CS601 Introduction to Artificial Intelligence

Assignment 2

Question 1

(a) Heuristic is admissible, given heuristic cost of each node is less than or equal to minimum cost to the goal.

Node n	Minimum cost to reach goal from n	h(n)
S	15	15
A	13	13
B	12	11
C	8	8
D	7	5
E	8	7
F	7	7
H	3	2

(b) Heuristic is not consistent. Consistent heuristic means for every node n, and all its successors p, will have $h(n) \leq c(n,p) + h(p)$. The blue font in the following table indicates that this inequality relation is not satisfied.

Node n	h(n)	A		B		C		D		E		F		H	
		c(n,A)	c(n,A)+h(A)	c(n,B)	c(n,B)+h(B)	c(n,C)	c(n,C)+h(C)	c(n,D)	c(n,D)+h(D)	c(n,E)	c(n,E)+h(E)	c(n,F)	c(n,F)+h(F)	c(n,H)	c(n,H)+h(H)
S	15	7	20	3	14	8	16	9	14	13	20	14	21	12	14
A	13	-	-	2	13	6	14	7	12	6	13	12	19	10	12
B	11	-	-	-	-	4	12	5	10	-	-	10	17	9	11
C	8	-	-	-	-	-	-	1	6	-	-	6	13	5	7
D	5	-	-	-	-	-	-	-	-	-	-	-	-	4	6
E	7	-	-	-	-	-	-	-	-	-	-	-	-	5	7
F	7	-	-	-	-	-	-	-	-	-	-	-	-	4	6
H	2	-	-	-	-	-	-	-	-	-	-	-	-	-	-

(c) Search steps

1. Algorithm: DFS

Data Structure of Open List: Stack

Step #	Open List	POP	Nodes to add
1	S	S	A^S, B^S, C^S
2	A^S, B^S, C^S	C^S	D^C, F^C
3	A^S, B^S, D^C, F^C	F^C	H^F
4	A^S, B^S, D^C, H^F	H^F	G^H
5	A^S, B^S, D^C, G^H	G^H	-

Path: S->C->F->H->G, Cost: 8+6+4+3=21

2. Algorithm: BFS

Data Structure of Open List: Queue

Step #	Open List	POP	Nodes to add	Nodes will add
1	S	S	A^S, B^S, C^S	A^S, B^S, C^S
2	A^S, B^S, C^S	A^S	E^A, H^A	E^A, H^A
3	B^S, C^S, E^A, H^A	B^S	D^B, C^B	D^B
4	C^S, E^A, H^A, D^B	C^S	D^C, F^C	F^C
5	E^A, H^A, D^B, F^C	E^A	H^E	-
6	H^A, D^B, F^C	H^A	G^H	G^H
7	D^B, F^C, G^H	D^B	H^D	-
8	F^C, G^H	F^C	H^F	-
9	G^H	G^H	-	-

Path: S->A->H->G, Cost: 7+10+3=20

3. Algorithm: BFS

Data Structure of Open List: Priority Queue

Step #	Open List	POP	Nodes to add	Update	Add
1	S	S	A^{S7}, B^{S3}, C^{S8}	-	A^{S7}, B^{S3}, C^{S8}
2	B^{S3}, A^{S7}, C^{S8}	B^{S3}	D^{B10}, C^{B7}	C^{S8} -> C^{B7}	D^{B10}
3	A^{S7}, C^{B7}, D^{B10}	A^{S7}	E^{A13}, H^{A17}	-	E^{A13}, H^{A17}
4	C^{B7}, D^{B10}, E^{A13}, H^{A17}	C^{B7}	D^{C8}, F^{C13}	D^{B10} -> D^{C8}	F^{C13}
5	D^{C8}, E^{A13}, F^{C13}, H^{A17}	D^{C8}	H^{D12}	H^{A17} -> H^{D12}	-
6	H^{D12}, E^{A13}	H^{D12}	G^{H15}	-	G^{H15}
7	E^{A13}, G^{H15}	E^{A13}	H^{E18}	-	-
8	G^{H15}	G^{H15}	-	-	-

