Practical 01

Aim: Program to Simulate an agent

1] Simple Reflex Agent

Code:

```
# Simple Reflex Agent for Noise Level
class SimpleReflexAgent:
    def __init__(self, threshold=70):
        self.threshold = threshold    # Noise level threshold in dB

    def act(self, noise_level):
        if noise_level > self.threshold:
            print(f"Noise Level = {noise_level}dB → ALERT (Possible Intruder)")
        else:
            print(f" Noise Level = {noise_level}dB → Safe")

# Run the agent
noise = int(input("Enter the noise level (in dB): "))
agent = SimpleReflexAgent()
agent.act(noise)
```

Output:

```
Enter the noise level (in dB): 80
Noise Level = 80dB → ALERT (Possible Intruder)
```

2] Model Based Agent

Code:

```
class ModelBasedAgent:
    def __init__(self, threshold=70):
        self.threshold = threshold

    def act(self, noise_level):
        if noise_level > self.threshold:
            print(f" Noise = {noise_level}dB → ALERT (Possible Intrusion)")
        else:
            print(f" Noise = {noise_level}dB → Safe")

# Run the agent
agent = ModelBasedAgent()
for _ in range(5):
    noise = int(input("Enter the noise level (in dB): "))
    agent.act(noise)
```

Output:

```
Enter the noise level (in dB): 200
```

```
Noise = 200dB → ALERT    (Possible Intrusion)
Enter the noise level (in dB): 10
Noise = 10dB → Safe
Enter the noise level (in dB): 40
Noise = 40dB → Safe
Enter the noise level (in dB): 30
Noise = 30dB → Safe
Enter the noise level (in dB): 80
Noise = 80dB → ALERT (Possible Intrusion)
```

3] Goal Based Agent

Code:

```
# Goal-Based Agent for Motion Detection
class GoalBasedAgent:
    def __init__(self, goal="Keep noise level low", threshold=70):
        self.goal = goal
        self.threshold = threshold

    def act(self, noise_level):
        if noise_level > self.threshold:
            print(f" Noise = {noise_level}dB → Locking Doors & Notifying Security ")
        else:
            print(f" Noise = {noise_level}dB → Home is Secure ")

# Run the agent
noise = int(input("Enter the noise level (in dB): "))
agent = GoalBasedAgent()
agent.act(noise)
```

Output:

```
Enter the noise level (in dB): 80
Noise = 80dB → Locking Doors & Notifying Security
```

4] Utility Based Agent

Code:

```
class UtilityBasedAgent:
    def __init__(self):
        pass

    def calculate_utility(self, noise_level):
        if noise_level < 50:
            return 10    # Very safe
        elif 50 <= noise_level <= 70:
            return 5    # Warning needed
        else:
            return 0    # High danger, immediate action

    def act(self, noise_level):
```

```
            utility = self.calculate_utility(noise_level)
            if utility == 10:
                print(f== {noise_level}dB (Utility: {utility}) → No Action Needed ")
            elif utility == 5:
                print(f" Noise = {noise_level}dB (Utility: {utility}) → Warning: Possible
 Disturbance ")
            else:
                print(f"Noise = {noise_level}dB (Utility: {utility}) → ALERT! Locking Doors &
 Notifying")

 # Run the agent
 noise = int(input("Enter the noise level (in dB): "))
 agent = UtilityBasedAgent()
 agent.act(noise)
```

Output:

Enter the noise level (in dB): 100
Noise = 100dB (Utility: 0) → ALERT! Locking Doors & Notifying

Practical 2

Implementation of Uninformed Search Technique (Breadth first Search)

Code:

```
from collections import deque

# Function to find the shortest path using BFS
def bfs_shortest_path(graph, start, end):
    queue = deque([(start, [start])])    # Store (current location, path)
    visited = set()

    while queue:
        current, path = queue.popleft()

        if current == end:
            return path    # Return shortest path found

        if current not in visited:
            visited.add(current)

            for neighbor in graph.get(current, []):
                if neighbor not in visited:
                    queue.append((neighbor, path + [neighbor]))

    return "No path found"

# Example road network (Graph representation)
road_map = {
    "A": ["B", "C"],
    "B": ["D"],
    "C": ["D", "F"],
    "D": ["E"],
    "F": ["E"]
}

# Collect all unique locations (both keys and values)
all_locations = set(road_map.keys())
for neighbors in road_map.values():
    all_locations.update(neighbors)

# Take user input for start and end locations
start_location = input("Enter the start location: ").strip().upper()
```

```
end_location = input("Enter the end location: ").strip().upper()

# Check if both locations exist in the graph
if start_location in all_locations and end_location in all_locations:
    shortest_path = bfs_shortest_path(road_map, start_location, end_location)
    print(f"Shortest path from {start_location} to {end_location}: {shortest_path}")
else:
    print("Invalid locations! Please enter valid points from the road map.")
```

