ANNAMALAI UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

B.E. COMPUTER SCIENCE AND ENGINEERING

(ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)

AICP 309 ARTIFICIAL INTELLIGENCE LABORATORY

Name : ______________________________

Roll no : ______________________________

ANNAMALAI UNIVERSITY

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

B.E. COMPUTER SCIENCE AND ENGINEERING

(ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)

III SEMESTER

AICP 309 ARTIFICIAL INTELLIGENCE LABORATORY

Bonafide Certificate

Certified that this is the Bonafide record of work done by Mr./Ms. ____ Reg No. _______________ of III semester B.E Computer science and Engineering (Artificial Intelligence and Machine Learning) in the AICP309 - Artificial Intelligence Laboratory during odd semester (July 23 - November 23)

Staff in-Charge

Internal Examiner **External Examiner**

Place : Annamalai Nagar
Date :

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

B.E. COMPUTER SCIENCE AND ENGINEERING (Artificial Intelligence and Machine Learning)

VISION

To provide a congenial ambience for individuals to develop and blossom as academically superior, socially conscious and nationally responsible citizens.

MISSION

M1: Impart high quality computer knowledge to the students through a dynamic scholastic environment wherein they learn to develop technical, communication and leadership skills to bloom as a versatile professional.

M2: Develop life-long learning ability that allows them to be adaptive and responsive to the changes in career, society, technology, and environment.

M3: Build student community with high ethical standards to undertake innovative research and development in thrust areas of national and international needs.

M4: Expose the students to the emerging technological advancements for meeting the demands of the industry.

B.E. COMPUTER SCIENCE AND ENGINEERING (Artificial Intelligence and Machine Learning)

PROGRAMME EDUCATIONAL OBJECTIVES (PEO)

PEO	PEO Statements
PEO1	To prepare graduates with potential to get employed in the right role and/or become entrepreneurs to contribute to the society.
PEO2	To provide the graduates with the requisite knowledge to pursue higher education and carry out research in the field of Computer Science and Engineering.
PEO3	To equip the graduates with the skills required to stay motivated and adapt to the dynamically changing world so as to remain successful in their career.
PEO4	To train the graduates to communicate effectively, work collaboratively and exhibit high levels of professionalism and ethical responsibility.

		L	T	C	P
AICP309	**ARTIFICIAL INTELLIGENCE LABORATORY**	0	0	3	1.5

COURSE OBJECTIVES:

1. To learn Python Programming and Key Python Libraries relate to AI.
2. To formulate Real World Problems for Al.
3. To study specific algorithm design methods related to game playing.
4. To understand the process involved in computing with natural language specifically: Texts and Words

LIST OF EXERCISES

1. Write a program to implement Blind Search Techniques like Breadth First and Depth First Search Traversal.
2. Write a program to implement Heuristic Search Technique.
3. Write a program to implement Cryptarithmetic Problem.
4. Write a program to Create a Knowledge Base of Facts and Convert them to Predicate Logic.
5. Write a program to Create a semantic network.
6. Write a program to Calculate conditional probability using–Naïve Bayes theorem.
7. Write a program to implement Game Playing Algorithm like Min-Max Algorithm and Alpha — Beta Pruning.
8. Write a program to implement Natural Language Processing.
9. Write a program to implement Spell Checking using NLTK.
10. Write a program to implement Developing a Simple Medical Expert System.

COURSE OUTCOMES:

At the end of this course, the students will be able to

1. Understand the problem as a state space, design heuristics and select amongst differentsearch based techniques to solve them.
2. Analyze the design heuristics and apply different game based techniques to solve gameplaying problems.
3. Demonstrate an ability to listen and answer the viva questions related to programming skills needed for solving real-world problems in Computer Science and Engineering.

Mapping of Course Outcomes with Programme Outcomes												
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	2	2	3	2	-	-	-	-	-	-	-	-
CO2	1	2	-	2	-	-	-	-	-	-	-	-
CO3	2	2	-	1	-	-	-	-	-	2	-	2

Rubric for CO3

Rubric for CO3 in Laboratory Courses				
Rubric	Distribution of 10 Marks for CIE/SEE Evaluation Out of 40/60 Marks			
	Up To 2.5 Marks	Up To 5 Marks	Up To 7.5 Marks	Up To 10 marks
Demonstrate an ability to listen and answer the viva questions related to programming skills needed for solving real-world problems in Computer Science and Engineering.	Poor listening and communication skills. Failed to relate the programming skills needed for solving the problem.	Showed better communication skill by relating the problem with the programming skills acquired but the description showed serious errors.	Demonstrated good communication skills by relating the problem with the programming skills acquired with few errors.	Demonstrated excellent communication skills by relating the problem with the programming skills acquired and have been successful in tailoring the description.

EX NO.	CONTENTS	PAGE NO.	MARKS	STAFF SIGNATURE
1	**Blind Search Techniques** a) Breadth First Search (BFS) b) Depth First Search (DFS)			
2	**Heuristic Search Technique**			
3	**Cryptarithmetic Problem**			
4	**Create a Knowledge Base of Facts and Convert them to Predicate Logic**			
5	**Create a semantic network**			
6	**Calculate conditional probability using– Naïve Bayes theorem**			
7	**Game Playing** a) MIN-MAX ALGORITHM b) ALPHA-BETA PRUNING			
8	**NLP: Tokenization and stemming using NLTK**			
9	**Spell Checking using NLTK**			
10	**Developing a Simple Medical Expert System**			

Exercise 01 – Blind Search Techniques

a) Breadth First Search (BFS)

AIM:

To write a python program to implement the BreadthFirst Search algorithm in a graph.