Path: S->B->C->D->H->G, Cost: 3+4+1+4+3=15

4. Algorithm: Best First Search
Data Structure of Open List: Priority Queue

Step #	Open List	POP	Nodes to add	Nodes not explored
1	S	S	A^{S13}, B^{S11}, C^{S8}	A^{S13}, B^{S11}, C^{S8}
2	C^{S8}, B^{S11}, A^{S13}	C^{S8}	D^{C5}, F^{C7}	D^{C5}, F^{C7}
3	D^{C5}, F^{C7}, B^{S11}, A^{S13}	D^{C5}	H^{D2}	H^{D2}
4	H^{D2}, F^{C7}, B^{S11}, A^{S13}	H^{D2}	G^{H0}	G^{H0}
5	G^{H0}, F^{C7}, B^{S11}, A^{S13}	G^{H0}	-	-

Path: S->C->D->H->G, Cost: 8+1+4+3=16

5. Algorithm: A* Search
Data Structure of Open List: Priority Queue

Step #	Open List	POP	Nodes to add	Reopen
1	S	S	A^{S20}, B^{S14}, C^{S16}	-
2	B^{S14}, C^{S16}, A^{S20}	B^{S14}	D^{B15}, C^{B15}	-
3	D^{B15}, C^{B15}, A^{S20}	D^{B15}	H^{D16}	-
4	C^{B15}, H^{D16}, A^{S20}	C^{B15}	D^{C13}, F^{C20}	D^{B15}
5	D^{C13}, H^{D16}, F^{C20}, A^{S20}	D^{C13}	H^{D14}	-
6	H^{D14}, F^{C20}, A^{S20}	H^{D14}	G^{H15}	-
7	G^{H15}, F^{C20}, A^{S20}	G^{H15}	-	-
8	F^{C20}, A^{S20}	F^{C20}	H^{F19}	-
9	A^{S20}	A^{S20}	E^{A20}, H^{A19}	-
10	E^{A20}	E^{A20}	H^{E20}	-

Path: S->B->C->D->H->G, Cost: 3+4+1+4+3=15

Question 2

A* search was carried out in finding path. Path cost was decided by Euclidean distance and after comparing different setting of heuristic cost, using half of height difference between current position and goal position depicted best results. Thus, the heuristic cost of each position to the goal position was measured by the half of the height difference between goal position and current position.

Distance from start to current position	Heuristic cost between current location and goal	Dynamic_config1	Dynamic_config2	Dynamic_config3	Dynamic_config4	Dynamic_config5
Euclidean	Euclidean/2	Not reachable	9	12	20	13
Euclidean	Manhattan/2	Not reachable	12	12	Not reachable	12
Euclidean	Height Difference/2	Not reachable	9	12	20	13

(a) SearchAgent.py
1. import static function from searchUtils.py

```
# import static function from searchUtils
from searchUtils import findBestPath
from searchUtils import reconstruct_path, heuristic_cost_estimate, distance_between
from searchUtils import getCompleteLocation
from searchUtils import isPresentStateInList
```

2. drive function under SearchAgent class

```
    def drive(self, goalstates, inputs):
        """Write your algorithm for self driving car"""
        start = self.state
        goalReached_path_current = []
        for goal in goalstates:
            print(self.A_Star(start, goal))
            goalReached_path_current.append(self.A_Star(start, goal))

        flag, best_path = findBestPath(goalReached_path_current)
        act_sequence = []
        if len(best_path) > 0:
            act_sequence = self.searchutil.retrieveActionSequenceFromPath(best_path)
        return act_sequence
```