Output:

```
Enter the start location: A
Enter the end location: E
Shortest path from A to E: ['A', 'B', 'D', 'E']
```

Cour
Nam
Nam
Roll
Batc
Date
Title
Cou
Sr. No.
1
2
3

Practical 3

Code:

```
import heapq

class Graph:
    def __init__(self):
        self.graph, self.heuristics = {}, {}

    def add_edge(self, u, v, cost):
        self.graph.setdefault(u, []).append((v, cost))
        self.graph.setdefault(v, []).append((u, cost))

    def set_heuristic(self, node, value):
        self.heuristics[node] = value

    def a_star_search(self, start, goal):
        open_list = [(0, start)]
        came_from, g_score = {}, {node: float('inf') for node in self.graph}
        g_score[start] = 0

        while open_list:
            _, current = heapq.heappop(open_list)
            if current == goal:wo
                path = []
                while current in came_from:
                    path.append(current)
                    current = came_from[current]
                return [start] + path[::-1]

            for neighbor, cost in self.graph[current]:
                new_g = g_score[current] + cost
                if new_g < g_score[neighbor]:
                    came_from[neighbor], g_score[neighbor] = current, new_g
                    heapq.heappush(open_list, (new_g + self.heuristics.get(neighbor, 0),
neighbor))

        return None  # No path found

def create_graph():
    g = Graph()
    for _ in range(int(input("Edges count: "))):
        u, v, cost = input().split()
        g.add_edge(u, v, int(cost))
```

```
        for _ in range(int(input("Heuristic count: "))):
            node, h = input().split()
            g.set_heuristic(node, int(h))
        return g

if __name__ == "__main__":
    graph = create_graph()
    path = graph.a_star_search(input("Start: "), input("Goal: "))
    print("Path:", " -> ".join(path) if path else "No path found")
```

Output:

```
Edges count: 6
A B 4
A C 3
B D 2
C D 5
B E 10
D E 3
Heuristic count: 5
A 7
B 6
C 4
D 2
E 0
Start: A
Goal: E
Path: A -> B -> D -> E
```

Practical 4

Code:

```
import random
import numpy as np

class RoomHeatingProblem:
    def __init__(self, grid_size, num_heaters, num_obstacles):
        self.grid_size, self.num_heaters, self.num_obstacles = grid_size, num_heaters, num_obstacles
        self.obstacles = self.random_obstacles() # Initialize obstacles first
        self.state = self.random_initial_state() # Then generate heaters

    def random_initial_state(self):
        heaters = set()
        while len(heaters) < self.num_heaters:
            pos = (random.randint(0, self.grid_size-1), random.randint(0, self.grid_size-1))
            if pos not in self.obstacles: heaters.add(pos)
        return heaters

    def random_obstacles(self):
        obstacles = set()
        while len(obstacles) < self.num_obstacles:
            pos = (random.randint(0, self.grid_size-1), random.randint(0, self.grid_size-1))
            obstacles.add(pos)
        return obstacles

    def evaluate(self, state):
        temp_grid = np.zeros((self.grid_size, self.grid_size))
        for x, y in state:
            for i in range(self.grid_size):
                for j in range(self.grid_size):
                    if (i, j) not in self.obstacles:
                        temp_grid[i][j] += 1 / (np.sqrt((x - i)**2 + (y - j)**2) + 1)
        return np.var(temp_grid)

    def get_neighbors(self, state):
        neighbors = []
        for (x, y) in state:
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                new_x, new_y = x + dx, y + dy
                if 0 <= new_x < self.grid_size and 0 <= new_y < self.grid_size and (new_x, new_y) not in
self.obstacles:
                    # Create a new state by moving a heater
                    new_state = state - {(x, y)} | {(new_x, new_y)}
```

```
            neighbors.append(new_state)
        return neighbors

class HillClimbing:
    def __init__(self, problem):
        self.problem = problem

    def hill_climb(self):
        current_state, current_value = self.problem.state, self.problem.evaluate(self.problem.state)
        while True:
            neighbors = self.problem.get_neighbors(current_state)
            if not neighbors: break
            next_state = min(neighbors, key=self.problem.evaluate)
            next_value = self.problem.evaluate(next_state)
            if next_value >= current_value:
                break
            current_state = next_state
            current_value = next_value
        return current_state, current_value

def display_room(problem, best_state):
    grid = [['.' for _ in range(problem.grid_size)] for _ in range(problem.grid_size)]
    for (x, y) in problem.obstacles: grid[x][y] = 'X'
    for (x, y) in best_state: grid[x][y] = 'H'
    for row in grid: print(' '.join(row))

def run_dynamic_hill_climbing(grid_size, num_heaters, num_obstacles):
    problem = RoomHeatingProblem(grid_size, num_heaters, num_obstacles)
    hc = HillClimbing(problem)
    best_state, best_value = hc.hill_climb()
    print(f"\nBest Heater Positions: {best_state}")
    print(f"Objective Value (Temperature Variance): {best_value}")
    print("\nRoom Layout:")
    display_room(problem, best_state)

# Input for grid size, heaters, obstacles
grid_size = int(input("Enter grid size: "))
num_heaters = int(input("Enter number of heaters: "))
num_obstacles = int(input("Enter number of obstacles: "))

run_dynamic_hill_climbing(grid_size, num_heaters, num_obstacles)
```

