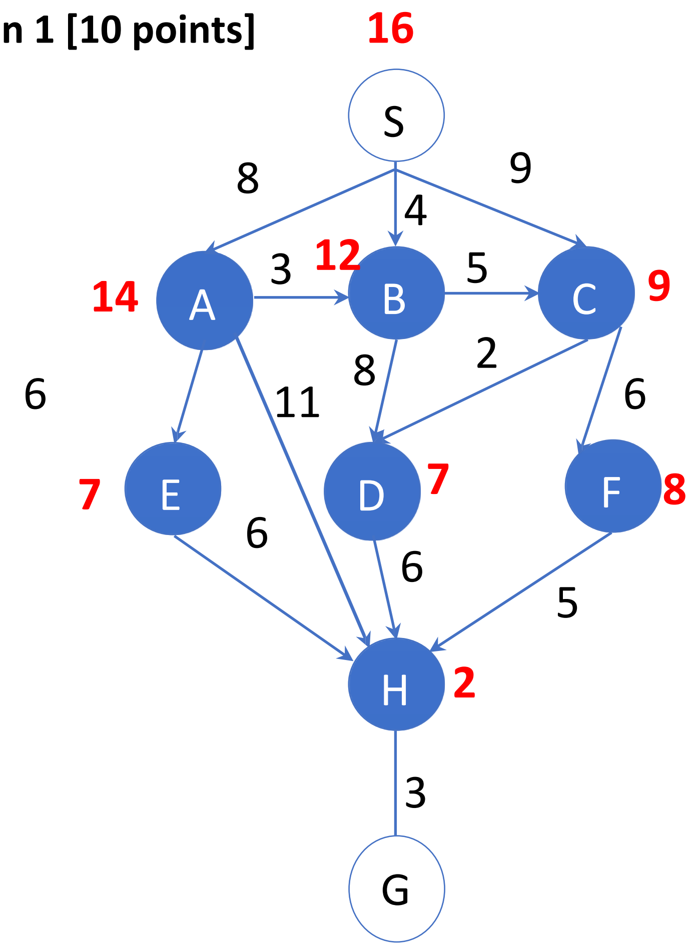
Assignment 2

Note: Your solution for this assignment should have two parts---a pdf document and code files.

- Have a **single** pdf document that shows your solution for different questions (show either numerical values if the question asks for it, and/or theoretical justification as required). Include in this pdf, the code you wrote for the solution for the respective question (if coding is required).
- Upload your real code files that you used to solve the particular question. Make sure your code is neatly organized per question, runs correctly, and has comments that highlight the part you implemented so that your TA can easily understand it
- Combine your solution pdf and code files in a single zip folder and upload it on the eLearn assignment folder
- Solution should be typeset using a profession software (word, keynote, latex etc). No handwritten solutions are allowed.
- **VERY IMPORTANT: This is an individual assignment. You SHOULD NOT collaborate with others. Supporting or indulging in plagiarism is a violation of academic policy and any such attempts will lead to strict disciplinary action.**

Question 1 [10 points]



For the above graph (heuristic values are provided in red color and actual costs are in black color), please provide answers to the following answers:

1. Is the heuristic admissible?

Response:
Yes, the above heuristic is admissible. Hereby, a heuristic is admissible if it never overestimates the cost of reaching the goal state. Thus, the costs that are estimated by the heuristic to reach the goal cannot be higher than the lowest possible cost from the current point in the path.

The minimum cost $h(n)$ to reach the goal from each $h(n) \leq c(n, g) \forall n$ is defined in the following table:

Node n	$h(n)$	$c(n, g)$	$h(n) \leq c(n, g)$
S	16	20	TRUE
A	14	14	TRUE
B	12	16	TRUE
C	9	11	TRUE
E	7	9	TRUE
D	7	9	TRUE
F	8	8	TRUE

H	2	3	TRUE

Node S:
The path starting from node S is $S \to C \to D \to H \to G$
$c(n, g) = 9 + 2 + 6 + 3 = 19$
$h(n) = 16$

Node A:
The path starting from node A is $A \to H \to G$
$c(n, g) = 11 + 3 = 14$
$h(n) = 14$

Node B:
The path starting from node B is $B \to C \to D \to H \to G$
$c(n, g) = 5 + 2 + 6 + 3 = 16$
$h(n) = 12$

Node C:
The path starting from node C is $C \to D \to H \to G$
$c(n, g) = 2 + 6 + 3 = 11$
$h(n) = 9$

Node D:
The path would be D H G
$c(n, g) = 6 + 3 = 9$
$h(n) = 7$

Node E:
The path starting from node E is $E \to H \to G$
$c(n, g) = 6 + 3 = 9$
$h(n) = 7$

Node F:
The path starting from node F is $F \to H \to G$
$c(n, g) = 5 + 3 = 8$
$h(n) = 8$

Node H:
The path starting from node H is $H \to G$
$c(n, g) = 3$
$h(n) = 2$

Hereby we can argue that the heuristic is admissible, as for all nodes it holds true that

$$h(n) \leq c(n, g) \forall n$$

2. Is the heuristic consistent?

Response:
No, the heuristic is not consistent. A heuristic is consistent if

I. The heuristic is admissible
II. The estimate is always less than or equal to the estimated distance from any neighbouring vertex to the goal, plus each cost of reaching the respective neighbours. In more formal terms:

$$h(n) \leq c(n,p) + h(p) \forall n$$

Hereby highlighting the cases in which the consistency is not true:

Node n	$h(n)$	p	$c(n,p) + h(p)$
S	16	A	20
S	16	B	16
S	16	C	18
A	14	B	15
A	14	E	13
A	14	H	13
B	12	C	14
B	12	D	15
C	9	D	9
C	9	F	14
D	7	H	8
E	7	H	8
F	8	H	7
H	2	G	3

As we can see in the above table there exist three distinct cases in which the property does not hold, thus the heuristic is not consistent.

3. Provide the search steps (as discussed in class) with DFS, BFS, BFS-Optimal, Best First Search and A* search.

Please note that S is start state and G is goal state.

Response:

Depth First Search (DFS)			
Step	Stack	Pop	Nodes to add
1	S	S	A^S, B^S, C^S
2	A^S, B^S, C^S	A^S	E^A, H^A
3	E^A, H^A, B^S, C^S	E^A	None
4	H^A, B^S, C^S	H^A	G^H
5	G^H, B^S, C^S	G^H	None

- DFS operates with a stack

- The path reached by DFS is $S \to A \to H \to G$
- Cost incurred: 8 + 11 + 3 = 22

Breadth First Search (BFS)			
Step	Open List	Pop	Nodes to add
1	S	S	A^S, B^S, C^S
2	A^S, B^S, C^S	A^S	E^A, H^A
3	B^S, C^S, E^A, H^A	B^S	D^B
4	C^S, E^A, H^A, D^B	C^S	F^C
5	E^A, H^A, D^B, F^C	E^A	None
6	H^A, D^B, F^C	H^A	G^H
7	D^B, F^C, G^H	D^B	None
8	F^C, G^H	F^C	None
9	G^H	G^H	None

- BFS operates with a queue
- The path reached by BFS is $S \to A \to H \to G$
- Cost incurred: 8 + 11 + 3 = 22

Breadth First Search (BFS- Optimal)			
Step	Open List	Pop	Nodes to add
1	S	S	$B^{S,4}, A^{S,8}, C^{S,9}$
2	$B^{S,4}, A^{S,8}, C^{S,9}$	$B^{S,4}$	$C^{B,9}, D^{B,12}$
3	$A^{S,8}, C^{B,9}, D^{B,12}$	$A^{S,8}$	$E^{A,14}, H^{A,19}$
4	$C^{B,9}, D^{B,12}, E^{A,14}, H^{A,19}$	$C^{B,9}$	$D^{C,11}, F^{C,15}$
5	$D^{C,11}, E^{A,14}, F^{C,15}, H^{A,19}$	$D^{C,11}$	$H^{D,17}$
6	$H^{D,17}, E^{A,14}, F^{C,15}$	$H^{D,17}$	$G^{H,20}$
7	$E^{A,14}, F^{C,15}, G^{H,20}$	$E^{A,14}$	$H^{E,20}$ **
8	$F^{C,15}, G^{H,20}$	$F^{C,15}$	$H^{F,20}$ **
9	$G^{H,20}$	$G^{H,20}$	None

** Not added to the list, as the node has already been explored

- BFS-Optimal operates with a priority queue
- The path reached by BFS is $S \to B \to C \to D \to H \to G$
- Cost incurred: 4 + 5 + 2 + 6 + 3 = 20