3. A_Star function under SearchAgent class

```
    def A_Star(self, start, goal):
        """implement A_Star search, return whether reachable and corresponding path"""
        closeSet = []
        openSet = [start]
        gScore = {start["location"]: 0}
        h_score = heuristic_cost_estimate(start["location"], goal["location"])
        fScore = {start["location"]: gScore[start["location"]] + h_score}

        while len(openSet) > 0:
            # get node in openSet have lowest fScore, name it as current
            openSet_fScore = dict(filter(lambda x: x[0] in [os["location"] for os in openSet], fScore.items()))
            openSet_fScore = dict(sorted(openSet_fScore.items(), key=lambda item: item[1]))
            current_loc = list(openSet_fScore.keys())[0]

            current = getCompleteLocation(openSet, current_loc)

            # remove current from open set and fScore
            openSet.remove(current)
            # add current to close set
            closeSet.append(current)

            if current["location"] == goal["location"]:
                return "Goal Reached", reconstruct_path(closeSet, current["location"])

            for action in self.valid_actions:
                # next position after take action
                next_pos = self.env.applyAction(self, current, action)
                next_pos["previous"] = next_pos["previous"]["location"]

                if next_pos["location"] == current["location"]:
                    continue
                # gScore, fScore on next position
                temp_gScore = gScore[current["location"]] + distance_between(current["location"], next_pos["location"])
                temp_fScore = temp_gScore + heuristic_cost_estimate(next_pos["location"], goal["location"])
                # update openSet/gScore/fScore
                if isPresentStateInList(next_pos, openSet):
                    # next position in openSet
                    if temp_fScore < fScore[next_pos["location"]]:
                        # replace old node in openSet
                        for o in openSet:
                            if o["location"] == next_pos["location"]:
                                openSet.remove(o)
                                openSet.append(next_pos)
                                break
                    else:
                        # fScore not less than before, ignore
                        continue

                elif isPresentStateInList(next_pos, closeSet):
                    # next position in closeSet
                    if temp_fScore < fScore[next_pos["location"]]:
                        # reopen: delete from closeSet, add new to openSet
                        for c in closeSet:
                            if c["location"] == next_pos["location"]:
                                closeSet.remove(c)
                                openSet.append(next_pos)
                                break
                    else:
                        # fScore not less than before, ignore
                        continue
```

(continue…)

```
            elif isPresentStateInList(next_pos, closeSet):
                # next position in closeSet
                if temp_fScore < fScore[next_pos["location"]]:
                    # reopen: delete from closeSet, add new to openSet
                    for c in closeSet:
                        if c["location"] == next_pos["location"]:
                            closeSet.remove(c)
                            openSet.append(next_pos)
                            break
                else:
                    # fScore not less than before, ignore
                    continue

            else:
                # next position not in openSet or closeSet, add as new node into openSet
                openSet.append(next_pos)

            # new add in or fScore less than before, update fScore & gScore
            fScore[next_pos["location"]] = temp_fScore
            gScore[next_pos["location"]] = temp_gScore

        # no need to care about this notification, since current will be initially filled with start
        if current["location"] == goal["location"]:
            return "Goal Reached", reconstruct_path(closeSet, current["location"])
        else:
            return "Goal Not Reachable", reconstruct_path(closeSet, current["location"])
```

4. update function under SearchAgent class

```
    def update(self):
        """ The update function is called when a time step is completed in the
            environment for a given trial. This function will build the agent
            state, choose an action, receive a reward, and learn if enabled. """

        goalstates = self.env.getGoalStates()
        inputs = self.env.sense(self)
        self.action_sequence = self.drive(goalstates, inputs)
        action = self.choose_action()  # Choose an action
        self.state = self.env.act(self, action)
        return
```