Output:

```
Enter grid
Enter numb
Enter numb

Best Heat
Objectiv

Room Lay
H . . .
X . . .
X . . .
. . . X
X . . .
```

```
Enter grid size: 5
Enter number of heaters: 3
Enter number of obstacles: 4

Best Heater Positions: {(4, 4), (0, 4), (0, 0)}
Objective Value (Temperature Variance): 0.14597820105231082

Room Layout:
H . . . H
X . . . .
X . . . .
. . . X .
X . . . H
```

Practical 5

Code:

```
import math

def print_board(board):
  for row in board: print(" | ".join(row)); print()

def check_winner(board):
  for i in range(3):
    if board[i][0] == board[i][1] == board[i][2] != " ": return board[i][0]
    if board[0][i] == board[1][i] == board[2][i] != " ": return board[0][i]
  if board[0][0] == board[1][1] == board[2][2] != " " or board[0][2] == board[1][1] == board[2][0] != " ":
    return board[1][1]
  return None

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

def minimax(board, is_max):
  winner = check_winner(board)
  if winner: return {"X": -10, "O": 10}.get(winner, 0)
  if is_full(board): return 0

  best_score = -math.inf if is_max else math.inf
  for i in range(3):
    for j in range(3):
      if board[i][j] == " ":
        board[i][j] = "O" if is_max else "X"
        score = minimax(board, not is_max)
        board[i][j] = " "
        best_score = max(best_score, score) if is_max else min(best_score, score)
  return best_score

def best_move(board):
  move, best_score = (-1, -1), -math.inf
  for i in range(3):
    for j in range(3):
      if board[i][j] == " ":
        board[i][j] = "O"
        score = minimax(board, False)
        board[i][j] = " "
        if score > best_score: best_score, move = score, (i, j)
  return move

def play_game():
```

```
board = [[" "] * 3 for _ in range(3)]
print("Tic-Tac-Toe! You: 'X', AI: 'O'.")

while True:
  print_board(board)
  row, col = map(int, input("Enter move (0-2 0-2): ").split())
  if board[row][col] != " ": print("Invalid! Try again."); continue
  board[row][col] = "X"

  if check_winner(board): print_board(board); print("You win!"); break
  if is_full(board): print_board(board); print("It's a draw!"); break

  ai_row, ai_col = best_move(board)
  board[ai_row][ai_col] = "O"
  print(f"AI plays at ({ai_row}, {ai_col})")

  if check_winner(board): print_board(board); print("AI wins!"); break

play_game()
```

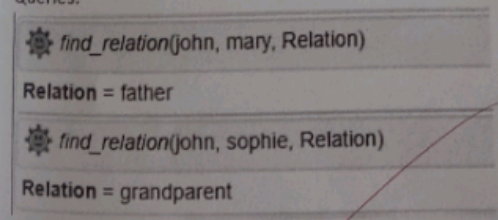
Output:

```
Tic-Tac-Toe! You: 'X', AI: 'O'.
  |   |

  |   |

  |   |

Enter move (0-2 0-2): 0 0
AI plays at (1, 1)
X |   |

  | O |

  |   |

Enter move (0-2 0-2): 0 1
AI plays at (0, 2)
X | X | O

  | O |

  |   |

Enter move (0-2 0-2): 1 2
AI plays at (2, 0)
X | X | O

  | O | X

O |   |

AI wins!
```