Best First Search			
Step	Open List	Pop	Nodes to add
1	S	S	$A^{S,14}, B^{S,12}, C^{S,9}$
2	$C^{S,9}, B^{S,12}, A^{S,14}$	$C^{S,9}$	$D^{C,7}, F^{C,8}$
3	$D^{C,7}, F^{C,8}, B^{S,12}, A^{S,14}$	$D^{C,7}$	$H^{D,2}$
4	$H^{D,2}, F^{C,8}, B^{S,12}, A^{S,14}$	$H^{D,2}$	$G^{H,0}$
5	$G^{H,0}, F^{C,8}, B^{S,12}, A^{S,14}$	$G^{H,0}$	None

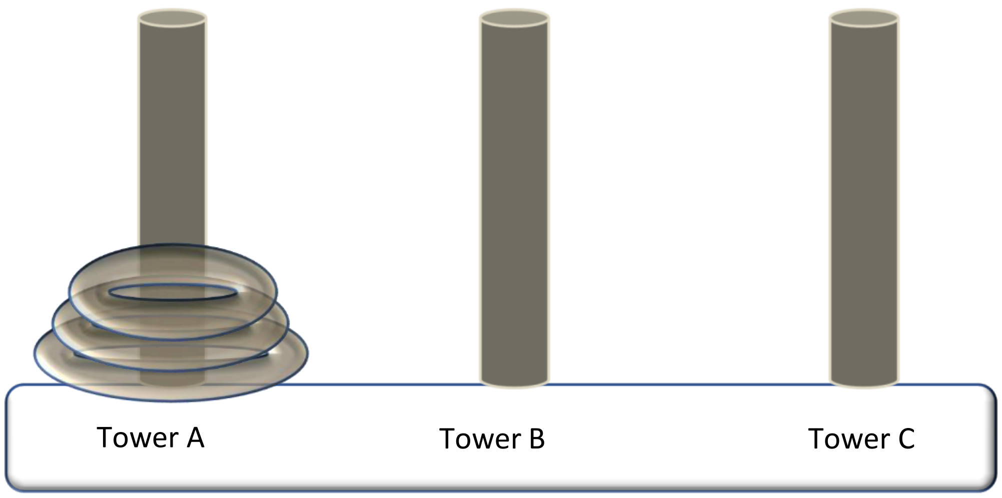
- Best First Search operates with a priority queue

- The path reached by BFS is $S \rightarrow C \rightarrow D \rightarrow H \rightarrow G$
- Cost incurred: 9 + 2 + 6 + 3 = 20

A* Search			
Step	Open List	Pop	Nodes to add
1	S	S^{16}	$B^{S,16}, C^{S,18}, A^{S,22}$
2	$B^{S,16}, C^{S,18}, A^{S,22}$	$B^{S,14}$	$C^{B,18}, D^{B,19}$
3	$C^{B,18}, D^{B,19}, A^{S,22}$	$C^{B,18}$	$D^{C,18}, F^{C,23}$
4	$D^{C,18}, A^{S,22}, F^{C,23}$	$D^{C,18}$	$H^{D,19}$
5	$H^{D,19}, A^{S,22}, F^{C,23}$	$H^{D,19}$	$G^{H,20}$
6	$G^{H,20}, A^{S,22}, F^{C,23}$	$G^{H,20}$	None

- A* operates with a priority queue
- The path reached by A* Search is $S \rightarrow B \rightarrow C \rightarrow D \rightarrow H \rightarrow G$
- Cost incurred: 4 + 5 + 2 + 6 + 3 = 20

Question 2 [10 points]: Towers of Hanoi



Towers of Hanoi is a famous problem where discs of varying sizes must be moved from tower A to tower C with the help of an intermediate tower, tower B in the least number of moves. The key constraint is that a bigger disc should never be put on top of a smaller disc on any tower. In case of 3 discs, the sequence of moves will thus be:

(1) Move top disc from tower A to tower C
(2) Move top disc from tower A to tower B
(3) Move top disc from tower C to tower B
(4) Move top disc from tower A to tower C
(5) Move top disc from tower B to tower A
(6) Move top disc from tower B to tower C
(7) Move top disc from tower A to tower C

Please answer the following questions:

- Given "N" discs, formulate towers of Hanoi problem as a search problem, i.e., indicate the states, actions, cost, successor states and objective.
- For using heuristic search method such as A*, provide an admissible heuristic and justify why it is admissible.

  Note: Make sure that heuristic is easy to compute, informative (i.e., do not assume heuristic value as zero for all the states) and intuitive enough so that you can justify that it is admissible.
- Show first three steps of DFS, and A* search. You can use the table structure as in question 1c to show different steps. Assume the start state is as shown in the figure with three rings placed on tower A.

Response:

1)

The tower of Hanoi-problem can be described as a typical recursive problem. A potential state representation would be to store three lists corresponding with the different discs on each peg. Hereby, it is to assume that the N discs D_n are numbered in increasing order from $1, \dots, n$, whereas one can then present each peg P_n as an ordered list of integers.

The initial and final states shall be defined as:

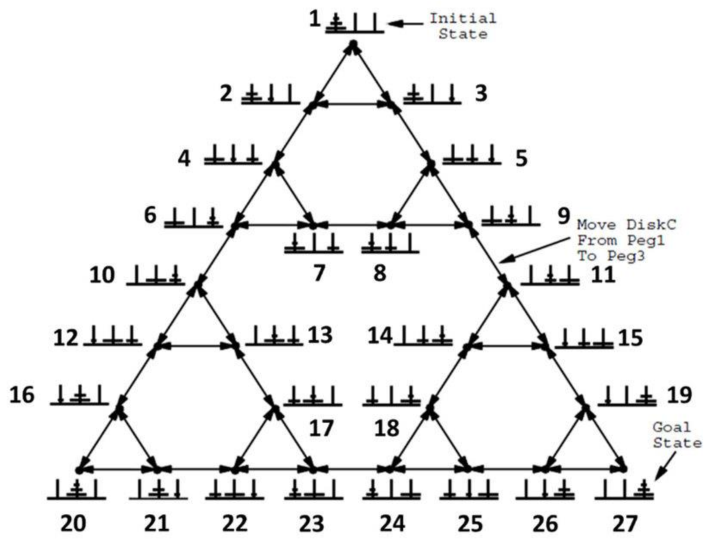
Initial State: $P_1\colon \{D_1, D_2, \dots D_n\}\ P_2\colon \{\}\ P_3\colon \{\}\ P_4\colon \{\}$

Final State: $P_1\colon \{\}\ P_2\colon \{\}\ \ P_3\colon \{\}\ \ P_4\colon \{D_1, D_2, \dots D_n\}$

All admissible actions are detailed by popping the top element of a list and then assigning it to another list with a bigger top element, or no element at all. The goal state of a configuration with n disks can be reached with $2^n - 1$ steps, due to the recursive nature of the properties and e.g. as the problem can be solved with 1 disk in 1 move, with 2 disks in 3 moves, with 3 disks in 7 moves and with 4 disk in 15 moves.

The cost of each move shall be one, to provide penalties for longer suboptimal solutions and the final objective is to move all pegs in an ordered manner from peg A to peg C. Each wrong move encounters an extra cost of at least 2 to the solution.

These properties can be shown more clearly when referring to a graph of all the possible walks available given D = 3.



Hereby, given the situation of the smallest disk being on a selected peg, it is not allowed to move onto a disk with a larger smallest disk.
The admissible actions are defined by disk D_i being able to be moved from P_i *to* P_j if:

- D_i *is the topmost disk on* P_i
- P_j *is either empty, or the value of the topmost disk of* P_j, D_{jt} *is bigger than* D_i.