(b) searchUtils.py

1. static functions

```
def computeTotalCost(path):
    """return total cost of one complete path"""
    total_cost = 0
    for i in range(len(path) - 1):
        total_cost += distance_between(path[i], path[i + 1])
    return total_cost

def computeHighestFarth(path):
    """return y of highest grid ever reached"""
    find_farth = 0
    for i in range(len(path)):
        if path[i][1] > find_farth:
            find_farth = path[i][1]
    return find_farth

def distance_between(start, end):
    """return Euclidean distance between two locations"""
    return math.sqrt(math.pow(end[0] - start[0], 2) + math.pow(end[1] - start[1], 2))

def manhattan_distance_between(start, end):
    """return Manhattan distance between two locations"""
    return math.fabs(end[0] - start[0]) + math.fabs(end[1] - start[1])
```

```python
def height_difference(start, end):
    """return height difference between two locations"""
    return math.fabs(end[1] - start[1]) / 2

def heuristic_cost_estimate(start, goal):
    """return heuristic distance between two locations"""
    # return manhattan_distance_between(start, goal) / 2
    # return distance_between(start, goal) / 2
    return height_difference(start, goal) / 2

def findBestPath(goalReached_path_current):
    """find best path"""
    best_path = []
    best_path_ = []
    reachable_cost = math.inf
    highest_farth = 0
    for reach_or_not, path in goalReached_path_current:
        if reach_or_not == "Goal Reached":
            total_cost = computeTotalCost(path)
            if reachable_cost > total_cost:
                reachable_cost = total_cost
                best_path = path
        elif reach_or_not == "Goal Not Reachable":
            find_farth = computeHighestFarth(path)
            if highest_farth < find_farth:
                highest_farth = find_farth
                best_path_ = path

    if len(best_path) > 0:
        return "Goal Reached", best_path
    return "Goal Not Reachable", best_path_

def getCompleteLocation(close_open_Set, loc):
    """return dictionary version of location"""
    current = {}
    for c in close_open_Set:
        if c["location"] == loc:
            current = c
    return current

def reconstruct_path(closeSet, curr_loc):
    """reconstruct path given closeSet and current location"""
    total_path = [curr_loc]
    point = getCompleteLocation(closeSet, curr_loc)
    while point.get("previous") is not None:
        prev = point["previous"]
        point = getCompleteLocation(closeSet, prev)
        total_path.insert(0, point["location"])
    return total_path

def isPresentStateInList(state, searchList):
    """return true if state in searchList"""
    for elem in searchList:
        if state["location"] == elem["location"]:
            return 1
    return 0
```

2. function under searchUtils class

```python
    def retrieveActionSequenceFromPath(self, path):
        """return action sequence given path"""
        action_sequence = []
        for i in range(len(path) - 1):
            action_sequence.append(self.env.getAction({"location": path[i]}, {"location": path[i + 1]}))
        return action_sequence
```

Question 3

(a) MDP model

s(current state)	a(action)	s′(new state)	Pr(s′|s,a)	Compute R(s, a, s′)	R(s, a, s′)
L1	Pickup	L1	0.7	$0	$0
	Pickup	L2	0.12	$(8-1)	$7
	Pickup	L3	0.105	$(13-1.5)	$11.5
	Pickup	L4	0.075	$(15-1.25)	$13.75
	Move	L2	1/3	-$1	-$1
	Move	L3	1/3	-$1.5	-$1.5
	Move	L4	1/3	-$1.25	-$1.25

s(current state)	a(action)	s′(new state)	Pr(s′|s,a)	Compute R(s, a, s′)	R(s, a, s′)
L2	Pickup	L1	0.32	$(10-1)	$9
	Pickup	L2	0.2	$0	$0
	Pickup	L3	0.48	$(9-0.75)	$8.25
	Move	L1	1/2	-$1	-$1
	Move	L3	1/2	-$0.75	-$0.75

s(current state)	a(action)	s′(new state)	Pr(s′|s,a)	Compute R(s, a, s′)	R(s, a, s′)
L3	Pickup	L1	0.06	$(13-1.5)	$11.5
	Pickup	L3	0.9	$0	$0
	Pickup	L4	0.04	$(10-0.8)	$9.2
	Move	L1	1/2	-$1.5	-$1.5
	Move	L4	1/2	-$0.8	-$0.8

s(current state)	a(action)	s′(new state)	Pr(s′|s,a)	Compute R(s, a, s′)	R(s, a, s′)
L4	Pickup	L1	0.39	$(9-1.25)	$7.75
	Pickup	L2	0.21	$(7-1)	$6
	Pickup	L4	0.4	$0	$0
	Move	L1	1/2	-$1.25	-$1.25
	Move	L2	1/2	-$1	-$1

(b)

s	a	$V^0(s)$	$Q^1(s,a)$	$V^1(s)$	$\pi^1(s)$	$Q^2(s,a)$	$V^2(s)$	$\pi^2(s)$	$Q^3(s,a)$	$V^3(s)$	$\pi^3(s)$
L1	Pickup	0	3.0788	3.0788	Pickup	6.4870	6.4870	Pickup	9.8952	9.8952	Pickup
	Move		-1.25			2.8102			6.8703		
L2	Pickup	0	6.84	6.84	Pickup	9.7010	9.7010	Pickup	12.5621	12.5621	Pickup
	Move		-0.875			1.1934			3.2618		
L3	Pickup	0	1.058	1.058	Pickup	2.3662	2.5306	Move	3.6745	6.2112	Move
	Move		-1.150			2.5306			6.2112		
L4	Pickup	0	4.283	4.283	Pickup	8.6326	8.6326	Pickup	12.9827	12.9827	Pickup
	Move		-1.125			3.8344			8.7938		