Practical 6

```
% Facts: Family relationships
parent(john, mary).
parent(john, mike).
parent(mary, sophie).
parent(mike, tom).

% Gender facts
male(john).
male(mike).
male(tom).
female(mary).
female(sophie).

% Rules: Defining relationships
father(F, C) :- parent(F, C), male(F).
mother(M, C) :- parent(M, C), female(M).
sibling(X, Y) :- parent(P, X), parent(P, Y), X \= Y.
grandparent(GP, GC) :- parent(GP, P), parent(P, GC).
grandfather(GF, GC) :- grandparent(GF, GC), male(GF).
grandmother(GM, GC) :- grandparent(GM, GC), female(GM).

% Find the relationship between two people
find_relation(X, Y, 'father') :- father(X, Y).
find_relation(X, Y, 'mother') :- mother(X, Y).
find_relation(X, Y, 'sibling') :- sibling(X, Y).
find_relation(X, Y, 'grandparent') :- grandparent(X, Y).
find_relation(X, Y, 'grandfather') :- grandfather(X, Y).
find_relation(X, Y, 'grandmother') :- grandmother(X, Y).
find_relation(_, _, 'No direct relation found').
```

Queries:



Practical 7

Code:

```
from pgmpy.models import BayesianNetwork
from pgmpy.factors.discrete import TabularCPD
from pgmpy.inference import VariableElimination

# Define the structure of the Bayesian Network
model = BayesianNetwork([
    ('I', 'E'),  # Intelligence affects Exam Performance
    ('H', 'E'),  # Hours of Study affects Exam Performance
    ('E', 'G'),  # Exam Performance affects Final Grade
    ('P', 'G')   # Previous Performance affects Final Grade
])

# Define Conditional Probability Distributions (CPDs)

# Intelligence (I) - Prior Probability
cpd_I = TabularCPD(variable='I', variable_card=2, values=[[0.7], [0.3]])  # 70% Low, 30% High

# Hours of Study (H) - Prior Probability
cpd_H = TabularCPD(variable='H', variable_card=2, values=[[0.6], [0.4]])  # 60% Less, 40%
More

# Previous Performance (P) - Prior Probability
cpd_P = TabularCPD(variable='P', variable_card=2, values=[[0.65], [0.35]])  # 65% Poor, 35%
Good

# Exam Performance (E) - Conditional on Intelligence (I) and Study Hours (H)
cpd_E = TabularCPD(variable='E', variable_card=2,
                   values=[[0.9, 0.6, 0.7, 0.1],  # P(E=Good | I, H)
                           [0.1, 0.4, 0.3, 0.9]],  # P(E=Bad | I, H)
                   evidence=['I', 'H'], evidence_card=[2, 2])

# Final Grade (G) - Conditional on Exam Performance (E) and Previous Performance (P)
cpd_G = TabularCPD(variable='G', variable_card=2,
                   values=[[0.95, 0.7, 0.8, 0.2],  # P(G=Pass | E, P)
                           [0.05, 0.3, 0.2, 0.8]],  # P(G=Fail | E, P)
                   evidence=['E', 'P'], evidence_card=[2, 2])

# Add CPDs to the model
model.add_cpds(cpd_I, cpd_H, cpd_P, cpd_E, cpd_G)

# Verify the model
assert model.check_model()

# Perform inference
inference = VariableElimination(model)

# Function to process user queries
```

```
def get_probability(query):
    query = query.lower().strip()

    # Define mappings for user-friendly input
    variable_map = {
        "intelligence": "I",
        "study hours": "H",
        "previous performance": "P",
        "exam performance": "E",
        "final grade": "G"
    }

    words = query.split()
    infer_var = None
    evidence_var = None

    # Extract variables from the query
    for phrase in variable_map:
        if f"find probability of {phrase}" in query:
            infer_var = variable_map[phrase]
        elif f"when {phrase}" in query:
            evidence_var = variable_map[phrase]

    # Ensure valid inference and evidence variables are found
    if not infer_var or not evidence_var:
        print("Error: Invalid query format! Use queries like 'Find probability of Final Grade when Exam Performance is Good'.")
        return

    # Setting evidence value as True (1) since it's explicitly mentioned in the query
    evidence_value = 1

    # Compute probability using Bayesian Inference
    result = inference.query(variables=[infer_var], evidence={evidence_var: evidence_value})

    print(f"P({infer_var} | {evidence_var}=1) = {result.values[1]:.3f}")

# Accept user queries
while True:
    user_query = input("\nEnter your query (or type 'exit' to stop): ")
    if user_query.lower() == "exit":
        break
    get_probability(user_query)
```