The costs are uniform, as each move consists of an action, whereas actions need to be minimized.
2) To be an admissible heuristic again, the estimated cost of the heuristic can never be bigger than the real cost encountered. In more formal terms:

$$h(n) \leq c(n, g) \forall n$$

A* search always calculates $g(n)$, which is the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
Additionally, A* search adds $g(n)$ with $h(n)$ at each step, which consists of the estimated movement cost to move from that given square on the grid to the destination to predict a more accurate cost for each action at any given state. At any given state A* search adds each encountered node to an open list and ordering it regarding their size and performs the lowest costing action.
Thus, the elements in the open list are ordered based on their value of the following equation:

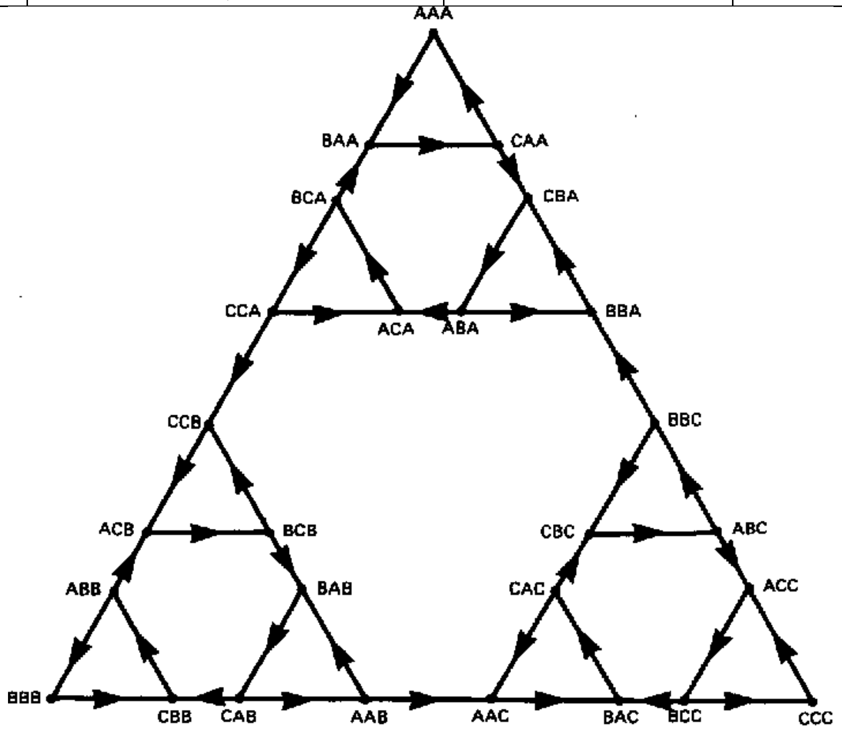
$$f(n) = g(n) + h(n)$$

Given that the heuristic estimation $g(n)$ for each node in the Tower of Hanoi follows the recursive property of finding an optimal solution in $2^n - 1$ steps, we can define $g(n)$ to be equal to $(2^n - 1) - k$, whereas k details the number of moves needed to reach this instance.
In regards, to the estimability of $h(n)$ guaranteeing that each path node of each node is admissible, it is important to consider the previously annotated $2^n - 1$ steps needed. Hereby the heuristic may be estimated as implies indirectly, that each step there at most $2^j_k - 1$ steps needed heuristically, whereas j represents the largest disk that is misplaced at step k.
3) The main differences hereby lie that DFS does not compute future prices and thus it is possible in this case that it will choose to move the smallest disk towards the middle peg, B,

in the first move and not the target peg, C, as it is operating with a uninformed search mode. Thus, it already has a 50% disadvantage of not solving the problem about optimality at its first action and incurring an extra cost of two moves, regarding the action of moving the smallest disk to the suboptimal.
In other terms, uniformed search algorithms such as DFS will run into trouble fairly quickly, as the search space is getting big fairly quickly and DFS risks exploring the full search space $O(n)$ before converging to the solution, which encounters problems with memory in comparison to informed search algorithms, as the cost of holding $3(n)$ in memory is fairly large when scaling the Tower of Hanoi-problem to a larger size.
Comparing three moves of both A* Search and DFS:

	A* Search			
Step	Open List	f(n) = g(n) + h(n)	Pop	Nodes to add
1	AAA^7	7	AAA^7	CAA^6, BAA^8
2	CAA^6, BAA^8	6	CAA^6	CBA^5,
3	CBA^5, BAA^8	5	CBA^5	BBA^4

Depth First Search (DFS)			
Step	Open List	Pop	Nodes to add
1	AAA	AAA	BAA, CAA
2	BAA, CAA	BAA	BCA
3	CCA, CAA	CCA	CCB



Question 3 [10 Points]

Taxi drivers in Singapore can pick up customers from any location that is not on highways. However, they require guidance on where to pick up customers when they do not have customers on board and there are no bookings. In this question, we address this guidance problem.

There are four locations: L1, L2, L3 and L4 from where taxi drivers can pick up and drop off customers. At any decision epoch, the chances of the taxi driver picking up a customer from different locations are provided in Table 1 below:

Location	**Chance of finding customer**
L1	0.2
L2	0.6
L3	0.5
L4	0.7

Table 1

Once the taxi driver picks up a customer, the customer determines the destination. Observed probabilities (from past data) of a customer starting from a source location and going to a destination location are given below:

Source→ Destination	**Probability**
L1 → L2	0.5
L1 → L3	0.3
L1 → L4	0.2
L2 → L1	0.6
L2 → L3	0.4
L3 → L1	0.5
L3 → L4	0.5
L4 → L1	0.35
L4 → L2	0.65

Table 2

Just to avoid any confusion, first row of Table 2 below the heading row indicates that a customer picked up from L1 gets dropped off at L2 with 0.5 probability.
Travel fare between different locations are as follows:

Source→ Destination	**Fare**
L1 → L2	9$
L1 → L3	7$
L1 → L4	12$
L2 → L1	9$
L2 → L3	10$
L3 → L1	10$
L3 → L4	13$
L4 → L1	11$
L4 → L2	9$

If a source destination pair does not appear in the above table, it indicates the fare is 0.

Cost of travelling between locations is as follows:

Source→ Destination	**Cost**
L1 → L2	1$
L1 → L3	1.5$
L1 → L4	1.25$
L2 → L1	1$
L2 → L3	0.75$
L3 → L1	1.5$
L3 → L4	0.8$
L4 → L1	1.25$
L4 → L2	1.00$

*****If a source destination pair does not appear in the above table, it indicates the cost is infinity***

The taxi driver can either pickup from the current location or move to another location. Pickup corresponds to picking up a customer (if one is found) and dropping them of at their destination. Pickup action succeeds with probabilities specified in *Table 1* and if the taxi picks up a customer, destination location is determined by the probabilities in *Table 2*. When Pickup action fails, the taxi remains in its current location. Move to another location is always successful and taxi moves to the desired location with probability 1.

Both pickup and move actions take one-time step each. Please provide the following:

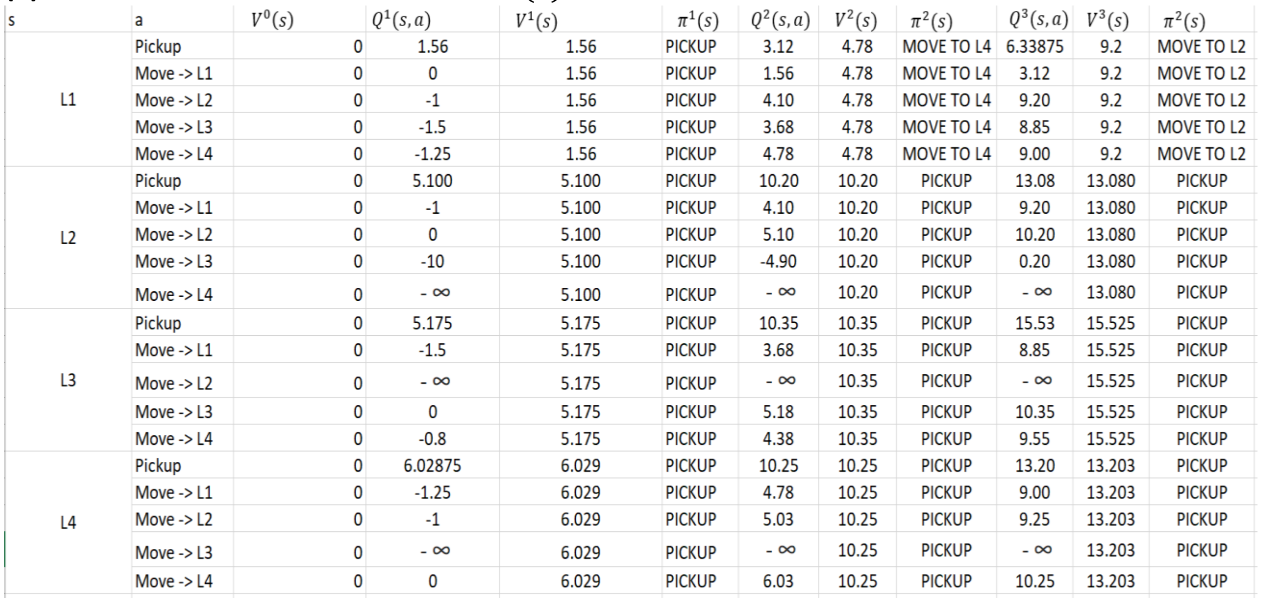
1. **You need to provide an MDP model that guides the taxi driver on "move and pickup customers"**. MDP is the tuple <S, A, P, R>, i.e., the states, actions, transition probability matrices (for different actions) and reward functions.
   The MDP model can be described by the following graph:
   Hereby states, actions, transition probabilities and reward functions are detailed in each column.