s	a	Compute $Q^1(s,a)$	Compute $Q^2(s,a)$	Compute $Q^3(s,a)$
L1	Pickup	$\sum_{s'\in\{L1,L2,L3,L4\}} Pr(s'\|s,a)[R(s,a,s')+V^0(s')]$	$\sum_{s'\in\{L1,L2,L3,L4\}} Pr(s'\|s,a)[R(s,a,s')+V^1(s')]$	$\sum_{s'\in\{L1,L2,L3,L4\}} Pr(s'\|s,a)[R(s,a,s')+V^2(s')]$
	Move	$\sum_{s'\in\{L2,L3,L4\}} Pr(s'\|s,a)[R(s,a,s')+V^0(s')]$	$\sum_{s'\in\{L2,L3,L4\}} Pr(s'\|s,a)[R(s,a,s')+V^1(s')]$	$\sum_{s'\in\{L2,L3,L4\}} Pr(s'\|s,a)[R(s,a,s')+V^2(s')]$
L2	Pickup	$\sum_{s'\in\{L1,L2,L3\}} Pr(s'\|s,a)[R(s,a,s')+V^0(s')]$	$\sum_{s'\in\{L1,L2,L3\}} Pr(s'\|s,a)[R(s,a,s')+V^1(s')]$	$\sum_{s'\in\{L1,L2,L3\}} Pr(s'\|s,a)[R(s,a,s')+V^2(s')]$
	Move	$\sum_{s'\in\{L1,L3\}} Pr(s'\|s,a)[R(s,a,s')+V^0(s')]$	$\sum_{s'\in\{L1,L3\}} Pr(s'\|s,a)[R(s,a,s')+V^1(s')]$	$\sum_{s'\in\{L1,L3\}} Pr(s'\|s,a)[R(s,a,s')+V^2(s')]$
L3	Pickup	$\sum_{s'\in\{L1,L3,L4\}} Pr(s'\|s,a)[R(s,a,s')+V^0(s')]$	$\sum_{s'\in\{L1,L3,L4\}} Pr(s'\|s,a)[R(s,a,s')+V^1(s')]$	$\sum_{s'\in\{L1,L3,L4\}} Pr(s'\|s,a)[R(s,a,s')+V^2(s')]$
	Move	$\sum_{s'\in\{L1,,L4\}} Pr(s'\|s,a)[R(s,a,s')+V^0(s')]$	$\sum_{s'\in\{L1,L4\}} Pr(s'\|s,a)[R(s,a,s')+V^1(s')]$	$\sum_{s'\in\{L1,L4\}} Pr(s'\|s,a)[R(s,a,s')+V^2(s')]$
L4	Pickup	$\sum_{s'\in\{L1,L2,L4\}} Pr(s'\|s,a)[R(s,a,s')+V^0(s')]$	$\sum_{s'\in\{L1,L2,L4\}} Pr(s'\|s,a)[R(s,a,s')+V^1(s')]$	$\sum_{s'\in\{L1,L2,L4\}} Pr(s'\|s,a)[R(s,a,s')+V^2(s')]$
	Move	$\sum_{s'\in\{L1,L2\}} Pr(s'\|s,a)[R(s,a,s')+V^0(s')]$	$\sum_{s'\in\{L1,L2\}} Pr(s'\|s,a)[R(s,a,s')+V^1(s')]$	$\sum_{s'\in\{L1,L2\}} Pr(s'\|s,a)[R(s,a,s')+V^2(s')]$