Output:

Enter your query (or type 'exit' to stop): Find probability of Exam Performance when Study Hours are More
P(E | H=1) = 0.550

Enter your query (or type 'exit' to stop): Find probability of Previous Performance when Final Grade is Good
P(P | G=1) = 0.717

Enter your query (or type 'exit' to stop): Find probability of Exam Performance when Intelligence is High and Study Hours are More
P(E | I=1) = 0.540

Enter your query (or type 'exit' to stop): exit

PRACTICAL 08

CODE:

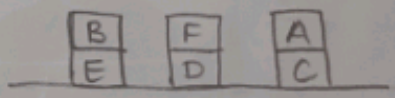
```
# Clean install of numpy compatible with everything
!pip install numpy==1.26.4 matplotlib==3.8.4

# Reinstall unified-planning and planners (with clean
dependencies)
!pip install -U "unified-
planning[pyperplan,tamer,fast-
downward,enhsp,plot]"

import numpy
print(numpy.__version__)  # should be 1.26.4
from unified_planning.shortcuts import *

# Install necessary packages (if not done)
!pip install -U "unified-
planning[pyperplan,tamer,fast-
downward,enhsp,plot]"

# ---- Import Libraries ----
from unified_planning.shortcuts import *
from unified_planning.model.types import BOOL

# ---- Define Block Type and Fluents ----
Block = UserType("Block")
clear = Fluent("clear", BOOL, obj=Block)
on_table = Fluent("on_table", BOOL, obj=Block)
arm_empty = Fluent("arm_empty", BOOL)
holding = Fluent("holding", BOOL, obj=Block)
on = Fluent("on", BOOL, above=Block,
below=Block)

# ---- Define Actions ----
pickup = InstantaneousAction("pickup", obj=Block)
pickup.add_precondition(clear(pickup.obj) &
on_table(pickup.obj) & arm_empty)
pickup.add_effect(holding(pickup.obj), True)
pickup.add_effect(clear(pickup.obj), False)
pickup.add_effect(on_table(pickup.obj), False)
pickup.add_effect(arm_empty, False)

putdown = InstantaneousAction("putdown",
obj=Block)
putdown.add_precondition(holding(putdown.obj))
putdown.add_effect(holding(putdown.obj), False)
putdown.add_effect(clear(putdown.obj), True)
putdown.add_effect(on_table(putdown.obj), True)
putdown.add_effect(arm_empty, True)

stack = InstantaneousAction("stack", obj=Block,
underobj=Block)
stack.add_precondition(clear(stack.underobj) &
holding(stack.obj))
stack.add_precondition(Not(Equals(stack.obj,
stack.underobj)))
stack.add_effect(arm_empty, True)
stack.add_effect(clear(stack.obj), True)
stack.add_effect(on(stack.obj, stack.underobj), True)
stack.add_effect(clear(stack.underobj), False)
stack.add_effect(holding(stack.obj), False)

unstack = InstantaneousAction("unstack", obj=Block,
underobj=Block)
unstack.add_precondition(on(unstack.obj,
unstack.underobj) & clear(unstack.obj) &
arm_empty)
unstack.add_effect(holding(unstack.obj), True)
unstack.add_effect(clear(unstack.underobj), True)
unstack.add_effect(on(unstack.obj,
unstack.underobj), False)
unstack.add_effect(clear(unstack.obj), False)
unstack.add_effect(arm_empty, False)

# ---- Define Problem ----
problem = Problem("Custom Blocks World")
for f in [clear, on_table, arm_empty, holding, on]:
    problem.add_fluent(f, default_initial_value=False)

problem.add_actions([pickup, putdown, stack,
unstack])
problem.set_initial_value(arm_empty, True)

# ---- Define Objects ----
blocks = ['A', 'B', 'C', 'D', 'E', 'F']
objects = [Object(b, Block) for b in blocks]
problem.add_objects(objects)

# ---- Initial State ----
# On Table
for b in ['C', 'D', 'E']:

problem.set_initial_value(on_table(problem.object(b)
), True)

# Clear blocks
```

```
for b in ['A', 'B', 'F']:
    problem.set_initial_value(clear(problem.object(b)),
True)

# Stacked relationships
problem.set_initial_value(on(problem.object('A'),
problem.object('C')), True)
problem.set_initial_value(on(problem.object('B'),
problem.object('E')), True)
problem.set_initial_value(on(problem.object('F'),
problem.object('D')), True)

# ---- Goal State ----
# A on B, B on C, C on D, D on E, E on F
goal_order = ['A', 'B', 'C', 'D', 'E', 'F']
for i in range(len(goal_order)-1):
    top = problem.object(goal_order[i])
    bottom = problem.object(goal_order[i+1])
    problem.add_goal(on(top, bottom))

# F must be on table
problem.add_goal(on_table(problem.object('F')))



# ---- Solve ----
up.shortcuts.get_environment().credits_stream =
None  # avoid printing credits

with OneshotPlanner(problem_kind=problem.kind)
as planner:
    result = planner.solve(problem)
    print(result)
```

