\_symptom\_mapping(symptom\_mapping\_file):**

 **try:**

 **with open(symptom\_mapping\_file, 'r') as f:**

 **mapping = json.load(f)**

 **return mapping**

 **except FileNotFoundError:**

 **print(f"Error: File '{symptom\_mapping\_file}' not found.")**

 **except json.JSONDecodeError:**

 **print(f"Error: Failed to load JSON from '{symptom\_mapping\_file}'.")**

**def diagnose\_disease(symptoms, mapping):**

 **diagnosis = []**

 **for symptom in symptoms:**

 **if symptom in mapping:**

 **diagnosis.extend(mapping[symptom])**

 **return list(set(diagnosis)) # Remove duplicates**

**# Load the symptom mapping from the JSON file**

**symptom\_mapping\_file = 'symptom\_mapping\_v1.json'**

**mapping = load\_symptom\_mapping(symptom\_mapping\_file)**

**if mapping:**

 **# Patient 1**

 **symptoms\_patient1 = ['abdominal pain', 'fever', 'vomiting']**

 **diagnosis\_patient1 = diagnose\_disease(symptoms\_patient1, mapping)**

 **print('Patient 1 - Most likely diagnosis:', diagnosis\_patient1)**

 **# Patient 2**

 **symptoms\_patient2 = ['vomiting', 'lower back pain', 'fever']**

 **diagnosis\_patient2 = diagnose\_disease(symptoms\_patient2, mapping)**

 **print('Patient 2 - Most likely diagnosis:', diagnosis\_patient2)**

 **# Patient 3**

 **symptoms\_patient3 = ['fever', 'cough', 'vomiting']**

 **diagnosis\_patient3 = diagnose\_disease(symptoms\_patient3, mapping)**

 **print('Patient 3 - Most likely diagnosis:', diagnosis\_patient3)**

import json

**2-8 : التشخيص الطبي الاصدار الثاني: ص 93 الى95**

def load\_symptom\_mapping(symptom\_mapping\_file):

 try:

 with open(symptom\_mapping\_file, 'r') as f:

 mapping = json.load(f)

 return mapping

 except FileNotFoundError:

 print(f"Error: File '{symptom\_mapping\_file}' not found.")

 except json.JSONDecodeError:

 print(f"Error: Failed to load JSON from '{symptom\_mapping\_file}'.")

def diagnose\_v2(patient\_symptoms, mapping, matching\_symptoms\_lower\_bound):

 diagnosis = []

 disease\_info = mapping['diseases']

 for disease in disease\_info:

 counter = 0

 disease\_symptoms = disease\_info[disease]

 for symptom in patient\_symptoms:

 if symptom in disease\_symptoms:

 counter += 1

 if counter >= matching\_symptoms\_lower\_bound:

 diagnosis.append(disease)

 return diagnosis

# Load the symptom mapping from the JSON file

symptom\_mapping\_file = 'symptom\_mapping\_v2.json'

mapping = load\_symptom\_mapping(symptom\_mapping\_file)

if mapping:

 # Example patient cases

 my\_symptoms\_1 = ["stuffy nose", "runny nose", "sneezing", "sore throat"]

 diagnosis\_1 = diagnose\_v2(my\_symptoms\_1, mapping, 3)

 print('Patient 1 - Most likely diagnosis:', diagnosis\_1)

 my\_symptoms\_2 = ["stuffy nose", "runny nose", "sneezing", "sore throat"]

 diagnosis\_2 = diagnose\_v2(my\_symptoms\_2, mapping, 4)

 print('Patient 2 - Most likely diagnosis:', diagnosis\_2)

 my\_symptoms\_3 = ['fever', 'cough', 'vomiting']

 diagnosis\_3 = diagnose\_v2(my\_symptoms\_3, mapping, 3)

 print('Patient 3 - Most likely diagnosis:', diagnosis\_3)