ALGORITHM:

Initialization:
- Create an empty set `visited` to keep track of visited vertices.
- Create an empty queue `queue` for BFS traversal.

Initialize the Starting Point:
- Add the starting vertex (e.g., `start`) to the `visited` set to mark it as visited.
- Enqueue the starting vertex into the `queue` for further exploration.

Main BFS Loop:
- While the `queue` is not empty, do the following:
  - Dequeue a vertex `node` from the front of the `queue`. This is the current node being visited.
  - Print the value of `node` to indicate that it has been visited.

Exploring Neighbors:
- For each neighbor of the `node`, which is obtained from the `graph[node]`:
  - If the neighbor is not in the `visited` set, do the following:
    - Add the neighbor to the `visited` set to mark it as visited.
    - Enqueue the neighbor into the `queue` for further exploration.

Repeat:
- Continue this process until the `queue` is empty, indicating that all vertices have been visited.

SOURCE CODE:

```
def bfs(graph, start):
    visited = set()
    queue = []

    visited.add(start)
    queue.append(start)

    while queue:
        node = queue.pop(0)
        print(node, end=' ')
```

```
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

# Example usage:
if _name_ == "_main_":
    graph = {
        0: [1, 2],
        1: [2],
        2: [0, 3],
        3: [3]
    }

    print("Breadth-First Traversal (starting from vertex 2):")
    bfs(graph, 2)
```

OUTPUT:

Breadth-First Traversal (starting from vertex 2):
2 0 3 1

b) Depth First Search (DFS)

AIM:

To write a python program to implement the Depth First Search algorithm in a graph.

ALGORITHM:

Initialize a Set for Visited Nodes:
- Create an empty set called `visited` to keep track of visited vertices.

Define the DFS Function:
- Create a function `dfs` that takes the following parameters:
  - `graph`: The graph representation.
  - `start`: The current vertex to explore.
  - `visited`: The set of visited vertices.

DFS Recursive Implementation:
- Inside the `dfs` function:
  - Check if the `start` vertex is not in the `visited` set.
  - If it's not visited:
    - Print the value of `start` to indicate that it has been visited.
    - Add the `start` vertex to the `visited` set to mark it as visited.
    - For each neighbor of `start` in `graph[start]`, do the following:
      - Recursively call the `dfs` function for the neighbor with the same parameters: `dfs(graph, neighbor, visited)`.
      - This recursive step explores the neighbor vertices in depth-first order.

SOURCE CODE:

```
def dfs(graph, start, visited):
    if start not in visited:
        print(start, end=' ')
        visited.add(start)
        for neighbor in graph[start]:
            dfs(graph, neighbor, visited)

# Example usage:
if _name_ == "_main_":
    graph = {
        'A': ['B', 'C'],
        'B': ['A', 'D', 'E'],
        'C': ['A', 'F'],
        'D': ['B'],
        'E': ['B', 'F'],
```

```python
    'F': ['C', 'E']
}

visited = set()  # Create a set to keep track of visited nodes
print("Depth-First Traversal (starting from vertex 'A'):")
dfs(graph, 'A', visited)
```

OUTPUT:

Depth-First Traversal (starting from vertex 'A'):
A B D E F C

Exercise 02 - Heuristic Search Technique

AIM:

To write a python program to implement A* algorithm.

ALGORITHM:

Initialize Data Structures:

- Create an empty priority queue called `open_list` to store nodes with their f-scores (f-score, node).
- Create an empty dictionary called `came_from` to store the parent node of each node.
- Initialize a dictionary called `g_score` with all nodes in the grid set to infinity, representing the cost to reach each node.
- Set the g-score of the starting node to 0.
- Initialize a dictionary called `f_score` with all nodes in the grid set to infinity, representing the estimated total cost to reach the goal from each node.
- Set the f-score of the starting node to the heuristic value (Manhattan distance) from the starting node to the goal.

A* Loop:

- While the `open_list` is not empty, do the following:
  - Pop the node with the lowest f-score from the `open_list`. This is the current node being considered.

Goal Check:

- If the current node is the goal node, construct and return the path from the start to the goal using the `came_from` dictionary.

Neighbor Expansion:

- For each neighbor of the current node in the up, down, left, and right directions:
  - Calculate the coordinates (x, y) of the neighbor.
  - Check if the neighbor is within the bounds of the grid and is not an obstacle (grid[x][y] == 0).
  - Calculate the tentative g-score for the neighbor by adding 1 to the g-score of the current node.
  - If the tentative g-score is less than the current g-score of the neighbor, do the following:
    - Update the `came_from` dictionary to store the current node as the parent of the neighbor.
    - Update the g-score of the neighbor with the tentative g-score.
    - Update the f-score of the neighbor by adding the tentative g-score to the heuristic value from the neighbor to the goal.
    - Push the neighbor with its f-score onto the `open_list`.

Path Not Found:

- If the `open_list` becomes empty and the goal has not been reached, return `None` to indicate that no path was found.

Path Construction (if found):

- If a path is found (i.e., a goal is reached), construct the path by backtracking from the goal to the start using the `came_from` dictionary.
  - Return the path in reverse order.