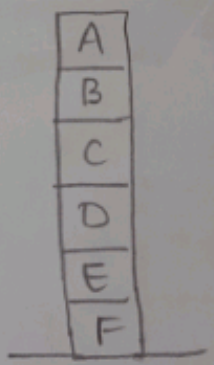
OUTPUT:

```
status: SOLVED_SATISFICING
engine: Fast Downward
plan: SequentialPlan:
    unstack(A, C)
    stack(A, B)
    unstack(F, D)
    putdown(F)
    unstack(A, B)
    putdown(A)
    unstack(B, E)
    stack(B, C)
    pickup(A)
    stack(A, B)
    pickup(E)
    stack(E, F)
    pickup(D)
    stack(D, E)
    unstack(A, B)
    putdown(A)
    unstack(B, C)
    putdown(B)
    pickup(C)
    stack(C, D)
    pickup(B)
    stack(B, C)
    pickup(A)
    stack(A, B)
```

Initial State:



Goal State:



Practical 09

Code:

```
#Download Libraries
!pip install nltk spacy textblob langdetect googletrans==4.0.0-rc1 scikit-learn

import nltk
import spacy
from textblob import TextBlob
from langdetect import detect
from googletrans import Translator
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize, sent_tokenize
from nltk.stem import PorterStemmer
from nltk.stem import WordNetLemmatizer
from collections import Counter
import string

nltk.download('punkt')
nltk.download('stopwords')
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')
nltk.download('punkt_tab')
nltk.download('averaged_perceptron_tagger_eng')

text = "John lives in New York. He loves programming and writes code every day. Sometimes, he
makes spelling mistakes."
print("Input Text:\n", text)
```

--Input Text:
John lives in New York. He loves programming and writes code every day. Sometimes, he makes spelling mistakes.

```
#Sentence Tokenization
sentences = sent_tokenize(text)
print("Sentence Tokenization:\n", sentences)
```

--Sentence Tokenization:
['John lives in New York.', 'He loves programming and writes code every day.', 'Sometimes, he makes spelling mistakes.']

```
#Word Tokenization
words = word_tokenize(text)
print("Word Tokenization:\n", words)
```