s (Current State)	a (Action)	s' (New State)	P(s'|s,a)	R(s,a,s')
L1	Pickup -> L1	L1	1 - 0.2 = 0.8	0
	Pickup -> L2	L2	0.2 x 0.5 = 0.1	9 - 1 = 8
	Pickup -> L3	L3	0.2 x 0.3 = 0.06	7 - 1.5 = 5.5
	Pickup -> L4	L4	0.2 x 0.2 = 0.04	12 - 1.25 = 9.75
	Move -> L1	L1	1	0
	Move -> L2	L2	1	0 - 1 = -1
	Move -> L3	L3	1	0 - 1.5 = -1.5
	Move -> L4	L4	1	-1.25
L2	Pickup -> L1	L1	0.6 x 0.6 = 0.36	9 - 1 = 8
	Pickup -> L2	L2	1 - 0.6 = 0.4	0
	Pickup -> L3	L3	0.6 x 0.4 = 0.24	10 - 0.75 = 9.25
	Pickup -> L4	L4	0	$-\infty$
	Move -> L1	L1	1	0 - 1 = -1
	Move -> L2	L2	1	0
	Move -> L3	L3	1	0 - 10 = -10
	Move -> L4	L4	1	$-\infty$
L3	Pickup -> L1	L1	0.5 x 0.5 = 0.25	10 - 1.5 = 8.5
	Pickup -> L2	L2	0	$-\infty$
	Pickup -> L3	L3	1 - 0.5 = 0.5	0
	Pickup -> L4	L4	0.5 x 0.5 = 0.25	13 - 0.8 = 12.2
	Move -> L1	L1	1	0 - 1.5 = -1.5
	Move -> L2	L2	1	$-\infty$
	Move -> L3	L3	1	0
	Move -> L4	L4	1	0 - 0.8 = -0.8
L4	Pickup -> L1	L1	0.7 x 0.35 = 0.245	11 - 1.25 = 9.75
	Pickup -> L2	L2	0.7 x 0.65 = 0.455	9 - 1 = 8
	Pickup -> L3	L3	0	$-\infty$
	Pickup -> L4	L4	1 - 0.7 = 0.3	0
	Move -> L1	L1	1	0 - 1.25 = -1.25
	Move -> L2	L2	1	0 - 1 = - 1
	Move -> L3	L3	1	$-\infty$
	Move -> L4	L4	1	0

2. **After providing the MDP, show three iterations of value iteration (obtained as output from the code). Initialize $\forall s, V^0(s) = 0$ $\quad and \quad$ calculate**

(i) $\quad \forall s, V^1(s),\ V^2(s) \ \ and \ \ V^3(s)$

(ii) $\quad \forall s, \pi^1(s),\ \pi^2(s) \ \ and \ \ \pi^3(s)$



s	a	$V^0(s)$	$Q^1(s,a)$	$V^1(s)$	$\pi^1(s)$	$Q^2(s,a)$	$V^2(s)$	$\pi^2(s)$	$Q^3(s,a)$	$V^3(s)$	$\pi^2(s)$
	Pickup	0	1.56	1.56	PICKUP	3.12	4.78	MOVE TO L4	6.33875	9.2	MOVE TO L2
	Move -> L1	0	0	1.56	PICKUP	1.56	4.78	MOVE TO L4	3.12	9.2	MOVE TO L2
L1	Move -> L2	0	-1	1.56	PICKUP	4.10	4.78	MOVE TO L4	9.20	9.2	MOVE TO L2
	Move -> L3	0	-1.5	1.56	PICKUP	3.68	4.78	MOVE TO L4	8.85	9.2	MOVE TO L2
	Move -> L4	0	-1.25	1.56	PICKUP	4.78	4.78	MOVE TO L4	9.00	9.2	MOVE TO L2
	Pickup	0	5.100	5.100	PICKUP	10.20	10.20	PICKUP	13.08	13.080	PICKUP
	Move -> L1	0	-1	5.100	PICKUP	4.10	10.20	PICKUP	9.20	13.080	PICKUP
L2	Move -> L2	0	0	5.100	PICKUP	5.10	10.20	PICKUP	10.20	13.080	PICKUP
	Move -> L3	0	-10	5.100	PICKUP	-4.90	10.20	PICKUP	0.20	13.080	PICKUP
	Move -> L4	0	$-\infty$	5.100	PICKUP	$-\infty$	10.20	PICKUP	$-\infty$	13.080	PICKUP
	Pickup	0	5.175	5.175	PICKUP	10.35	10.35	PICKUP	15.53	15.525	PICKUP
	Move -> L1	0	-1.5	5.175	PICKUP	3.68	10.35	PICKUP	8.85	15.525	PICKUP
L3	Move -> L2	0	$-\infty$	5.175	PICKUP	$-\infty$	10.35	PICKUP	$-\infty$	15.525	PICKUP
	Move -> L3	0	0	5.175	PICKUP	5.18	10.35	PICKUP	10.35	15.525	PICKUP
	Move -> L4	0	-0.8	5.175	PICKUP	4.38	10.35	PICKUP	9.55	15.525	PICKUP
	Pickup	0	6.02875	6.029	PICKUP	10.25	10.25	PICKUP	13.20	13.203	PICKUP
	Move -> L1	0	-1.25	6.029	PICKUP	4.78	10.25	PICKUP	9.00	13.203	PICKUP
L4	Move -> L2	0	-1	6.029	PICKUP	5.03	10.25	PICKUP	9.25	13.203	PICKUP
	Move -> L3	0	$-\infty$	6.029	PICKUP	$-\infty$	10.25	PICKUP	$-\infty$	13.203	PICKUP
	Move -> L4	0	0	6.029	PICKUP	6.03	10.25	PICKUP	10.25	13.203	PICKUP

Question 4 [10 Points]

Game is played with an infinite deck of cards

Each draw from deck results in value between 1 and 10 (uniformly distributed) with a color of red (probability 1/4) or black (probability 3/4)
At the start of the game both the player and dealer draw one black card

Each turn the player may either stick or hit

If player hits, then she draws another card from the deck

If player sticks, she receives no further cards

The value of player cards is added (black cards) or subtracted (red cards)

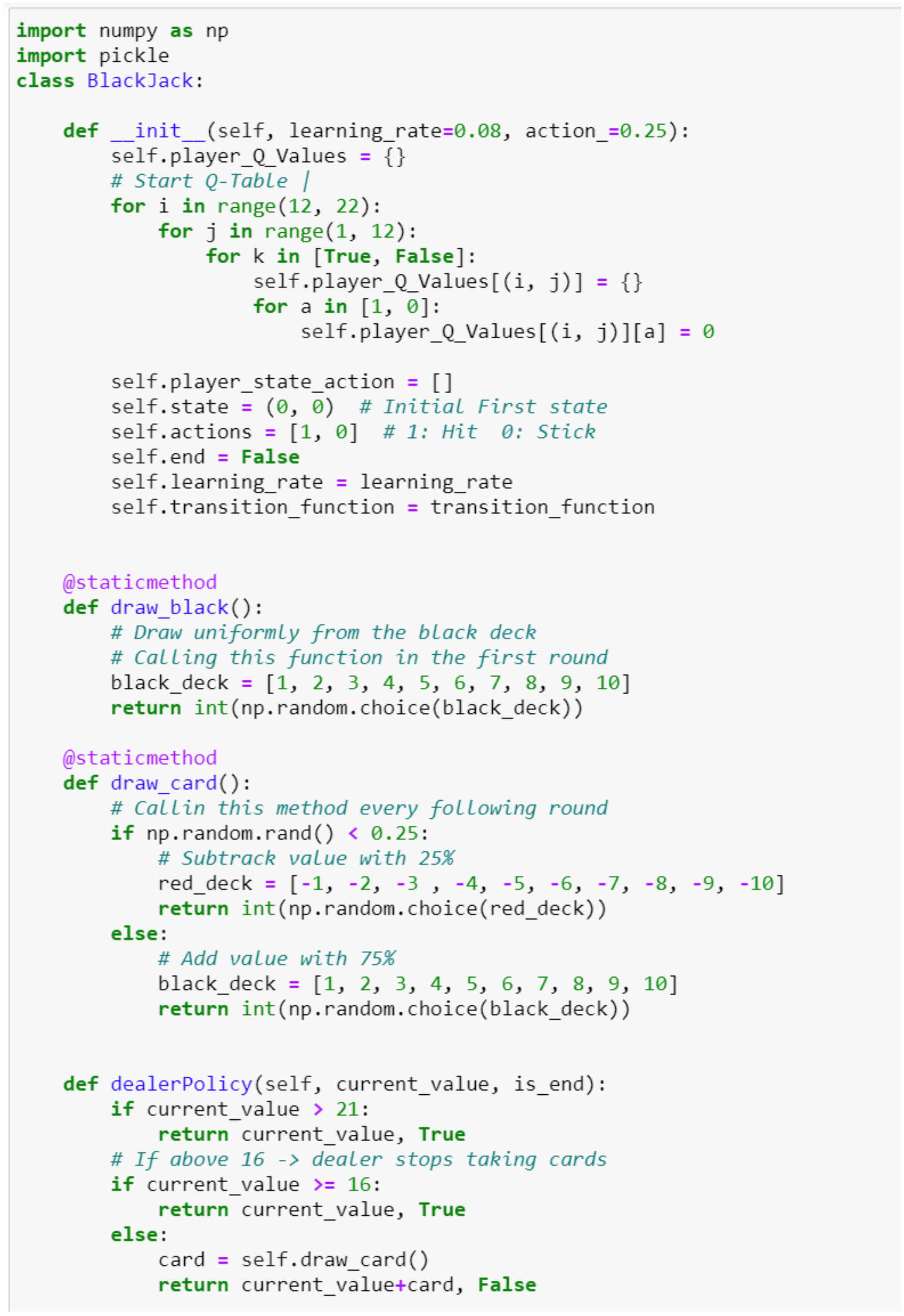
If player's sum exceeds 21, or becomes less than 1, then she "goes bust" and loses the game (reward -1)