**2-8 : التشخيص الطبي الاصدار الثالث: ص 96 الى98**

**symptom\_mapping\_file='symptom\_mapping\_v3.json'**

**# open the mapping JSON file and load it into a dictionary**

**with open(symptom\_mapping\_file) as f:**

 **mapping=json.load(f)**

**# print the JSON file**

**print(json.dumps(mapping, indent=2))**

**from collections import defaultdict**

**def diagnose\_v3(patient\_symptoms:list,**

 **symptom\_mapping\_file:str,**

 **very\_common\_weight:float=1,**

 **less\_common\_weight:float=0.5**

 **):**

 **with open(symptom\_mapping\_file) as f:**

 **mapping=json.load(f)**

 **disease\_info=mapping['diseases']**

 **# holds a symptom-based score for each potential disease**

 **disease\_scores=defaultdict(int)**

 **for disease in disease\_info:**

 **# get the very common symptoms of the disease**

 **very\_common\_symptoms=disease\_info[disease]['very common']**

 **# get the less common symptoms for this disease**

 **less\_common\_symptoms=disease\_info[disease]['less common']**

 **for symptom in patient\_symptoms:**

 **if symptom in very\_common\_symptoms:**

 **disease\_scores[disease]+=very\_common\_weight**

 **elif symptom in less\_common\_symptoms:**

 **disease\_scores[disease]+=less\_common\_weight**

 **# find the max score all candidate diseases**

 **max\_score=max(disease\_scores.values())**

 **if max\_score==0:**

 **return []**

 **else:**

 **# get all diseases that have the max score**

 **diagnosis=[disease for disease in disease\_scores if disease\_scores**

 **[disease]==max\_score]**

 **return diagnosis, max\_score**

**# Patient 1**

**my\_symptoms=["headache", "tiredness", "cough"]**

**diagnosis=diagnose\_v3(my\_symptoms, 'symptom\_mapping\_v3.json')**

**print('Most likely diagnosis:',diagnosis)**

**# Patient 2**

**my\_symptoms=["stuffy nose", "runny nose", "sneezing", "sore throat"]**

**diagnosis=diagnose\_v3(my\_symptoms, 'symptom\_mapping\_v3.json')**

**print('Most likely diagnosis:',diagnosis)**

**# Patient 3**

**my\_symptoms=["stuffy nose", "runny nose", "sneezing", "sore throat"]**

**diagnosis=diagnose\_v3(my\_symptoms, 'symptom\_mapping\_v3.json', 1, 1)**

**print('Most likely diagnosis:',diagnosis)**

**2-8 : التشخيص الطبي الاصدار الرابع: ص 99 الى104**

**import pandas as pd # import pandas to load and process spreadsheet-type data**

**medical\_dataset=pd.read\_csv('medical\_data.csv') # load a medical dataset.**

**medical\_dataset**

**set(medical\_dataset['diagnosis'])**

**from sklearn.tree import DecisionTreeClassifier**

**def diagnose\_v4(train\_dataset:pd.DataFrame):**

**# create a DecisionTreeClassifier**

 **model=DecisionTreeClassifier(random\_state=1)**

**# drop the diagnosis column to get only the symptoms**

 **train\_patient\_symptoms=train\_dataset.drop(columns=['diagnosis'])**

**# get the diagnosis column, to be used as the classification target**

 **train\_diagnoses=train\_dataset['diagnosis']**

**# build a decision tree**

 **model.fit(train\_patient\_symptoms, train\_diagnoses)**

**# return the trained model**

 **return model**

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

**# use the function to split the data, get 30% for testing and 70% for training.