--Word Tokenization:
['John', 'lives', 'in', 'New', 'York', '.', 'He', 'loves', 'programming', 'and', 'writes', 'code', 'every', 'day', '.', 'Sometimes', ',', 'he', 'makes', 'spelling', 'mistakes', '.']

```
#Stopword Removal
```

```
stop_words = set(stopwords.words('english'))
filtered_words = [w for w in words if w.lower() not in stop_words]
print("After Stopword Removal:\n", filtered_words)
--After Stopword Removal:
['John', 'lives', 'New', 'York', '.', 'loves', 'programming', 'writes', 'code', 'every', 'day', '.', 'Sometimes', ',',
'makes', 'spelling', 'mistakes', '.']

#Stemming
stemmer = PorterStemmer()
stemmed = [stemmer.stem(w) for w in filtered_words]
print("After Stemming:\n", stemmed)
--After Stemming:
['john', 'live', 'new', 'york', '.', 'love', 'program', 'write', 'code', 'everi', 'day', '.', 'sometim', ',', 'make',
'spell', 'mistak', '.']

#Lemmatization
lemmatizer = WordNetLemmatizer()
lemmatized = [lemmatizer.lemmatize(w) for w in filtered_words]
print("After Lemmatization:\n", lemmatized)
--After Lemmatization:
['John', 'life', 'New', 'York', '.', 'love', 'programming', 'writes', 'code', 'every', 'day', '.', 'Sometimes', ',',
'make', 'spelling', 'mistake', '.']

#Part-of-Speech (POS) Tagging
pos_tags = nltk.pos_tag(words)
print("POS Tags:\n", pos_tags)
--POS Tags:
[('John', 'NNP'), ('lives', 'VBZ'), ('in', 'IN'), ('New', 'NNP'), ('York', 'NNP'), ('.', '.'), ('He', 'PRP'),
('loves', 'VBZ'), ('programming', 'VBG'), ('and', 'CC'), ('writes', 'NNS'), ('code', 'VBP'), ('every', 'DT'),
('day', 'NN'), ('.', '.'), ('Sometimes', 'RB'), (',', ','), ('he', 'PRP'), ('makes', 'VBZ'), ('spelling', 'VBG'),
('mistakes', 'NNS'), ('.', '.')]

#Named Entity Recognition (NER)
nlp = spacy.load("en_core_web_sm")
doc = nlp(text)
print("Named Entities:")
for ent in doc.ents:
  print(ent.text, "-", ent.label_)
--Named Entities:
John - PERSON
New York - GPE

#Text Normalization
normalized_text = text.lower().translate(str.maketrans('', '', string.punctuation))
print("Normalized Text:\n", normalized_text)
--Normalized Text:
```

```
john lives in new york he loves programming and writes code every day sometimes he makes
spelling mistakes

#Word Frequency Counting
word_freq = Counter(word_tokenize(normalized_text))
print("Word Frequencies:\n", word_freq)
--Word Frequencies:
Counter({'he': 2, 'john': 1, 'lives': 1, 'in': 1, 'new': 1, 'york': 1, 'loves': 1, 'programming': 1, 'and': 1,
'writes': 1, 'code' 1, 'every': 1, 'day': 1, 'sometimes': 1, 'makes': 1, 'spelling': 1, 'mistakes': 1})

#Bag of Words (BOW) Representation
texts = ["John lives in New York.",
      "He loves programming and writes code every day.",
      "Sometimes, he makes spelling mistakes."]
vectorizer = CountVectorizer()
bow_matrix = vectorizer.fit_transform(texts)
print("Bag of Words Representation:\n", bow_matrix.toarray())
print("Vocabulary:\n", vectorizer.vocabulary_)
--Bag of Words Representation:
[[0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 0 1]
[1 1 1 1 1 0 0 0 1 0 0 0 1 0 0 1 0]
[0 0 0 0 1 0 0 0 0 1 1 0 0 1 1 0 0]]
Vocabulary:
{'john': 6, 'lives': 7, 'in': 5, 'new': 11, 'york': 16, 'he': 4, 'loves': 8, 'programming': 12, 'and': 0, 'writes':
15, 'code': 1, 'every': 3, 'day': 2, 'sometimes': 13, 'makes': 9, 'spelling': 14, 'mistakes': 10}

#Vectorization
vectorizer = TfidfVectorizer()
tfidf_matrix = vectorizer.fit_transform(texts)
print("TF-IDF Matrix:\n", tfidf_matrix.toarray())
print("Vocabulary:\n", vectorizer.vocabulary_)
--TF-IDF Matrix:
[[0.       0.       0.       0.       0.       0.4472136
 0.4472136  0.4472136  0.       0.       0.       0.4472136
 0.       0.       0.       0.       0.4472136 ]
[0.36325471 0.36325471 0.36325471 0.36325471 0.27626457 0.
 0.       0.       0.36325471 0.       0.       0.
 0.36325471 0.       0.       0.36325471 0.       ]
[0.       0.       0.       0.       0.35543247 0.
 0.       0.       0.       0.46735098 0.46735098 0.
 0.       0.46735098 0.46735098 0.       0.       ]]
Vocabulary:
{'john': 6, 'lives': 7, 'in': 5, 'new': 11, 'york': 16, 'he': 4, 'loves': 8, 'programming': 12, 'and': 0, 'writes':
15, 'code': 1, 'every': 3, 'day': 2, 'sometimes': 13, 'makes': 9, 'spelling': 14, 'mistakes': 10}

#Sentiment Analysis
blob = TextBlob(text)
```

```
print("Sentiment Polarity:", blob.sentiment.polarity)
print("Sentiment Subjectivity:", blob.sentiment.subjectivity)
--Sentiment Polarity: 0.13636363636363635
Sentiment Subjectivity: 0.45454545454545453

#Language Detection
language = detect(text)
print("Detected Language:", language)
--Detected Language: en

#Translation
translator = Translator()
translated = translator.translate("John lives in New York. He loves programming.", dest='hi')
print("Translated Text:\n", translated.text)
--Translated Text:
जॉन न्यूयॉर्क में रहता है।वह प्रोग्रामिंग से प्यार करता है।

#Spelling Correction
text_with_typos = "He loevs progrraming and wrties code every dya. Sometimes, he makes sppeelling
mistokes."
corrected_text = TextBlob(text_with_typos).correct()
print("Corrected Text:\n", corrected_text)
--Corrected Text:
He loves programming and writes code every day. Sometimes, he makes spelling mistakes.
```