SOURCE CODE:

```
import heapq

# Define a grid (0 represents empty, 1 represents obstacles)
grid = [
    [0, 0, 0, 0, 0],
    [0, 1, 1, 0, 0],
    [0, 0, 0, 0, 1],
    [0, 1, 1, 0, 0],
    [0, 0, 0, 0, 0]
]

# Define the start and goal positions
start = (0, 0)
goal = (4, 4)

# Define a heuristic function (Manhattan distance)
def heuristic(node, goal):
    return abs(node[0] - goal[0]) + abs(node[1] - goal[1])

# A* algorithm
def astar(grid, start, goal):
    open_list = [(0, start)]  # Priority queue (f-score, node)
    came_from = {}  # Dictionary to store the parent node of each node

    # Initialize g_score with all nodes set to infinity
    g_score = {(x, y): float('inf') for x in range(len(grid)) for y in range(len(grid[0]))}
    g_score[start] = 0

    f_score = {node: float('inf') for node in g_score}
    f_score[start] = heuristic(start, goal)

    while open_list:
        _, current = heapq.heappop(open_list)

        if current == goal:
            path = []
            while current in came_from:
                path.append(current)
```

```
            current = came_from[current]
        return path[::-1]

    for neighbor in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
        x, y = current[0] + neighbor[0], current[1] + neighbor[1]

        if 0 <= x < len(grid) and 0 <= y < len(grid[0]) and grid[x][y] == 0:
            tentative_g_score = g_score[current] + 1

            if tentative_g_score < g_score[(x, y)]:
                came_from[(x, y)] = current
                g_score[(x, y)] = tentative_g_score
                f_score[(x, y)] = tentative_g_score + heuristic((x, y), goal)
                heapq.heappush(open_list, (f_score[(x, y)], (x, y)))

    return None  # No path found

# Find the path
path = astar(grid, start, goal)

if path:
    print("Path found:")
    for node in path:
        print(node)
else:
    print("No path found.")
```

OUTPUT:

Path found:
(0, 1)
(0, 2)
(0, 3)
(1, 3)
(2, 3)
(3, 3)
(3, 4)
(4, 4)

Exercise 03 - Cryptarithmetic Problem

AIM:

To write a python program to solve cryptarithmetic problems.

ALGORITHM:

Input Parsing:

- Read the cryptarithmetic puzzle input from the user in the format 'WORD1 WORD2 WORD3'.
- Split the input into individual words and store them in the `words` list.
- Identify unique letters in the puzzle by concatenating all words and converting them into a set. These letters will be candidates for digit assignment.

Digit Assignment Permutations:

- Generate all permutations of digits from 0 to 9 for the unique letters in the puzzle.
- For each permutation, create a dictionary `letter_to_digit` that maps unique letters to digits based on the current permutation.

Word Digit Conversion:

- For each word in the puzzle, convert it to a corresponding number by replacing each letter with its corresponding digit according to the `letter_to_digit` dictionary.
- Store the converted numbers in the `digit_words` list.

Equation Evaluation:

- Check if the sum of the first two converted numbers (represented as integers) is equal to the third converted number (also represented as an integer).
- If the equation is satisfied, return the `letter_to_digit` dictionary, which represents a valid digit assignment.

Output Display (if solution is found):

- If a solution is found, print "Solution found:" and display the digit assignment, mapping each letter to its assigned digit.
- Iterate through the items in the `letter_to_digit` dictionary and print them.

No Solution Found (if solution is not found):

- If no valid digit assignment is found after checking all permutations, print "No solution found."

SOURCE CODE:

```
from itertools import permutations

def solve_cryptarithmetic(puzzle):
    words = puzzle.split()
    unique_letters = set("".join(words))
    digit_permutations = permutations("0123456789", len(unique_letters))
```

```
    for perm in digit_permutations:
            letter_to_digit = dict(zip(unique_letters, perm))
            digit_words = ["".join([letter_to_digit[letter] for letter in word]) for word in words]

            if (sum(int(i) for i in digit_words[:-1:]) == int(digit_words[2])):
                    return letter_to_digit
    return None

if __name__ == "__main__":
    puzzle = input("Enter the cryptarithmetic puzzle in the format 'WORD1  WORD2  WORD3':")
    solution = solve_cryptarithmetic(puzzle)

    if solution:
        print("Solution found:")
        for letter, digit in solution.items():
            print(f"{letter} = {digit}")
    else:
        print("No solution found.")
```

OUTPUT:

```
Enter the cryptarithmetic puzzle in the format 'WORD1  WORD2  WORD3':let lee all
Solution found:
t = 6
l = 1
e = 5
a = 3
```

Exercise 04 - Create a Knowledge Base of Facts and Convert them to Predicate Logic

AIM:

To create a friendly knowledge base system enabling addition, removal, verification, and conversion of facts into first-order logic statements in python.

ALGORITHM:

Class Definition:
- Define a class named `KnowledgeBase`.
- Initialize an empty set `facts` within the class to store facts.

Initialization:
- Implement the `_init_` method in the class to initialize the `facts` set.

Adding a Fact:
- Implement the `add_fact` method to add a fact to the `facts` set.

Removing a Fact:
- Implement the `remove_fact` method to remove a fact from the `facts` set.

Checking a Fact:
- Implement the `check_fact` method to check if a given fact exists in the `facts` set.

Displaying Facts:
- Implement the `display_facts` method to display all the facts stored in the `facts` set.

Converting to First-Order Logic:
- Implement the `to_first_order_logic` method to convert the stored facts into first-order logic statements.
- Split each fact into words and check if it follows the format: `[subject] [is/are] [object(s)]`.
- If yes, convert the fact into first-order logic format: `[predicate]([subject] [objects])`.

Whole program:
- Create an instance of the `KnowledgeBase` class.
- Accept facts from the user until the user types 'q'.
- Allow the user to check if a specific fact exists in the knowledge base.
- Display all facts stored in the knowledge base.
- Convert and display the facts in first-order logic format.