**

**train\_data, test\_data = train\_test\_split(medical\_dataset, test\_size=0.3,**

**random\_state=1)**

**#print the shapes (rows x columns) of the two datasets**

**print(train\_data.shape)**

**print(test\_data.shape)**

**from sklearn.tree import plot\_tree**

**import matplotlib.pyplot as plt**

**my\_tree=diagnose\_v4(train\_data) # train a model**

**print(my\_tree.classes\_) # print the possible target labels (diagnoses)**

**plt.figure(figsize=(12,6)) # size of the visualization, in inches**

**# plot the tree**

**plot\_tree(my\_tree,**

 **max\_depth=2,**

 **fontsize=10,**

 **feature\_names=medical\_dataset.columns[:-1]**

 **)**

**# functions used to evaluate a classifier**

**from sklearn.metrics import accuracy\_score,confusion\_matrix**

**# drop the diagnosis column to get only the symptoms**

**test\_patient\_symptoms=test\_data.drop(columns=['diagnosis'])**

**# get the diagnosis column, to be used as the classification target**

**test\_diagnoses=test\_data['diagnosis']**

**# guess the most likely diagnoses**

**pred=my\_tree.predict(test\_patient\_symptoms)**

**# print the achieved accuracy score**

**accuracy\_score(test\_diagnoses,pred)**

**confusion\_matrix(test\_diagnoses,pred)**

**2-9 : أنشاء متاهة 3\*3: ص 109**

**import random**

**random\_maze=np.zeros((10,10))**

**# coordinates of 30 random cells occupied by blocks**

**blocks=[(random.randint(0,9),random.randint(0,9)) for i in range(30)]**

**for block in blocks:**

 **random\_maze[block]=1**

**import random**

**2-10 : أنشاء متاهة كبيرة و معقدة: ص 110**

**import numpy as np**

**import matplotlib.pyplot as plt**

**# Generate a random maze**

**random\_maze = np.zeros((10, 10))**

**# Coordinates of 30 random cells occupied by blocks**

**blocks = [(random.randint(0, 9), random.randint(0, 9)) for \_ in range(30)]**

**for block in blocks:**

 **random\_maze[block] = 1**

**def plot\_maze(maze):**

 **ax = plt.gca() # Create a new figure**

 **ax.invert\_yaxis() # Invert the y-axis to match the matrix**

 **ax.axis('off') # Hide the axis labels**

 **ax.set\_aspect('equal') # Make sure the cells are rectangular**

 **plt.pcolormesh(maze, edgecolors='black', linewidth=2, cmap='Accent')**

 **plt.show()**

**# Plot the random maze**

**plot\_maze(random\_maze)**

**2-11 : تحديد الخلايا الفارغة المجاورة لخلية محددة: ص 111**

**import numpy as np**

**def get\_accessible\_neighbors(maze: np.ndarray, cell: tuple):**

 **# List of accessible neighbors, initialized to empty**

 **neighbors = []**

 **x, y = cell**

 **# For each adjacent cell position**

 **for i, j in [(x - 1, y - 1), (x - 1, y), (x - 1, y + 1), (x, y - 1), (x, y + 1), (x + 1, y - 1), (x + 1, y), (x + 1, y + 1)]:**

 **# If the adjacent cell is within the bounds of the grid and is not occupied by a block**

 **if 0 <= i < len(maze) and 0 <= j < len(maze[0]) and maze[i, j] == 0:**

 **neighbors.append(((i, j), 1))**

 **return neighbors**

**# Example usage with a small maze (assuming 'small\_maze' is defined)**

**small\_maze = np.array([[0, 1, 0],**

 **[0, 0, 1],**

 **[0, 1, 0]])**

**accessible\_neighbors = get\_accessible\_neighbors(small\_maze, (0, 0))**

**print(accessible\_neighbors)**

**2-12 :استخدام خوارزمية البحث بالاتساع في حل المتاهة: ص 112 الى 114ص**

**import numpy as np**

**# Define a small maze (example usage)**

**small\_maze = np.array([[0, 1, 0],**

 **[0, 0, 1],**

 **[0, 1, 0]])**

**def reconstruct\_shortest\_path(parent: dict, start\_cell: tuple, target\_cell: tuple):**

 **shortest\_path = []**

 **my\_parent = target\_cell**

 **while my\_parent != start\_cell:**

 **shortest\_path.append(my\_parent)**

 **my\_parent = parent[my\_parent]**

 **shortest\_path.append(start\_cell)**

 **shortest\_path.reverse()**

 **return shortest\_path**

**def get\_accessible\_neighbors(maze: np.ndarray, cell: tuple):**

 **neighbors = []**

 **x, y = cell**

 **for i, j in [(x - 1, y - 1), (x - 1, y), (x - 1, y + 1), (x, y - 1), (x, y + 1), (x + 1, y - 1), (x + 1, y), (x + 1, y + 1)]:**

 **if 0 <= i < len(maze) and 0 <= j < len(maze[0]) and maze[i, j] == 0:**

 **neighbors.append(((i, j), 1))**

 **return neighbors**

**def bfs\_maze\_solver(start\_cell: tuple, target\_cell: tuple, maze: np.ndarray, get\_neighbors: callable, verbose: bool = False):**