If the player sticks, then the dealer starts taking turns. The dealer always sticks on any sum of 16 or greater, and hits otherwise. If the dealer goes bust, then the player wins; otherwise the outcome-win (reward +1), lose (reward -1), or draw (reward 0) - is the player with the largest sum.

(1) Build a simulator that simulates the draw of cards and play of the game between player and dealer.

Response:
By repetitively sampling the Blackjack-type and saving the outcomes in Q-tables, the agent is able to learn how to outsmart the dealer and win games. Given that there is the complete information of the game, the player agent can learn from the dealer's attitudes, which unlike the player sticks on any value of the hand above 16. Hereby, the player agent can use this small distinction to win more games than the dealer by evaluating risk and learning from past game's outcomes.

The game was implemented with the following code:



```python
import numpy as np
import pickle
class BlackJack:

    def __init__(self, learning_rate=0.08, action_=0.25):
        self.player_Q_Values = {}
        # Start Q-Table |
        for i in range(12, 22):
            for j in range(1, 12):
                for k in [True, False]:
                    self.player_Q_Values[(i, j)] = {}
                    for a in [1, 0]:
                        self.player_Q_Values[(i, j)][a] = 0

        self.player_state_action = []
        self.state = (0, 0)  # Initial First state
        self.actions = [1, 0]  # 1: Hit  0: Stick
        self.end = False
        self.learning_rate = learning_rate
        self.transition_function = transition_function


    @staticmethod
    def draw_black():
        # Draw uniformly from the black deck
        # Calling this function in the first round
        black_deck = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
        return int(np.random.choice(black_deck))

    @staticmethod
    def draw_card():
        # Callin this method every following round
        if np.random.rand() < 0.25:
            # Subtrack value with 25%
            red_deck = [-1, -2, -3 , -4, -5, -6, -7, -8, -9, -10]
            return int(np.random.choice(red_deck))
        else:
            # Add value with 75%
            black_deck = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
            return int(np.random.choice(black_deck))


    def dealerPolicy(self, current_value, is_end):
        if current_value > 21:
            return current_value, True
        # If above 16 -> dealer stops taking cards
        if current_value >= 16:
            return current_value, True
        else:
            card = self.draw_card()
            return current_value+card, False
```

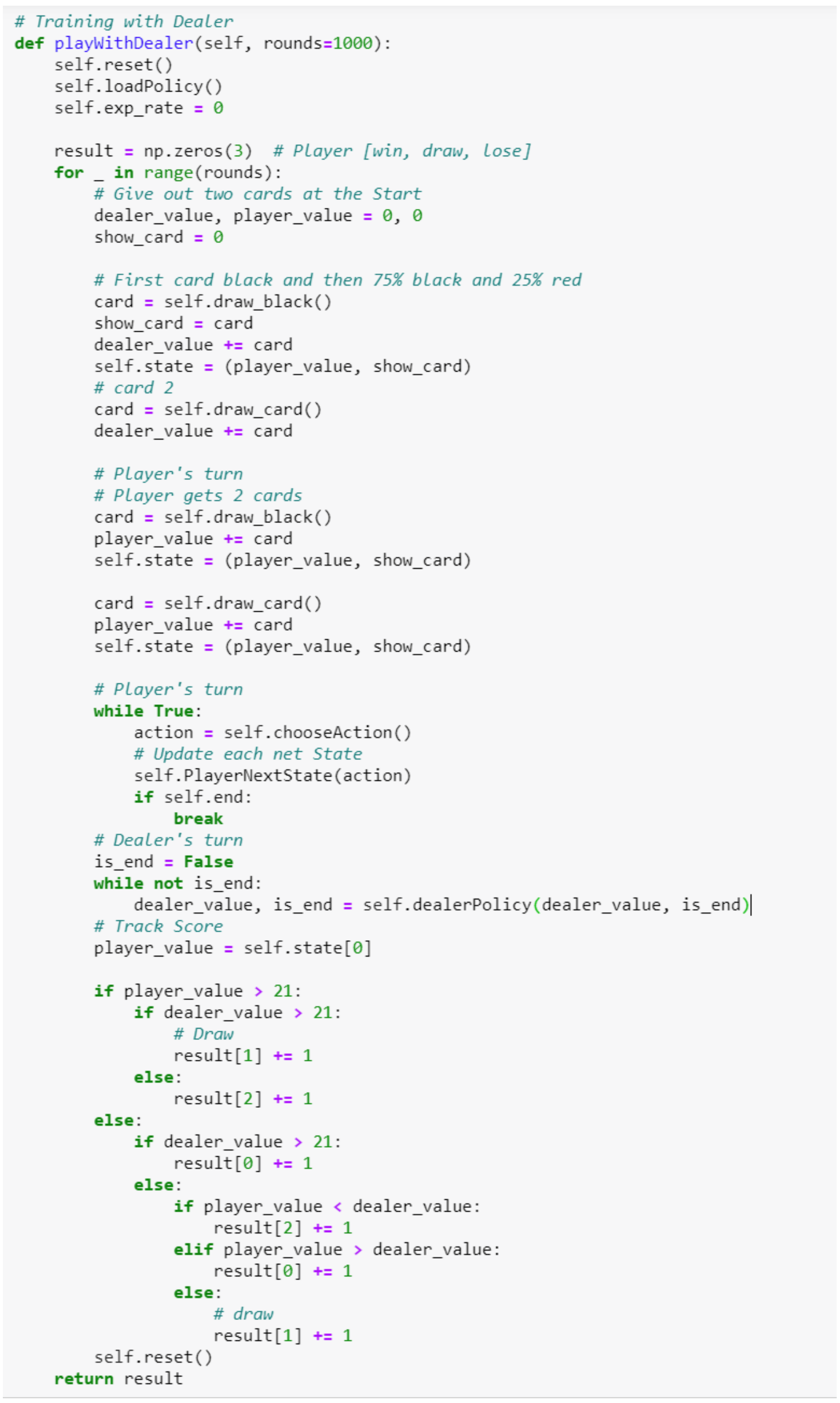
```python
def dealerPolicy(self, current_value, is_end):
    if current_value > 21:
        return current_value, True

    # If above 16 -> dealer stops taking cards
    if current_value >= 16:
        return current_value, True
    else:
        card = self.draw_card()
        return current_value+card, False


def chooseAction(self):
    current_value = self.state[0]
    if current_value <= 11:
        return 1

    elif np.random.uniform(0, 1) <= self.exp_rate:
        action = np.random.choice(self.actions)
    else:
        v = -np.inf
        action = 0
        #print(self.player_Q_Values)

        for a in self.player_Q_Values[self.state]:
            #print('update state', self.player_Q_Values[self.state])

            if self.player_Q_Values[self.state][a] > v:
                action = a
                #print('update state with action', self.state[a])
                v = self.player_Q_Values[self.state][a]
    return action
```

```python
def PlayerNextState(self, action):
    current_value = self.state[0]
    show_card = self.state[1]

    if current_value > 21:
        self.end = True
        self.state = (current_value, show_card) # End Turn
        return

    if action:
        card = self.draw_card()
        current_value += card
    else:
        self.end = True
    # If getting above 21 in the round
    if current_value > 21:
        self.end = True
    self.state = (current_value, show_card)

def _KeepScore(self, player_value, dealer_value, is_end=True):
    reward = 0
    if is_end:
        if player_value > 21:
            if dealer_value > 21:
                reward = 0
                print('Both over 21 - Draw')
            else:
                reward = -1
                print('Dealer won the round')
        else:
            if dealer_value > 21:
                reward = 1
                print('Player won the round')
            else:
                if player_value < dealer_value:
                    reward = -1
                    print('Dealer won the round')
                elif player_value > dealer_value:
                    reward = 1
                    print('Player won the round')
                else:
                    reward = 0
                    print('Draw')

    # Backpropagate each Reward
    for s in reversed(self.player_state_action):
        state, action = s[0], s[1]
        reward = self.lr*(reward - self.player_Q_Values[state][action])
        self.player_Q_Values[state][action] += reward

def reset(self):
    self.player_state_action = []
    self.state = (0, 0)  # initial state
    self.end = False
```

```python
def play(self, rounds=1000):
    wins = np.zeros(2)
    for i in range(rounds):
        if i % 1000 == 0:
            print("round", i)

        # Giving out two cards each (21 can't be reached with 10 as max number)
        dealer_value, player_value = 0, 0
        show_card = 0

        # Player receives 2 cards, first black

        card = self.draw_black()
        player_value += card
        self.state = (player_value, show_card)

        card = self.draw_card()
        player_value += card
        self.state = (player_value, show_card)

        # Dealer receives 2 cards, first black

        card = self.draw_black()
        show_card = card
        dealer_value += card

        self.state = (player_value, show_card)
        card = self.draw_card()
        dealer_value += card

        # Play the game until a player sticks
        while True:
            action = self.chooseAction()
            if self.state[0] >= 12:
                self.player_state_action.append([self.state, action])
            # Update player's next state
            self.PlayerNextState(action)
            if self.end:
                break

        # Dealer's turn
        is_end = False
        while not is_end:
            dealer_value, is_end = self.dealerPolicy(dealer_value, is_end)

        # Detail Winner
        # Rewards + update Q values
        player_value = self.state[0]
        print("Player value {} | dealer value {}".format(player_value, dealer_value))
        self._KeepScore(player_value, dealer_value)
        self.reset()
    print(wins)
```