SOURCE CODE:

```
class KnowledgeBase:
```

```
    def _init_(self):
         self.facts = []

    def add_fact(self, fact):
         self.facts.add(fact)

    def remove_fact(self, fact):
         self.facts.discard(fact)

    def check_fact(self, fact):
         return fact in self.facts

    def display_facts(self):
      print("Facts in the Knowledge Base:")
      for fact in self.facts:
        print(fact)

    def to_first_order_logic(self):
       first_order_logic_facts=[]
       for fact in self.facts:
        words = fact.split()
        if len(words) >= 3 and words[1] in ["is", "are"]:
                subject = words[0]
                predicate = words[2]
                objects = words[3:]
        if len(objects) == 1:
            first_order_logic = f"{predicate}({subject} {objects[0]})"
        else:
            first_order_logic = f"{predicate}({subject} " + " ".join(objects) + ")"
         first_order_logic_facts.append(first_order_logic)
      return first_order_logic_facts


kb = KnowledgeBase()
```

```
# Accept facts from the user
print("Enter facts (one per line). Type 'q' to quit.")
while True:
    fact_str = input("Enter a fact: ")
    if fact_str.lower() == 'q':
        break
    kb.add_fact(fact_str)

# Check if a fact is in the knowledge base
check_fact_str = input("Enter a fact to check: ")
if  kb.check_fact(check_fact_str):
    print(f"'{check_fact_str}' is a fact in the knowledge base.")
else:
    print(f"'{check_fact_str}' is not a fact in the knowledge base.")


# Display all facts in the knowledge base
kb.display_facts()

# Convert facts to first-order logic and display
first_order_logic_facts = kb.to_first_order_logic()
print("\nFacts in First-Order Logic:")
for fact in first_order_logic_facts:
    print(fact)
```

SAMPLE I/O:

Enter facts (one per line). Type 'q' to quit.
Enter a fact: Sam is tall
Enter a fact: radha is cooking
Enter a fact: book is on table
Enter a fact: AI is the world
Enter a fact: q
Enter a fact to check: radha is cooking

'radha is cooking' is a fact in the knowledge base.

Facts in the Knowledge Base:
book is on table AI

is the world radha

is cooking

Sam is tall

Facts in First-Order Logic:
on(book table)

the(AI world)

cooking(radha)

tall(Sam)

Exercise 05 - Create a semantic network

AIM:

To create a user-driven semantic network for defining and visualizing relationships between nodes using a simple interactive interface in python.

ALGORITHM:

class `Node`:
Initialization function:
- Accepts a `name` parameter and initializes the node with the given name.
- Initializes an empty list called `edges` to store relationships with other nodes.

`add_edge(self, relation, node)` function:
- Accepts a `relation` and a `node`.
- Adds a tuple `(relation, node)` to the `edges` list, representing the relationship with the specified node.

`_str_(self)` function:
- Returns the string representation of the node (its name).

class `SemanticNetwork`:
Initialization function:
- Initializes an empty list called `nodes` to store nodes in the semantic network.

`add_node(self, name)` function:
- Accepts a `name` parameter.
- Creates a new node with the given name using the `Node` class.
- Adds the created node to the `nodes` list.
- Returns the created node.

`_str_(self)` function:
- Returns a string representation of the semantic network.
- Iterates through each node in the `nodes` list and prints its relationships with other nodes.

Whole program:
- Provides a menu-driven interface with options to:
- Add a new node to the semantic network.
- Add a relationship between nodes.
- Exit the program.

SOURCE CODE:

```
class Node:
    def _init_(self, name):
        self.name = name
```

```python
        self.edges = []

    def add_edge(self, relation, node):
        self.edges.append((relation, node))

    def _str_(self):
        return self.name

class SemanticNetwork:
    def _init_(self):
        self.nodes = []

    def add_node(self, name):
        node = Node(name)
        self.nodes.append(node)
        return node

    def _str_(self):
        result = "Semantic Network:\n"
        for node in self.nodes:
            result += f"{node} has relations:\n"
            for relation, related_node in node.edges:
                result += f"  - {relation}: {related_node}\n"
        return result

# Create a semantic network
semantic_net = SemanticNetwork()

while True:
    print("1. Add Node")
    print("2. Add Relationship")
    print("3. Exit")
    choice = input("Enter your choice: ")

    if choice == '1':
        node_name = input("Enter node name: ")
        semantic_net.add_node(node_name)
        print(f"Node '{node_name}' added.")
    elif choice == '2':
        node_name = input("Enter node name: ")
        relation = input("Enter relation: ")
        related_node_name = input("Enter related node name: ")
        node = next((n for n in semantic_net.nodes if n.name == node_name), None)
        related_node = next((n for n in semantic_net.nodes if n.name == related_node_name), None)
```

```
        if node and related_node:
            node.add_edge(relation, related_node)
            print(f"Relationship added: {node_name} -> {relation} -> {related_node_name}")
        else:
            print("One or both nodes not found.")
    elif choice == '3':
        break
    else:
        print("Invalid choice. Please try again.")

# Display the semantic network
print(semantic_net)
```

SAMPLE I/O:

1. Add Node
2. Add Relationship
3. Exit

Enter your choice: 1
Enter node name: rrr Node

'rrr' added.

1. Add Node
2. Add Relationship 3. Exit

Enter your choice: 1
Enter node name: ttt Node

'ttt' added.