Practical 10

Code:

```
pip install pandas numpy matplotlib scikit-learn tensorflow
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.optimizers import Adam

df = pd.read_csv('creditcard.csv.zip')

# Drop 'Time' column (not needed)
df.drop('Time', axis=1, inplace=True)
# Normalize 'Amount' using StandardScaler
scaler = StandardScaler()
df['Amount'] = scaler.fit_transform(df['Amount'].values.reshape(-1, 1))
# Separate features and labels
X = df.drop('Class', axis=1)
y = df['Class']

# Keep only normal transactions (Class = 0)
df_normal = df[df['Class'] == 0].drop('Class', axis=1)
# Split into training and validation sets
X_train, X_val = train_test_split(df_normal, test_size=0.2, random_state=42)

input_dim = X_train.shape[1]
```

```
# Define layers
input_layer = Input(shape=(input_dim,))
encoded = Dense(14, activation='relu')(input_layer)
encoded = Dense(7, activation='relu')(encoded)
decoded = Dense(14, activation='relu')(encoded)
decoded = Dense(input_dim, activation='linear')(decoded)


# Create and compile the model
autoencoder = Model(inputs=input_layer, outputs=decoded)
autoencoder.compile(optimizer=Adam(learning_rate=0.001), loss='mse')
autoencoder.fit(X_train, X_train,
                epochs=10,
                batch_size=32,
                validation_data=(X_val, X_val),
                verbose=1)
```

```
Epoch 1/10
7108/7108 ━━━━━━━━━━━━━━━━━━━━ 19s 3ms/step - loss: 0.3474 - val_loss: 0.3505
Epoch 2/10
7108/7108 ━━━━━━━━━━━━━━━━━━━━ 15s 2ms/step - loss: 0.3424 - val_loss: 0.3404
Epoch 3/10
7108/7108 ━━━━━━━━━━━━━━━━━━━━ 16s 2ms/step - loss: 0.3388 - val_loss: 0.3358
Epoch 4/10
7108/7108 ━━━━━━━━━━━━━━━━━━━━ 19s 3ms/step - loss: 0.3279 - val_loss: 0.3322
Epoch 5/10
7108/7108 ━━━━━━━━━━━━━━━━━━━━ 16s 2ms/step - loss: 0.3254 - val_loss: 0.3309
Epoch 6/10
7108/7108 ━━━━━━━━━━━━━━━━━━━━ 17s 2ms/step - loss: 0.3277 - val_loss: 0.3283
Epoch 7/10
7108/7108 ━━━━━━━━━━━━━━━━━━━━ 20s 2ms/step - loss: 0.3275 - val_loss: 0.3289
Epoch 8/10
7108/7108 ━━━━━━━━━━━━━━━━━━━━ 16s 2ms/step - loss: 0.3267 - val_loss: 0.3287
Epoch 9/10
7108/7108 ━━━━━━━━━━━━━━━━━━━━ 20s 2ms/step - loss: 0.3219 - val_loss: 0.3254
Epoch 10/10
7108/7108 ━━━━━━━━━━━━━━━━━━━━ 23s 3ms/step - loss: 0.3198 - val_loss: 0.3345
<keras.src.callbacks.history.History at 0x7cef5f362cd0>
```

```
# Normalize entire dataset again
X_scaled = scaler.fit_transform(X)


# Predict reconstruction
reconstructions = autoencoder.predict(X_scaled)
# Calculate reconstruction error
mse = np.mean(np.power(X_scaled - reconstructions, 2), axis=1)
# Define threshold (e.g., 95th percentile)
```

```
threshold = np.percentile(mse, 95)

# Predict fraud

y_pred = (mse > threshold).astype(int)
```

```
8901/8901 ——————————— 10s 1ms/step
```

```
print(classification_report(y, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.95      0.98    284315
           1       0.03      0.88      0.06       492

    accuracy                           0.95    284807
   macro avg       0.52      0.91      0.52    284807
weighted avg       1.00      0.95      0.97    284807
```