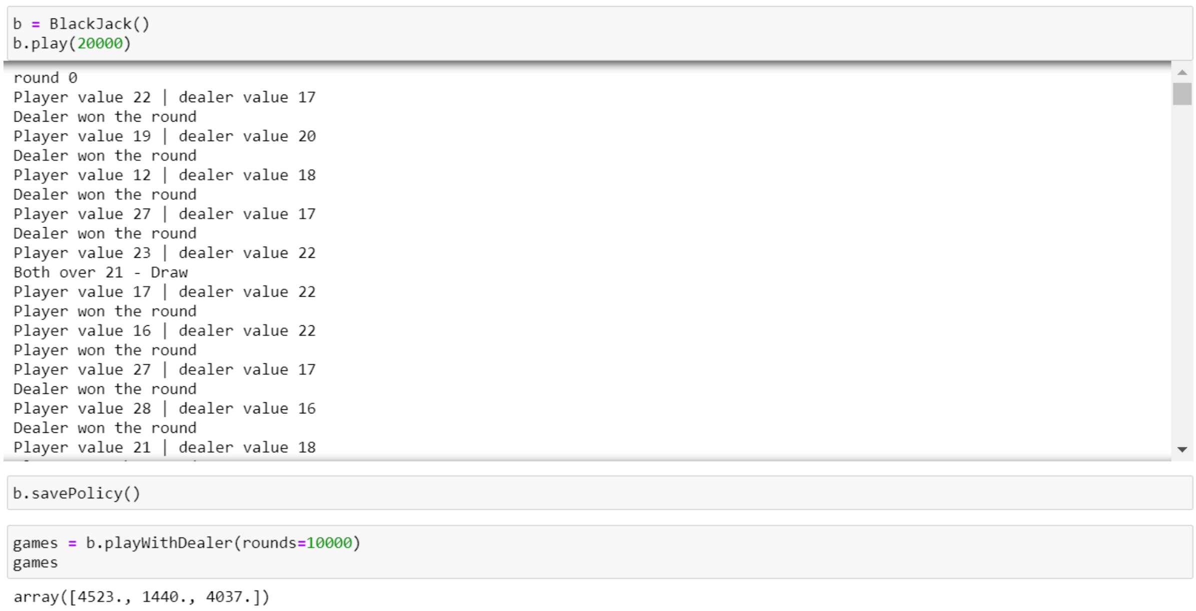
```python
# Saving the policy
def savePolicy(self, file="policy"):
    fw = open(file, 'wb')
    pickle.dump(self.player_Q_Values, fw)
    fw.close()

def loadPolicy(self, file="policy"):
    fr = open(file,'rb')
    self.player_Q_Values = pickle.load(fr)
    fr.close()
```

Followingly, implementing the auxiliary function to test the plays with the dealer:



```python
# Training with Dealer
def playWithDealer(self, rounds=1000):
    self.reset()
    self.loadPolicy()
    self.exp_rate = 0

    result = np.zeros(3)  # Player [win, draw, lose]
    for _ in range(rounds):
        # Give out two cards at the Start
        dealer_value, player_value = 0, 0
        show_card = 0

        # First card black and then 75% black and 25% red
        card = self.draw_black()
        show_card = card
        dealer_value += card
        self.state = (player_value, show_card)
        # card 2
        card = self.draw_card()
        dealer_value += card

        # Player's turn
        # Player gets 2 cards
        card = self.draw_black()
        player_value += card
        self.state = (player_value, show_card)

        card = self.draw_card()
        player_value += card
        self.state = (player_value, show_card)

        # Player's turn
        while True:
            action = self.chooseAction()
            # Update each net State
            self.PlayerNextState(action)
            if self.end:
                break
        # Dealer's turn
        is_end = False
        while not is_end:
            dealer_value, is_end = self.dealerPolicy(dealer_value, is_end)
        # Track Score
        player_value = self.state[0]

        if player_value > 21:
            if dealer_value > 21:
                # Draw
                result[1] += 1
            else:
                result[2] += 1
        else:
            if dealer_value > 21:
                result[0] += 1
            else:
                if player_value < dealer_value:
                    result[2] += 1
                elif player_value > dealer_value:
                    result[0] += 1
                else:
                    # draw
                    result[1] += 1
        self.reset()
    return result
```

Training the player agent and followingly testing the out how the player agent fares in playing against the dealer after training.



```
b = BlackJack()
b.play(20000)
```

```
round 0
Player value 22 | dealer value 17
Dealer won the round
Player value 19 | dealer value 20
Dealer won the round
Player value 12 | dealer value 18
Dealer won the round
Player value 27 | dealer value 17
Dealer won the round
Player value 23 | dealer value 22
Both over 21 - Draw
Player value 17 | dealer value 22
Player won the round
Player value 16 | dealer value 22
Player won the round
Player value 27 | dealer value 17
Dealer won the round
Player value 28 | dealer value 16
Dealer won the round
Player value 21 | dealer value 18
```

```
b.savePolicy()
```

```
games = b.playWithDealer(rounds=10000)
games
```

```
array([4523., 1440., 4037.])
```

The agent wins in about 45.2% of the cases against the dealer, who wins still in around 40.4% of the cases and about 14.4% of the cases end in a draw. Given that the dealer is not utilizing mostly the whole range of achievable values, e.g. not taking any cards above the value of 16 the player agent has the ability to learn and utilize this to his personal advantage.
Within the constructor-function the self.player_Q_Values-method is created, which initialises the estimates for the combination of actions at each state, which is continuously being updated at each episode. Hereby, the learning rate, lr, is determining the update speed and exp_rate is assigned manually to determine how inclined an agent is to change in between respective states.
The agent performs in each state s an action a, whether to hit or stick, ending up at the new state s' and lending the agent a reward r. Hereby, as expected an episode is defined as:

$$(s, a, s', r)$$

Each time a card is being dealt (other than the very first round), np.random.rand() calculates in whether to assign a player with a black card with 75% probability, or with a red card a 25% probability by accessing the drawCard()-function. After receiving the card the player agent is determining through the chooseAction()-function a strategy on whether continuing to hit, or to stick. Hereby, this policy is a balancing of exploration and exploitation, whereas through continuous learning the player agent learns to take the action that leads to the highest current estimated rewards based on the Q-estimates.
Furthermore, after taking an action the PlayerNextState()-function determines whether the episode has ended, or if the player agent is still able to take another action.

At the end of each round i.e. an episode the player agents receives a reward for the action taken in the past round and the values within the self.player_Q_Values-method are being updated.