 **cell\_visits = 0**

 **visited = set()**

 **to\_expand = []**

 **visited.add(start\_cell)**

 **to\_expand.append(start\_cell)**

 **shortest\_distance = {}**

 **shortest\_distance[start\_cell] = 0**

 **parent = {}**

 **parent[start\_cell] = start\_cell**

 **while len(to\_expand) > 0:**

 **next\_cell = to\_expand.pop(0)**

 **if verbose:**

 **print('\nExpanding cell', next\_cell)**

 **for neighbor, cost in get\_neighbors(maze, next\_cell):**

 **if verbose:**

 **print('\tVisiting neighbor cell', neighbor)**

 **cell\_visits += 1**

 **if neighbor not in visited:**

 **visited.add(neighbor)**

 **to\_expand.append(neighbor)**

 **parent[neighbor] = next\_cell**

 **shortest\_distance[neighbor] = shortest\_distance[next\_cell] + cost**

 **if neighbor == target\_cell:**

 **shortest\_path = reconstruct\_shortest\_path(parent, start\_cell, target\_cell)**

 **return shortest\_path, shortest\_distance[target\_cell], cell\_visits**

 **else:**

 **if shortest\_distance[neighbor] > shortest\_distance[next\_cell] + cost:**

 **parent[neighbor] = next\_cell**

 **shortest\_distance[neighbor] = shortest\_distance[next\_cell] + cost**

 **return None, None, None**

**start\_cell = (2, 0) # start cell**

**target\_cell = (1, 2) # target cell**

**solution, distance, cell\_visits = bfs\_maze\_solver(start\_cell, target\_cell, small\_maze, get\_accessible\_neighbors, verbose=True)**

**print('\nShortest Path:', solution)**

**print('Cells on the Shortest Path:', len(solution))**

**print('Shortest Path Distance:', distance)**

**print('Number of cell visits:', cell\_visits)**

**import numpy as np**

**2-13 :استخدام خوارزمية البحث بأولوية الأفضل : ص 117 الى ص 120**

**import sys**

**# Define a small maze (example usage)**

**small\_maze = np.array([[0, 1, 0],**

 **[0, 0, 1],**

 **[0, 1, 0]])**

**def reconstruct\_shortest\_path(parent: dict, start\_cell: tuple, target\_cell: tuple):**

 **shortest\_path = []**

 **my\_parent = target\_cell**

 **while my\_parent != start\_cell:**

 **shortest\_path.append(my\_parent)**

 **my\_parent = parent[my\_parent]**

 **shortest\_path.append(start\_cell)**

 **shortest\_path.reverse()**

 **return shortest\_path**

**def constant\_heuristic(candidate\_cell: tuple, target\_cell: tuple):**

 **return 1**

**def get\_accessible\_neighbors(maze: np.ndarray, cell: tuple):**

 **neighbors = []**

 **x, y = cell**

 **for i, j in [(x - 1, y - 1), (x - 1, y), (x - 1, y + 1), (x, y - 1), (x, y + 1), (x + 1, y - 1), (x + 1, y), (x + 1, y + 1)]:**

 **if 0 <= i < len(maze) and 0 <= j < len(maze[0]) and maze[i, j] == 0:**

 **neighbors.append(((i, j), 1))**

 **return neighbors**

**def astar\_maze\_solver(start\_cell: tuple, target\_cell: tuple, maze: np.ndarray, get\_neighbors: callable, heuristic: callable, verbose: bool = False):**