Question 4

(a) Implement Q-Learning

```
import gym
import numpy as np

import matplotlib.pyplot as plt
```

```
def select_action_with_epsilon_greedy(curr_s, q_table, epsilon=0.1):
    action = np.argmax(q_table[curr_s, :])
    if np.random.rand() < epsilon:
        action = np.random.randint(q_table.shape[1])
    return action
```

```
env = gym.make("FrozenLake-v0")

state_n = env.observation_space.n
action_n = env.action_space.n

# Meta parameters for the RL agent
alpha = 0.1
gamma = 0.95
epsilon = 0.5
epsilon_decay = 0.999

# Experimental setup
n_episode = 5000
max_step = 100

# Initialize Q-table
q_table = np.zeros([state_n, action_n])

# Initialize a list for storing simulation history
history = []
```

```
for episode in range(n_episode):
    # reset accumulated reward for this episode
    episode_reward = 0
    # Start a new episode and sample the initial state
    state = env.reset()
    # Select the first action in this episode
    action = select_action_with_epsilon_greedy(state, q_table, epsilon=epsilon)

    for step in range(max_step):
        # get result of action from the environment
        next_state, reward, done, info = env.step(action)
        # update accumulate reward
        episode_reward = reward + gamma * episode_reward
        # select an action
        next_action = select_action_with_epsilon_greedy(next_state, q_table, epsilon=epsilon)
        # calculate TD error
        delta = reward + gamma * np.max(q_table[next_state, :]) - q_table[state, action]
        # update q_table
        q_table[state, action] += alpha * delta

        state = next_state
        action = next_action

        if done:
            history.append([episode, step, episode_reward, reward, epsilon])
            break

    # Decay epsilon expolentially
    epsilon = epsilon * epsilon_decay


history = np.array(history)
```

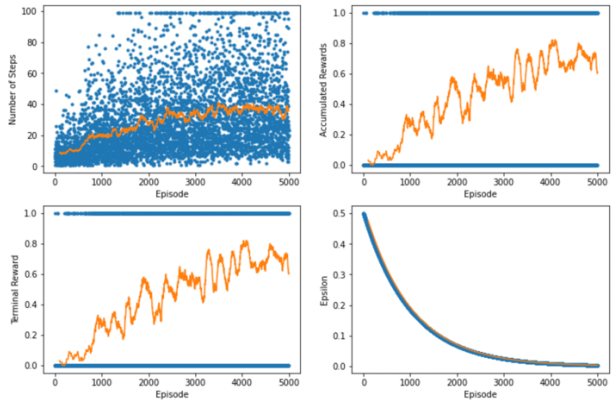
(b) Plot average return

```
window_size = 100

def running_average(x, window_size, mode='valid'):
    return np.convolve(x, np.ones(window_size)/window_size, mode=mode)

fig, ax = plt.subplots(2, 2, figsize=[12, 8])
# Number of Steps
ax[0, 0].plot(history[:, 0], history[:, 1], '.')
ax[0, 0].set_xlabel('Episode')
ax[0, 0].set_ylabel('Number of Steps')
ax[0, 0].plot(history[window_size-1:, 0], running_average(history[:, 1], window_size))
# Accumulated Rewards
ax[0, 1].plot(history[:, 0], history[:, 2], '.')
ax[0, 1].set_xlabel('Episode')
ax[0, 1].set_ylabel('Accumulated Rewards')
ax[0, 1].plot(history[window_size-1:, 0], running_average(history[:, 2], window_size))
# Terminal Reward
ax[1, 0].plot(history[:, 0], history[:, 3], '.')
ax[1, 0].set_xlabel('Episode')
ax[1, 0].set_ylabel('Terminal Reward')
ax[1, 0].plot(history[window_size-1:, 0], running_average(history[:, 3], window_size))
# Epsilon
ax[1, 1].plot(history[:, 0], history[:, 4], '.')
ax[1, 1].set_xlabel('Episode')
ax[1, 1].set_ylabel('Epsilon')
ax[1, 1].plot(history[window_size-1:, 0], running_average(history[:, 4], window_size))
```

(target figure is located at (0,1), other three figures are related graphs)



Question 5

```
import pandas as pd
import numpy as np
import re
import math
from sklearn.decomposition import TruncatedSVD
from gensim.models import word2vec
```

```
import nltk
from nltk.stem import WordNetLemmatizer
import sys
from nltk.corpus import wordnet
from nltk.corpus import stopwords
from nltk.parse.malt import MaltParser
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')
nltk.download('stopwords')
```