The agent thus at each step computes the intermediate q-values

$$\hat{Q}(s,a) = R(s,a,s') + \gamma \max_{a'} Q_k(s',a').$$

Hereby, the player agent incorporates new evidence into the Q-table

$$Q_{k+1}(s,a) = Q_k(s,a) + \alpha\big(\hat{Q}(s,a) - Q_k(s,a)\big)$$

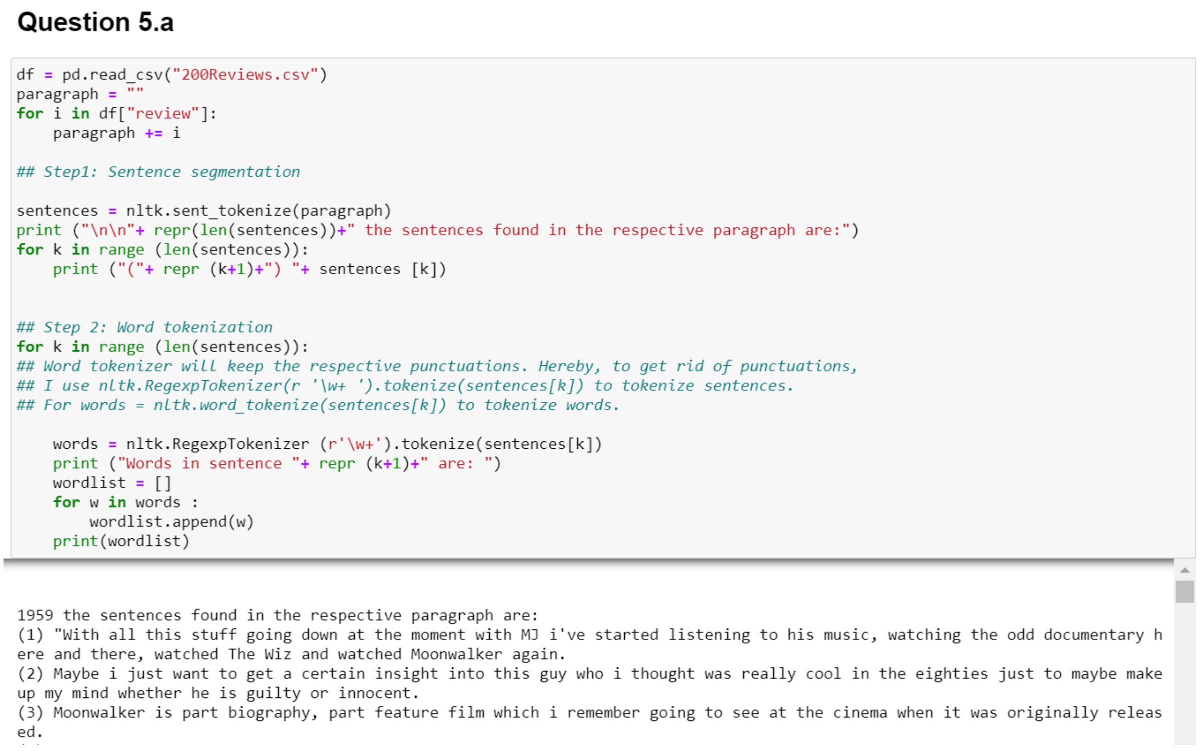
Incorporating a variable approach through algorithms such as simulated annealing to better estimate the optimal values e.g. the weights for the learning rate, as well as the exp_rate would potentially help improve the current values to above 50% total win rates.

Question 5 [10 Points]

In this question, you will employ Singular Value Decomposition to obtain word embeddings and compare the generated word embeddings with the word embeddings generated using word2vec. The corpus (or dataset) to be considered is “200Reviews.csv”. You need to do the following:

(a) Parse the reviews in “200Reviews.csv”, i.e., divide reviews into sentences and sentences into words and remove the stop words. You can employ the “NLP-pipeline-example.ipynb” example we talked about in class.

Response:



Question 5.a

```
df = pd.read_csv("200Reviews.csv")
paragraph = ""
for i in df["review"]:
    paragraph += i

## Step1: Sentence segmentation

sentences = nltk.sent_tokenize(paragraph)
print ("\n\n"+ repr(len(sentences))+" the sentences found in the respective paragraph are:")
for k in range (len(sentences)):
    print ("("+ repr (k+1)+") "+ sentences [k])


## Step 2: Word tokenization
for k in range (len(sentences)):
## Word tokenizer will keep the respective punctuations. Hereby, to get rid of punctuations,
## I use nltk.RegexpTokenizer(r '\w+ ').tokenize(sentences[k]) to tokenize sentences.
## For words = nltk.word_tokenize(sentences[k]) to tokenize words.

    words = nltk.RegexpTokenizer (r'\w+').tokenize(sentences[k])
    print ("Words in sentence "+ repr (k+1)+" are: ")
    wordlist = []
    for w in words :
        wordlist.append(w)
    print(wordlist)
```

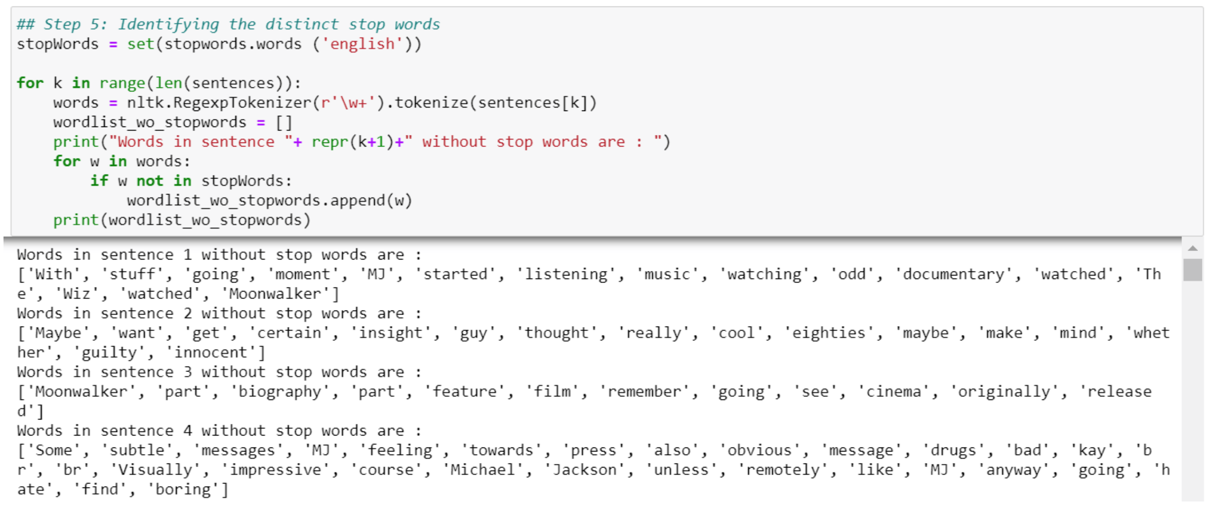
```
1959 the sentences found in the respective paragraph are:
(1) "With all this stuff going down at the moment with MJ i've started listening to his music, watching the odd documentary h
ere and there, watched The Wiz and watched Moonwalker again.
(2) Maybe i just want to get a certain insight into this guy who i thought was really cool in the eighties just to maybe make
up my mind whether he is guilty or innocent.
(3) Moonwalker is part biography, part feature film which i remember going to see at the cinema when it was originally releas
ed.
```

```
## Step 3: Predicting parts off speech for each token.
## Tagging all words:

## nltk.help.upenn_tagset () details the description of each of POS-Tag.
## Uncomment following line to print the list of all tags

#nltk.help.upenn_tagset()
#nltk.download('tagsets')

for k in range(len(sentences)):
    words = nltk.RegexpTokenizer (r'\w+').tokenize(sentences[k])
    tagged_words = nltk.pos_tag(words)
    print("Tagged Words in sentence "+ repr (k+1)+" are : ")
    print(tagged_words)
```

```
Tagged Words in sentence 1 are :
[('With', 'IN'), ('all', 'PDT'), ('this', 'DT'), ('stuff', 'NN'), ('going', 'VBG'), ('down', 'RP'), ('at', 'IN'), ('the', 'D
T'), ('moment', 'NN'), ('with', 'IN'), ('MJ', 'NNP'), ('i', 'NN'), ('ve', 'NN'), ('started', 'VBD'), ('listening', 'VBG'),
('to', 'TO'), ('his', 'PRP$'), ('music', 'NN'), ('watching', 'VBG'), ('the', 'DT'), ('odd', 'JJ'), ('documentary', 'NN'), ('h
ere', 'RB'), ('and', 'CC'), ('there', 'RB'), ('watched', 'VBD'), ('The', 'DT'), ('Wiz', 'NNP'), ('and', 'CC'), ('watched', 'V
BD'), ('Moonwalker', 'NNP'), ('again', 'RB')]
Tagged Words in sentence 2 are :
[('Maybe', 'RB'), ('i', 'JJ'), ('just', 'RB'), ('want', 'VBP'), ('to', 'TO'), ('get', 'VB'), ('a', 'DT'), ('certain', 'JJ'),
('insight', 'NN'), ('into', 'IN'), ('this', 'DT'), ('guy', 'NN'), ('who', 'WP'), ('i', 'VBP'), ('thought', 'VBN'), ('was', 'V
BD'), ('really', 'RB'), ('cool', 'JJ'), ('in', 'IN'), ('the', 'DT'), ('eighties', 'NNS'), ('just', 'RB'), ('to', 'TO'), ('may
be', 'RB'), ('make', 'VB'), ('up', 'RP'), ('my', 'PRP$'), ('mind', 'NN'), ('whether', 'IN'), ('he', 'PRP'), ('is', 'VBZ'),
('guilty', 'JJ'), ('or', 'CC'), ('innocent', 'JJ')]
```

```
## Step 4: Text Lemmatization
## Wordnet Lemmatizer and the the standard NLTK POS-tags are treebank tags
## Thus, we need to convert the treebank tag to wordnet tags.

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 ''

wordnet_lemmatizer = WordNetLemmatizer()

for k in range (len(sentences)):
    words = nltk . RegexpTokenizer (r'\w+').tokenize(sentences[k])
    tagged_words = nltk.pos_tag(words)
    lemmatized_wordlist =[]
    print ("Words: Lemmatized Word in sentence "+ repr (k+1)+" are the following: ")
    for w in tagged_words :
        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
        lemmatized_wordlist.append((w[0],lemmatizedword))
    print(lemmatized_wordlist)
```