1. Add Node
2. Add Relationship
3. Exit

Enter your choice: 2
Enter node name: rrr
Enter relation: isa
Enter related node name: ttt
Relationship added: rrr -> isa -> ttt

1. Add Node
2. Add Relationship
3. Exit Enter your choice: 3 Semantic Network:

rrr has relations:
  - isa: ttt
ttt has relations:

Exercise 06 - Calculate conditional probability using– Naïve Bayes theorem

AIM:

To implement a Naïve Bayes Classifier to classify text messages as "spam" or "ham" based on word frequencies.

ALGORITHM:

Preprocessing and Vocabulary Creation:
- Import the `re` module for regular expressions and `numpy` as `np`.
- Define sample training data as a list of tuples, each containing a text message and its corresponding label ("spam" or "ham").
- Tokenize words from training data and create a vocabulary of unique words.
- Initialize counters for spam and ham messages.

Word Counting:
- Initialize two NumPy arrays (`spam_word_count` and `ham_word_count`) filled with zeros to count word occurrences in spam and ham messages.
- Iterate through the training data, tokenize text, and update word counts in the corresponding arrays.

Prior Probability Calculation:
- Calculate prior probabilities for spam and ham messages based on the counts and total number of messages.

Input Text Processing:
- Input a text message to classify.
- Tokenize and process the input text similar to training data.

Likelihood Calculation:
- Calculate likelihoods for spam and ham messages using the Naïve Bayes formula:
- Multiply individual word likelihoods for each class.
- Use Laplace smoothing by adding 1 to the numerator and `(spam_count + len(vocab))` to the denominator for each word in likelihood calculation.

Bayes' Theorem and Posterior Probabilities:
- Apply Bayes' theorem to calculate posterior probabilities for both spam and ham messages.
- Calculate posterior probabilities using likelihoods and prior probabilities.

Classification Decision:
- Make a classification decision based on posterior probabilities:
- If `posterior_spam > posterior_ham`, classify the input text as "Spam." - Otherwise, classify it as "Ham."

SOURCE CODE:

```
import re   # In Python, the import re statement is used to import the re module, which stands for "regular expressions."
import numpy as np

# Sample training data
data = [
   ("This is a spam email", "spam"),
   ("Buy one get one free", "spam"),
   ("Hello, how are you?", "ham"),
   ("Congratulations, you've won a prize!", "spam"),
   ("Meeting at 3 PM", "ham"),
   ("Get a discount on your next purchase", "spam"),
]

# Preprocess the training data
word_set = set()
for text, label in data:
   words = re.findall(r'\w+', text.lower())
word_set.update(words)
word_list = list(word_set)
word_list.sort()


# Create a vocabulary
vocab = {word: index for index, word in enumerate(word_list)}


# Initialize counts for spam and ham
spam_count = 0 ham_count = 0


# Count the occurrences of words in spam and ham messages
spam_word_count = np.zeros(len(vocab))   # creating a NumPy array named spam_word_count filled with zeros.
ham_word_count = np.zeros(len(vocab))

# Populate the counts
for text, label in data:
        words = re.findall(r'\w+', text.lower())   # used to tokenize a given text into words
```

```
    label_count = spam_word_count if label == 'spam' else ham_word_count
        for word in words:
            if word in vocab:
                word_index = vocab[word]
                label_count[word_index] += 1

# Calculate the prior probabilities
total_messages = len(data)
prior_spam = spam_count / total_messages
prior_ham = ham_count / total_messages

# Input text to classify
input_text = "You've won a free vacation!"

# Tokenize and process the input
text input_words = re.findall(
    r'\w+',
    input_text.lower()
)

# Calculate likelihoods and apply the Naive Bayes formula
likelihood_spam = 1.0
likelihood_ham = 1.0

for word in input_words:
  if word in vocab:
    word_index = vocab[word]
    likelihood_spam *= (spam_word_count[word_index] + 1) / (spam_count +
    len(vocab))
    likelihood_ham *= (ham_word_count[word_index] + 1) / (ham_count + len(vocab))

# Apply Bayes' theorem
```

```
posterior_spam = (likelihood_spam * prior_spam) / ((likelihood_spam * prior_spam) + (likelihood_ham *
prior_ham))
posterior_ham = 1 - posterior_spam

# Make a classification decision
if posterior_spam > posterior_ham:
    print("Classified as: Spam")
else:
    print("Classified as: Ham")
```

OUTPUT:

Classified as: Spam

Exercise 07 - Game Playing
a)MIN-MAX ALGORITHM

AIM:
To write a python program to implement a minimax algorithm.

ALGORITHM:
Initialize the Tic-Tac-Toe Board:
- Create a list `board` to represent the 3x3 game board, with empty spaces initially.

Print the Board:
- Create a function `print_board()` to display the current state of the board on the console.

Check if the Board is Full:
- Create a function `is_full(board)` that checks if there are no empty spaces left on the board.

Check for a Winner:
- Create a function `is_winner(board, player)` that checks if the specified player ('X' or 'O') has won by checking rows, columns, and diagonals.

Minimax Algorithm:
- Create a function `minimax(board, depth, is_maximizing)` that uses the Minimax algorithm to determine the best move for the computer ('X') by evaluating potential future game states.
- The Minimax algorithm is a recursive algorithm that evaluates game states by assigning scores and selecting the move with the highest score for 'X' and the lowest score for 'O'.
- The algorithm considers different game states, checks for a win, loss, or a tie, and assigns scores accordingly.
- It iterates through all possible moves and recursively calls itself to find the best move.

Find the Best Move:
- Create a function `find_best_move(board)` that uses the Minimax algorithm to find the best move for 'X'.

Main Game Loop:
- Start the main game loop:
  - Print the current state of the board.
  - Take input from the player for their move (0-8).
  - Check if the move is valid (an empty space).
  - Place the player's move ('O') on the board.
  - Check if the player has won.
  - Check if the board is full (a tie).
  - If the game is not over, use the Minimax algorithm to find the best move for the computer ('X').