(a) Parse the reviews in “200Reviews.csv”:

```
q5_rd = pd.read_csv("200Reviews.csv")
q5_df = q5_rd.drop(["Unnamed: 0", "id"], axis=1)
# remove <br /> of raw text
q5_df["review"] =  q5_df["review"].apply(lambda x: re.sub("<br />", " ", x))
# sentence segmentation
q5_df["sentence"] = q5_df["review"].apply(lambda x: nltk.sent_tokenize(x))
# word tokenization
q5_df["word"] = q5_df["sentence"].apply(lambda x: [nltk.word_tokenize(x[k]) for k in range(len(x))])
# parts of speech
q5_df["parts_of_speech"] = q5_df["word"].apply(lambda x: [nltk.pos_tag(x[k]) for k in range(len(x))])
```

```
# lemmatization
def get_wordnet_pos(treebank_tag):
    if treebank_tag.startswith('J'):
        return wordnet.ADJ
    elif treebank_tag.startswith('V'):
        return wordnet.VERB
    elif treebank_tag.startswith('N'):
        return wordnet.NOUN
    elif treebank_tag.startswith('R'):
        return wordnet.ADV
    else:
        return ''


def transfer(w):
    wordnettag = get_wordnet_pos(w[1])
    if wordnettag == '':
        lemmatizedword = wordnet_lemmatizer.lemmatize(w[0].lower())
    else:
        lemmatizedword = wordnet_lemmatizer.lemmatize(w[0].lower(),pos=wordnettag)
    if w[0].istitle():
        lemmatizedword = lemmatizedword.capitalize()
    elif w[0].upper()==w[0]:
        lemmatizedword = lemmatizedword.upper()
    else:
        lemmatizedword = lemmatizedword
    return lemmatizedword


wordnet_lemmatizer = WordNetLemmatizer()

q5_df["lemmatization"] = q5_df["parts_of_speech"].apply(lambda x: [[transfer(w) for w in x[k]]
                                                                   for k in range(len(x))])
```

```
# remove stop word
stopWords = set(stopwords.words('english'))
q5_df["no_stop_word"] = q5_df["lemmatization"].apply(lambda x: [[w for w in x[k] if w.lower() not in stopWords]
                                                                for k in range(len(x))])
# filter out not alpha
q5_df["no_stop_word_isalpha"] = q5_df["no_stop_word"].apply(lambda x: [[w for w in x[k] if w.isalpha()]
                                                                       for k in range(len(x))])
```

(b) Create co-occurrence matrix for all the remaining words:

```
# get corpus
corpus = []
for i in q5_df["no_stop_word_isalpha"].values:
    for j in i:
        corpus.extend(j)

# corpus
```

```
def co_matix(corpus, unique_word, window_left_right):
    res = np.zeros((len(unique_word), len(unique_word)))
    for i in range(len(corpus)):
        if i - window_left_right < 0:
            for j in range(i + window_left_right):
                if i != j:
                    res[unique_word.index(corpus[i])][unique_word.index(corpus[j])] += 1
        elif i + window_left_right > len(corpus)-1:
            for j in range(i - window_left_right, len(corpus)):
                if i != j:
                    res[unique_word.index(corpus[i])][unique_word.index(corpus[j])] += 1
        else:
            for j in range(i - window_left_right, i + window_left_right):
                if i != j:
                    res[unique_word.index(corpus[i])][unique_word.index(corpus[j])] += 1
    return res
```

```
# unique words of corpus
unique_word = list(set(corpus))
# get co-occurance matrix for all remaining words with window size equals to 5
comatrix = co_matix(corpus, unique_word, 5)
```

(c) Apply SVD and obtain word embeddings of size 100:

```
n = 100
svd = TruncatedSVD(n_components=n)
U_sigma_trunc = svd.fit_transform(comatrix)
```

(d) Generate word embedding of size 100 using Word2Vec.ipynb:

```
sentences = []
for i in q5_df["no_stop_word_isalpha"].values:
    for j in i:
        sentences.append(j)

# sentences
```

```
# Creating the model and setting values for the various parameters
num_features = 100  # Word vector dimensionality
min_word_count = 1 # Minimum word count
num_workers = 4     # Number of parallel threads
context = 5        # Context window size
downsampling = 1e-3 # (0.001) Downsample setting for frequent words

# Initializing the train model
print("Training model....")
model = word2vec.Word2Vec(sentences,\
                          workers=num_workers,\
                          size=num_features,\
                          min_count=min_word_count,\
                          window=context,
                          sample=downsampling)

# To make the model memory efficient
model.init_sims(replace=True)

# Saving the model for later use. Can be loaded using Word2Vec.load()
model_name = "q5_word2vec"
model_path = f"{model_name}"
model.save(model_path)
print("model saved")
```