 **cell\_visits = 0**

 **shortest\_distance = {}**

 **shortest\_distance[start\_cell] = 0**

 **parent = {}**

 **parent[start\_cell] = start\_cell**

 **expansion\_candidates = set([start\_cell])**

 **fully\_expanded = set()**

 **while len(expansion\_candidates) > 0:**

 **best\_cell = get\_best\_candidate(expansion\_candidates, shortest\_distance, heuristic)**

 **if best\_cell == None:**

 **break**

 **if verbose:**

 **print('\nExpanding cell', best\_cell)**

 **if best\_cell == target\_cell:**

 **shortest\_path = reconstruct\_shortest\_path(parent, start\_cell, target\_cell)**

 **return shortest\_path, shortest\_distance[target\_cell], cell\_visits**

 **for neighbor, cost in get\_neighbors(maze, best\_cell):**

 **if verbose:**

 **print('\nVisiting neighbor cell', neighbor)**

 **cell\_visits += 1**

 **if neighbor not in expansion\_candidates and neighbor not in fully\_expanded:**

 **expansion\_candidates.add(neighbor)**

 **parent[neighbor] = best\_cell**

 **shortest\_distance[neighbor] = shortest\_distance[best\_cell] + cost**

 **elif shortest\_distance[neighbor] > shortest\_distance[best\_cell] + cost:**

 **shortest\_distance[neighbor] = shortest\_distance[best\_cell] + cost**

 **parent[neighbor] = best\_cell**

 **if neighbor in fully\_expanded:**

 **fully\_expanded.remove(neighbor)**

 **expansion\_candidates.add(neighbor)**

 **expansion\_candidates.remove(best\_cell)**

 **fully\_expanded.add(best\_cell)**

 **return None, None, None**

**def get\_best\_candidate(expansion\_candidates: set, shortest\_distance: dict, heuristic: callable):**

 **winner = None**

 **best\_estimate = sys.maxsize**

 **for candidate in expansion\_candidates:**

 **candidate\_estimate = shortest\_distance[candidate] + heuristic(candidate, target\_cell)**

 **if candidate\_estimate < best\_estimate:**

 **winner = candidate**

 **best\_estimate = candidate\_estimate**

 **return winner**

**start\_cell = (2, 0) # start cell**

**target\_cell = (1, 2) # target cell**

**solution, distance, cell\_visits = astar\_maze\_solver(start\_cell, target\_cell, small\_maze, get\_accessible\_neighbors, constant\_heuristic, verbose=True)**

**print('\nShortest Path:', solution)**

**print('Cells on the Shortest Path:', len(solution))**

**print('Shortest Path Distance:', distance)**

**print('Number of cell visits:', cell\_visits)**

# Install the pandas and scikit-plot libraries if they are missing.

**3-1 :التنبؤ بالانطباع العام عن الفيلم : ص 135 الى ص 142**

!pip install pandas scikit-plot

import pandas as pd

import scikitplot as skplt

from sklearn.feature\_extraction.text import CountVectorizer

from sklearn.naive\_bayes import MultinomialNB

from sklearn.metrics import accuracy\_score

# Read training and testing data from CSV files.

imdb\_train\_reviews = pd.read\_csv('imdb\_train.csv')

imdb\_test\_reviews = pd.read\_csv('imdb\_test.csv')

# Extract text and labels from the DataFrames.

X\_train\_text = imdb\_train\_reviews['text']

X\_test\_text = imdb\_test\_reviews['text']

Y\_train = imdb\_train\_reviews['label']

Y\_test = imdb\_test\_reviews['label']

# Create a CountVectorizer with min\_df parameter.

vectorizer\_v1 = CountVectorizer(min\_df=10)

# Fit the vectorizer on the training data and transform the data.

X\_train\_v1 = vectorizer\_v1.fit\_transform(X\_train\_text)

# Create and fit a Multinomial Naive Bayes classifier.

model\_v1 = MultinomialNB()

model\_v1.fit(X\_train\_v1, Y\_train)

# Create a prediction pipeline: vectorization and prediction.

prediction\_pipeline\_v1 = make\_pipeline(vectorizer\_v1, model\_v1)

# Make predictions on test data.

predictions\_v1 = prediction\_pipeline\_v1.predict(X\_test\_text)

# Calculate and print the accuracy of the classifier.

accuracy = accuracy\_score(Y\_test, predictions\_v1)

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

# Plot the confusion matrix.

skplt.metrics.plot\_confusion\_matrix(

 [class\_names[i] for i in Y\_test],

 [class\_names[i] for i in predictions\_v1],

 title="Confusion Matrix",

 cmap="Purples",

 figsize=(5, 5)

)

# Install the LIME library

**3-2 :النموذج المحايد المحلي القابل للتفسير و الشرح: ص 142 الى ص 145**

!pip install lime

from lime.lime\_text import LimeTextExplainer

# Create a local explainer for explaining individual predictions

explainer\_v1 = LimeTextExplainer(class\_names=class\_names)

# An example of an obviously negative review

easy\_example = 'This movie was horrible. The actors were terrible and the plot was very boring.'

# Use the prediction pipeline to get the prediction probabilities for this example

print(prediction\_pipeline\_v1.predict\_proba([easy\_example]))

# Explain the prediction for this example

exp = explainer\_v1.explain\_instance(easy\_example.lower(), prediction\_pipeline\_v1.predict\_proba, num\_features=10)

# Print the words with the strongest influence on the prediction

exp.as\_list()

# Visualize the impact of the most influential words

fig = exp.as\_pyplot\_figure()

# An example of a positive review that is mis-classified as negative by prediction\_pipeline\_v1

mistake\_example = X\_test\_text[4600]

# Get the correct label of this example

print('Correct Label:', class\_names[Y\_test[4600]])

# Get the prediction probabilities for this example

print('Prediction Probabilities for neg, pos:', prediction\_pipeline\_v1.predict\_proba([mistake\_example]))