- Place the computer's move on the board.
- Check if the computer has won.
- Check if the board is full.
- Repeat the loop until there is a winner or a tie.

Game Outcome:
- Depending on the outcome (player win, computer win, or tie), display a corresponding message.

SOURCE CODE:

```
# Tic-Tac-Toe Board
board = [' '  for _ in range(9)]

# Function to print the board
def print_board():
   print(f"{board[0]} | {board[1]} | {board[2]}")
   print("---------")
   print(f"{board[3]} | {board[4]} | {board[5]}")
   print("---------")
   print(f"{board[6]} | {board[7]} | {board[8]}")

# Function to check if the board is full
def is_full(board):
   return ' ' not in board

# Function to check if a player has won
def is_winner(board, player):
   # Check rows
   for i in range(0, 9, 3):
      if board[i] == board[i + 1] == board[i + 2] == player:
         return True
   # Check columns
   for i in range(3):
      if board[i] == board[i + 3] == board[i + 6] == player:
         return True
   # Check diagonals
   if board[0] == board[4] == board[8] == player:
      return True
   if board[2] == board[4] == board[6] == player:
      return True
   return False
```

```python
# Min-Max algorithm
def minimax(board, depth, is_maximizing):
    scores = {
        'X': 1,
        'O': -1,
        'Tie': 0,
    }

    if is_winner(board, 'X'):
        return scores['X'] - depth
    if is_winner(board, 'O'):
        return scores['O'] + depth
    if is_full(board):
        return scores['Tie']

    if is_maximizing:
        best_score = float('-inf')
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'X'
                score = minimax(board, depth + 1, False)
                board[i] = ' '
                best_score = max(score, best_score)
        return best_score
    else:
        best_score = float('inf')
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'O'
                score = minimax(board, depth + 1, True)
                board[i] = ' '
                best_score = min(score, best_score)
        return best_score

# Function to find the best move
def find_best_move(board):
    best_move = -1
    best_score = float('-inf')
    for i in range(9):
        if board[i] == ' ':
            board[i] = 'X'
            score = minimax(board, 0, False)
            board[i] = ' '
            if score > best_score:
```

```
            best_score = score
            best_move = i
    return best_move

# Main game loop
while True:
    print_board()
    move = int(input("Enter your move (0-8): "))

    if board[move] != ' ':
        print("Invalid move. Try again.")
        continue

    board[move] = 'O'

    if is_winner(board, 'O'):
        print_board()
        print("You win!")
        break

    if is_full(board):
        print_board()
        print("It's a tie!")
        break

    best_move = find_best_move(board)
    board[best_move] = 'X'

    if is_winner(board, 'X'):
        print_board()
        print("Computer wins!")
        break

    if is_full(board):
        print_board()
        print("It's a tie!")
        break
```

SAMPLE I/O:

```
  |   |
---------
  |   |
---------
  |   |
Enter your move (0-8): 4
X |   |
---------
  | O |
---------
  |   |
Enter your move (0-8): 2
X | X | O
---------
  | O |
---------
  |   |
Enter your move (0-8): 6
X | X | O
---------
  | O |
---------
O |   |
You win!
```

b) ALPHA-BETA PRUNING

AIM:

To write a python program to implement a minimax algorithm with alpha-beta pruning.

ALGORITHM:

Define a TreeNode Class:
- Create a class `TreeNode` with the following attributes:
  - `score`: The score associated with the node.
  - `children`: A list of child nodes.

Build a Tree with Scores:
- Create a tree structure with nodes representing game states or positions.
- Assign scores to the nodes to represent the quality or desirability of those game states.
- Establish parent-child relationships among the nodes by assigning child nodes to their respective parent nodes.

Define the Alpha-Beta Pruning Function:
- Create a function `alpha_beta(node, depth, alpha, beta, is_maximizing)` that performs the Alpha-Beta Pruning algorithm. This function will be a recursive search of the game tree.
  - `node`: The current node being evaluated.
  - `depth`: The maximum depth to search in the game tree.
  - `alpha`: The best score found for the maximizing player (initially negative infinity).
  - `beta`: The best score found for the minimizing player (initially positive infinity).
  - `is_maximizing`: A boolean indicating whether the current node represents a maximizing player's turn.

Base Case (Leaf Node or Maximum Depth Reached):
- If `depth` is 0 or the `node` has no children, return the `node`'s `score` (evaluated value).

Maximizing Player's Turn:
- If `is_maximizing` is `True`, indicating the maximizing player's turn:
  - Initialize `max_eval` to negative infinity (worst possible value for the maximizing player).
  - Iterate through the `node`'s children:
    - Recursively call `alpha_beta` on each child with `depth - 1`, `alpha`, and `beta` while switching to the minimizing player's turn (`is_maximizing = False`).
    - Update `max_eval` to be the maximum of the current `max_eval` and the evaluated value.
    - Update `alpha` to be the maximum of the current `alpha` and the evaluated value.
    - Perform alpha-beta pruning: If `beta` is less than or equal to `alpha`, break out of the loop (no need to consider other child nodes).
  - Return `max_eval` as the best score for the maximizing player.

Minimizing Player's Turn:
- If `is_maximizing` is `False`, indicating the minimizing player's turn:
  - Initialize `min_eval` to positive infinity (worst possible value for the minimizing player).