(e) Comparison between those two methods result:

```
# calculate similarity for SVD
def most_similar_words(word, topn, word_matrix, word_to_id, id_to_word):
    row = word_to_id[word]
    vec = word_matrix[row,:]
    m = word_matrix
    dot_m_v = m.dot(vec.T) # vector
    dot_m_m = np.sum(m * m, axis=1) # vector
    dot_v_v = vec.dot(vec.T) # float
    sims = dot_m_v / (math.sqrt(dot_v_v) * np.sqrt(dot_m_m))

    return [(id_to_word[id], round(sims[id],4)) for id in (-sims).argsort()[1:topn+1]]
```

```
# Build unigram <-> index lookup.
word_to_id, id_to_word = {}, {}
for i, x in enumerate(unique_word):
    word_to_id[x] = i
    id_to_word[i] = x
```

```
# comparison between two methods
def compare(word, w2v_model, topn, word_matrix, word_to_id, id_to_word):
    print(f"Most similar words to '{word}':")
    print()
    print("SVD: ", most_similar_words(word, 10, word_matrix, word_to_id, id_to_word))
    print()
    print("Word2Vec: ", [(i, round(j,4)) for i, j in model.wv.most_similar(word)])
```

```
compare("film", model, 10, U_sigma_trunc, word_to_id, id_to_word)
```

```
Most similar words to 'film':

SVD:  [('movie', 0.774), ('one', 0.7732), ('good', 0.745), ('bad', 0.742), ('make', 0.7345), ('something', 0.7199),
('understand', 0.7198), ('actually', 0.7191), ('even', 0.7159), ('really', 0.7124)]

Word2Vec:  [('movie', 0.9024), ('go', 0.8825), ('make', 0.8808), ('like', 0.8799), ('scene', 0.8799), ('get', 0.873
1), ('one', 0.8729), ('play', 0.8583), ('time', 0.8578), ('actor', 0.8577)]
```

```
compare("Man", model, 10, U_sigma_trunc, word_to_id, id_to_word)
```

```
Most similar words to 'Man':

SVD:  [('see', 0.7756), ('favorite', 0.7608), ('watch', 0.7506), ('must', 0.7409), ('Moon', 0.7404), ('superior', 0
.7351), ('far', 0.731), ('among', 0.7255), ('numerous', 0.7207), ('five', 0.7195)]

Word2Vec:  [('defeat', 0.3451), ('CID', 0.3369), ('wit', 0.3296), ('Rather', 0.324), ('male', 0.3233), ('futuristic
', 0.319), ('enter', 0.3147), ('exact', 0.3086), ('people', 0.3072), ('bring', 0.3043)]
```

```
compare("good", model, 10, U_sigma_trunc, word_to_id, id_to_word)
```

```
Most similar words to 'good':

SVD:  [('bad', 0.8227), ('make', 0.8078), ('movie', 0.7922), ('one', 0.7799), ('definitely', 0.7726), ('watch', 0.7
718), ('see', 0.7716), ('thing', 0.7712), ('overall', 0.7706), ('really', 0.7685)]

Word2Vec:  [('film', 0.8374), ('give', 0.8294), ('movie', 0.8107), ('make', 0.8107), ('even', 0.8035), ('like', 0.8
023), ('first', 0.7998), ('go', 0.7988), ('try', 0.798), ('play', 0.797)]
```

From comparison results above, Word2Vec method performs better in that:

1. Word2Vec give higher similarity score of similar word. In getting most similar word of “film” these two methods all return “movie” as the word with the highest similarity, while SVD gives similarity score of 0.774, which is lower than Word2Vec who gives similarity score of 0.9024.

2. Word2Vec return more related word. In getting most similar word of “Man”, it is noticed that Word2Vec return a word “male”, which is synonyms for “Man”; and in getting most similar word of “good”, SVD return “bad” as the most similar one, which is questionable.