# Explain the prediction for this example

exp = explainer\_v1.explain\_instance(mistake\_example, prediction\_pipeline\_v1.predict\_proba, num\_features=10)

# Visualize the explanation

fig = exp.as\_pyplot\_figure()

**3-3 :تحسين البرمجة الاتجاهية للنصوص: ص 146 الى ص 149**

!pip install nltk

!pip install gensim

import nltk # import nltk

nltk.download('punkt') # install nltk's tokenization tool, used to split a text into sentences.

import re # import re

from gensim.models.phrases import Phrases, ENGLISH\_CONNECTOR\_WORDS # import tools from the gensim library

# convert a given doc to a list of tokenized sentences.

def tokenize\_doc(doc:str):

 return [re.findall(r'\b\w+\b', sent.lower()) for sent in nltk.sent\_tokenize(doc)]

raw\_text='The movie was too long. I fell asleep after the first 2 hours.'

tokenized\_sentences=tokenize\_doc(raw\_text)

tokenized\_sentences

sentences=[] # list of all the tokenized sentences across all the docs in this dataset

for doc in X\_train\_text: # for each doc in this dataset

 sentences+=tokenize\_doc(doc) # get the list of tokenized sentences in this doc

# build a phrase model on the given data

imdb\_phrase\_model = Phrases(sentences,

 connector\_words=ENGLISH\_CONNECTOR\_WORDS,

 scoring='npmi',

imdb\_phrase\_model[tokenized\_sentences[0]]

imdb\_phrase\_model[tokenized\_sentences[1]]

def annotate\_phrases(doc:str, phrase\_model):

 sentences=tokenize\_doc(doc)# split the document into tokenized sentences.

 tokens=[] # list of all the words and phrases found in the doc

 for sentence in sentences: # for each sentence

# use the phrase model to get tokens and append them to the list.

 tokens+=phrase\_model[sentence]

 return ' '.join(tokens) # join all the tokens together to create a new annotated document

# annotate all the test and train reviews.

X\_train\_text\_annotated=[annotate\_phrases(doc,imdb\_phrase\_model) for doc in X\_train\_text]

X\_test\_text\_annotated=[annotate\_phrases(text,imdb\_phrase\_model) for text in X\_test\_text]

# visualize the explanation.

# an example of an annotated document from the imdb training data

X\_train\_text\_annotated[0]

from sklearn.feature\_extraction.text import TfidfVectorizer

**3-4 :تحسين البرمجة الاتجاهية للنصوص: ص 149 الى ص 151**

from sklearn.metrics import accuracy\_score

# Train a TF-IDF model with the IMDb training dataset

vectorizer\_tf = TfidfVectorizer(min\_df=10)

vectorizer\_tf.fit(X\_train\_text\_annotated)

X\_train\_tf = vectorizer\_tf.transform(X\_train\_text\_annotated)

# Train a new Naive Bayes Classifier on the newly vectorized data

model\_tf = MultinomialNB()

model\_tf.fit(X\_train\_tf, Y\_train)

# Create a new prediction pipeline

prediction\_pipeline\_tf = make\_pipeline(vectorizer\_tf, model\_tf)

# Get predictions using the new pipeline

predictions\_tf = prediction\_pipeline\_tf.predict(X\_test\_text\_annotated)

# Print the achieved accuracy

accuracy = accuracy\_score(Y\_test, predictions\_tf)

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

# Get the review example that confused the previous algorithm

mistake\_example\_annotated = X\_test\_text\_annotated[4600]

print('\nReview:', mistake\_example\_annotated)

# Get the correct labels of this example

print('Correct Label:', class\_names[Y\_test[4600]])

# Get the prediction probabilities for this example

print('Prediction Probabilities for neg, pos:', prediction\_pipeline\_tf.predict\_proba([mistake\_example\_annotated]))

# Create an explainer

explainer\_tf = LimeTextExplainer(class\_names=class\_names)

# Explain the prediction of the second pipeline for this example

exp = explainer\_tf.explain\_instance(mistake\_example\_annotated, prediction\_pipeline\_tf.predict\_proba, num\_features=10)

# Visualize the results

fig = exp.as\_pyplot\_figure()

****

**رابط المجموعات والقنوات:**

[**https://t.me/addlist/NzeSzwwOius5MTA0**](https://t.me/addlist/NzeSzwwOius5MTA0)