- Iterate through the `node`'s children:
  - Recursively call `alpha_beta` on each child with `depth - 1`, `alpha`, and `beta` while switching to the maximizing player's turn (`is_maximizing = True`).
  - Update `min_eval` to be the minimum of the current `min_eval` and the evaluated value.
  - Update `beta` to be the minimum of the current `beta` and the evaluated value.
  - Perform alpha-beta pruning: If `beta` is less than or equal to `alpha`, break out of the loop (no need to consider other child nodes).
- Return `min_eval` as the best score for the minimizing player.

Main Game Loop:
- In the provided example usage (if `__name__ == "__main__":`), call the `alpha_beta` function on the `root` node with specified parameters (`root`, `3`, `float('-inf')`, `float('inf')`, `True`) to find the optimal value of the game tree.
- Print the optimal value found by the Alpha-Beta Pruning algorithm.

SOURCE CODE:

```
class TreeNode:
    def __init__(self, score):
        self.score = score
        self.children = []

# Build a tree with scores at each node
root = TreeNode(2)
root.children = [TreeNode(7), TreeNode(5), TreeNode(4)]

root.children[0].children = [TreeNode(3), TreeNode(8), TreeNode(3)]
root.children[1].children = [TreeNode(1), TreeNode(2), TreeNode(6)]
root.children[2].children = [TreeNode(2), TreeNode(4), TreeNode(7)]

# Define the alpha-beta pruning function
def alpha_beta(node, depth, alpha, beta, is_maximizing):
    if depth == 0 or not node.children:
        return node.score

    if is_maximizing:
        max_eval = float('-inf')
        for child in node.children:
            eval = alpha_beta(child, depth - 1, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval
    else:
```

```
        min_eval = float('inf')
        for child in node.children:
            eval = alpha_beta(child, depth - 1, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval

if __name__ == "__main__":
    result = alpha_beta(root, 3, float('-inf'), float('inf'), True)
    print("Optimal value:", result)
```

OUTPUT:

```
Optimal value: 3
```

Exercise 08 - NATURAL LANGUAGE PROCESSING

Tokenization and stemming using NLTK

AIM:

To write a python program to Tokenize and stem words or sentences using NLTK

ALGORITHM:

Algorithm: NLP Processing using NLTK

Download NLTK Resources:
- Check if the NLTK resources 'punkt' (for tokenization) and 'stopwords' (for stop words) are already downloaded. If not, download them using `nltk.download()`.

Sample Text:
- Define a sample text for processing, which is a string containing natural language text.

Tokenization:
- Tokenization is the process of splitting the text into individual words or tokens.
- Use the `word_tokenize` function from NLTK to tokenize the text.
- Store the resulting tokens in a list.

Stopwords Removal:
- Stopwords are common words (e.g., "the," "and," "is") that are often removed from text as they do not carry significant meaning.
- Create a set of English stop words using `stopwords.words('english')` from NLTK.
- Filter the tokens to remove any words that are present in the stopwords set.
- Store the filtered tokens in a new list.

Stemming:
- Stemming is the process of reducing words to their base or root form (e.g., "running" to "run").
- Create a stemmer object, in this case, `PorterStemmer`, using `PorterStemmer()`.
- Apply stemming to the filtered tokens using the stemmer's `stem` method.
- Store the stemmed tokens in a new list.

Display the Results:
- Print the original text, tokenized text, text after stopwords removal, and text after stemming to the console.

SOURCE CODE:

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
```

```
from nltk.stem import PorterStemmer

# Download NLTK resources if not already downloaded
nltk.download('punkt')
nltk.download('stopwords')

# Sample text for processing
text = "Natural language processing (NLP) is a subfield of artificial intelligence (AI) that deals with the interaction between computers and humans through natural language."

# Tokenization
tokens = word_tokenize(text)

# Stopwords removal
stop_words = set(stopwords.words('english'))
filtered_tokens = [word for word in tokens if word.lower() not in stop_words]

# Stemming
stemmer = PorterStemmer()
stemmed_tokens = [stemmer.stem(word) for word in filtered_tokens]

# Print the results
print("Original Text:")
print(text)

print("\nTokenized Text:")
print(tokens)

print("\nAfter Stop Words Removal:")
print(filtered_tokens)

print("\nAfter Stemming:")
print(stemmed_tokens)
```

OUTPUT:

```
Original Text:
Natural language processing (NLP) is a subfield of artificial intelligence (AI) that deals with the interaction between computers and humans through natural language.

Tokenized Text:
['Natural', 'language', 'processing', '(', 'NLP', ')', 'is', 'a', 'subfield', 'of', 'artificial', 'intelligence', '(', 'AI', ')', 'that', 'deals', 'with', 'the', 'interaction', 'between', 'computers', 'and', 'humans', 'through', 'natural', 'language', '.']
```

After Stop Words Removal:
['Natural', 'language', 'processing', '(', 'NLP', ')', 'subfield', 'artificial', 'intelligence', '(', 'AI', ')', 'deals', 'interaction', 'computers', 'humans', 'natural', 'language', '.']

After Stemming:
['natur', 'languag', 'process', '(', 'nlp', ')', 'subfield', 'artifici', 'intellig', '(', 'ai', ')', 'deal', 'interact', 'comput', 'human', 'natur', 'languag', '.']

Exercise 09 - Spell Checking using NLTK

AIM:

To write a python program to check the spelling of a word using NLTK.

ALGORITHM:

Download NLTK Resources:
- Check if the NLTK word corpus is already downloaded. If not, download it using `nltk.download()`.

Load English Words Corpus:
- Load the NLTK word corpus as a set of English words.
- This corpus contains a comprehensive list of English words.

Define the Spell Check Function:
- Create a function `spell_check(text)` that takes the input text to be spell-checked.
- Tokenize the input text into individual words using `nltk.word_tokenize()`.

Spell Checking:
- For each word in the tokenized text, check if the lowercase version of the word is in the set of English words.
- Create a list of misspelled words, which are the words not found in the set of English words.

User Input:
- Prompt the user to enter a sentence for spell checking.
- Accept the input sentence.

Perform Spell Checking:
- Call the `spell_check` function with the user's input sentence to identify misspelled words.

Display Results:
- If misspelled words are found, print a list of those words, indicating that they are misspelled.
- If no misspelled words are found, print a message indicating that no misspelled words were detected.

End of Algorithm:
- The spell-checking process is complete.

SOURCE CODE:

```
import nltk
from nltk.corpus import words

nltk.download('words')
```

```
# Load the NLTK words corpus as a set
english_words = set(words.words())

def spell_check(text):
    # Tokenize the input text
    words_to_check = nltk.word_tokenize(text)

    # Check each word in the text against the NLTK English words corpus
    misspelled_words = [word for word in words_to_check if word.lower() not in english_words]

    return misspelled_words

# Input from the user
input_text = input("Enter a sentence for spell checking: ")

# Perform spell checking
misspelled_words = spell_check(input_text)

if len(misspelled_words) > 0:
    print("\nMisspelled words:")
    for word in misspelled_words:
        print(word)
else:
    print("\nNo misspelled words found.")
```

SAMPLE I/O:

```
 Enter a sentence for spell checking: Thes is a smaple sentnce with soem mitsakes.

Misspelled words:
Thes
smaple
sentnce
soem
mitsakes

Enter a sentence for spell checking: how are you

No misspelled words found.
```

Exercise 10 - Developing a Simple Medical Expert System

AIM:

To write a python program to develop a simple Medical Expert System.

ALGORITHM:

Algorithm: Medical Expert System

Define the `MedicalExpertSystem` Class:
- Define a class `MedicalExpertSystem` with the following attributes and methods:
  - Attributes:
    - `symptoms`: A list to store the patient's reported symptoms.
    - `diagnosis`: A variable to store the system's diagnosis.
    - `patient_data`: A pandas DataFrame to store patient data (symptoms and diagnosis).
  - Methods:
    - `__init__()`: The constructor initializes the attributes.
    - `ask_question(question)`: Prompts the user with a yes/no question and returns the user's response.
    - `diagnose()`: Asks the patient about specific symptoms, stores the symptoms, and provides a diagnosis.
    - `run()`: Executes the medical expert system, providing the diagnosis and displaying patient data.

Initialize the System:
- Create an instance of the `MedicalExpertSystem` class, such as `expert_system`.

Run the System:
- In the provided example usage, if `__name__ == "__main__":`, the system runs by calling the `run` method on the `expert_system` instance.

Welcome Message:
- Display a welcome message to the user.

Diagnosis Process:
- Call the `diagnose` method to initiate the diagnosis process.
- In the `diagnose` method, ask the patient a series of yes/no questions about symptoms, such as fever, headache, and cough.
- If the patient responds "yes" to a symptom question, add the symptom to the `symptoms` list.
- Create a new row in the `patient_data` DataFrame to store the symptoms and an empty diagnosis.
- Determine the diagnosis based on the reported symptoms and update the `diagnosis` attribute.
- Update the corresponding row in the `patient_data` DataFrame with the diagnosis.

Display Diagnosis and Patient Data:
- Print the diagnosis based on the reported symptoms.
- Display the patient data, including symptoms and diagnosis, stored in the `patient_data` DataFrame.

SOURCE CODE:

```
import pandas as pd
class MedicalExpertSystem:
    def _init_(self):
        self.symptoms = []
        self.diagnosis = None
        self.patient_data = pd.DataFrame(columns=["Symptoms", "Diagnosis"])

    def ask_question(self, question):
        answer = input(question + " (yes/no): ").strip().lower()
        if answer == "yes":
            return True
        elif answer == "no":
            return False
        else:
            print("Invalid input. Please answer with 'yes' or 'no'.")
            return self.ask_question(question)

    def diagnose(self):
        if self.ask_question("Do you have a fever?"):
            self.symptoms.append("fever")
        if self.ask_question("Do you have a headache?"):
            self.symptoms.append("headache")
        if self.ask_question("Do you have a cough?"):
            self.symptoms.append("cough")

        self.patient_data = self.patient_data.append({"Symptoms": ", ".join(self.symptoms), "Diagnosis": ""},
ignore_index=True)

        if "fever" in self.symptoms and "headache" in self.symptoms:
            self.diagnosis = "You might have the flu."
        elif "fever" in self.symptoms and "cough" in self.symptoms:
            self.diagnosis = "You might have a cold."
        else:
            self.diagnosis = "Your condition is unclear. Please consult a doctor."

        self.patient_data.loc[self.patient_data.index[-1], "Diagnosis"] = self.diagnosis

    def run(self):
        print("Welcome to the Medical Expert System.")
        self.diagnose()
        print("Diagnosis:", self.diagnosis)
        print("Patient Data:")
        print(self.patient_data)
```

```
if _name_ == "_main_":
    expert_system = MedicalExpertSystem()
    expert_system.run()
```

OUTPUT:

```
Welcome to the Medical Expert System.

Do you have a fever? (yes/no): yes

Do you have a headache? (yes/no): yes

Do you have a cough? (yes/no): yes

Diagnosis: You might have the flu.

Patient Data:

             Symptoms             Diagnosis

0  fever, headache, cough  You might have the flu.
```