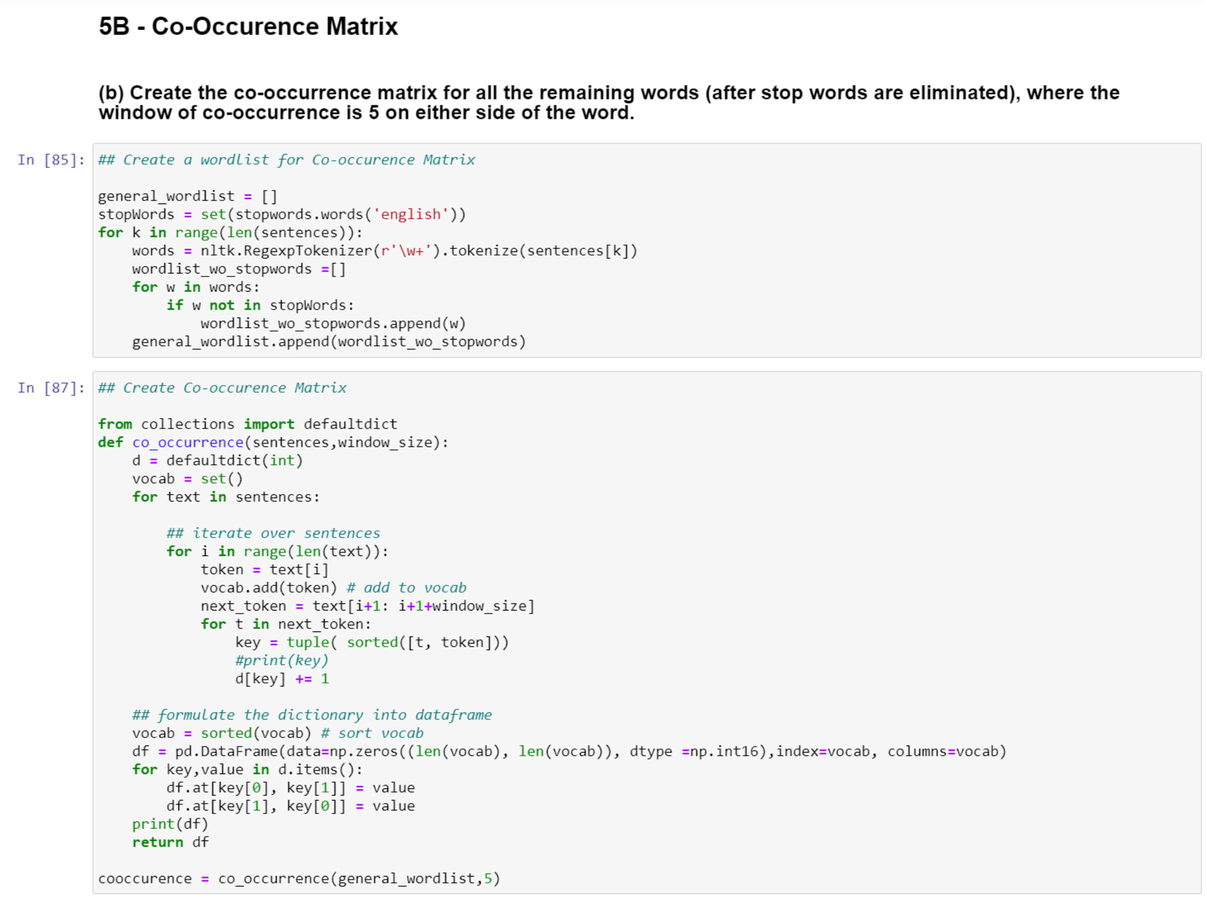
```
 Words: Lemmatized Word in sentence 1 are the following:
[('With', 'With'), ('all', 'all'), ('this', 'this'), ('stuff', 'stuff'), ('going', 'go'), ('down', 'down'), ('at', 'at'), ('t
he', 'the'), ('moment', 'moment'), ('with', 'with'), ('MJ', 'MJ'), ('i', 'i'), ('ve', 've'), ('started', 'start'), ('listenin
g', 'listen'), ('to', 'to'), ('his', 'his'), ('music', 'music'), ('watching', 'watch'), ('the', 'the'), ('odd', 'odd'), ('doc
umentary', 'documentary'), ('here', 'here'), ('and', 'and'), ('there', 'there'), ('watched', 'watch'), ('The', 'The'), ('Wi
z', 'Wiz'), ('and', 'and'), ('watched', 'watch'), ('Moonwalker', 'Moonwalker'), ('again', 'again')]
 Words: Lemmatized Word in sentence 2 are the following:
[('Maybe', 'Maybe'), ('i', 'i'), ('just', 'just'), ('want', 'want'), ('to', 'to'), ('get', 'get'), ('a', 'a'), ('certain', 'c
ertain'), ('insight', 'insight'), ('into', 'into'), ('this', 'this'), ('guy', 'guy'), ('who', 'who'), ('i', 'i'), ('thought',
```



```
## Step 5: Identifying the distinct stop words
stopWords = set(stopwords.words ('english'))

for k in range(len(sentences)):
    words = nltk.RegexpTokenizer(r'\w+').tokenize(sentences[k])
    wordlist_wo_stopwords = []
    print("Words in sentence "+ repr(k+1)+" without stop words are : ")
    for w in words:
        if w not in stopWords:
            wordlist_wo_stopwords.append(w)
    print(wordlist_wo_stopwords)
```

```
Words in sentence 1 without stop words are :
['With', 'stuff', 'going', 'moment', 'MJ', 'started', 'listening', 'music', 'watching', 'odd', 'documentary', 'watched', 'Th
e', 'Wiz', 'watched', 'Moonwalker']
Words in sentence 2 without stop words are :
['Maybe', 'want', 'get', 'certain', 'insight', 'guy', 'thought', 'really', 'cool', 'eighties', 'maybe', 'make', 'mind', 'whet
her', 'guilty', 'innocent']
Words in sentence 3 without stop words are :
['Moonwalker', 'part', 'biography', 'part', 'feature', 'film', 'remember', 'going', 'see', 'cinema', 'originally', 'release
d']
Words in sentence 4 without stop words are :
['Some', 'subtle', 'messages', 'MJ', 'feeling', 'towards', 'press', 'also', 'obvious', 'message', 'drugs', 'bad', 'kay', 'b
r', 'br', 'Visually', 'impressive', 'course', 'Michael', 'Jackson', 'unless', 'remotely', 'like', 'MJ', 'anyway', 'going', 'h
ate', 'find', 'boring']
```

(b) Create the co-occurrence matrix for all the remaining words (after stop words are eliminated), where the window of co-occurrence is 5 on either side of the word.



5B - Co-Occurence Matrix

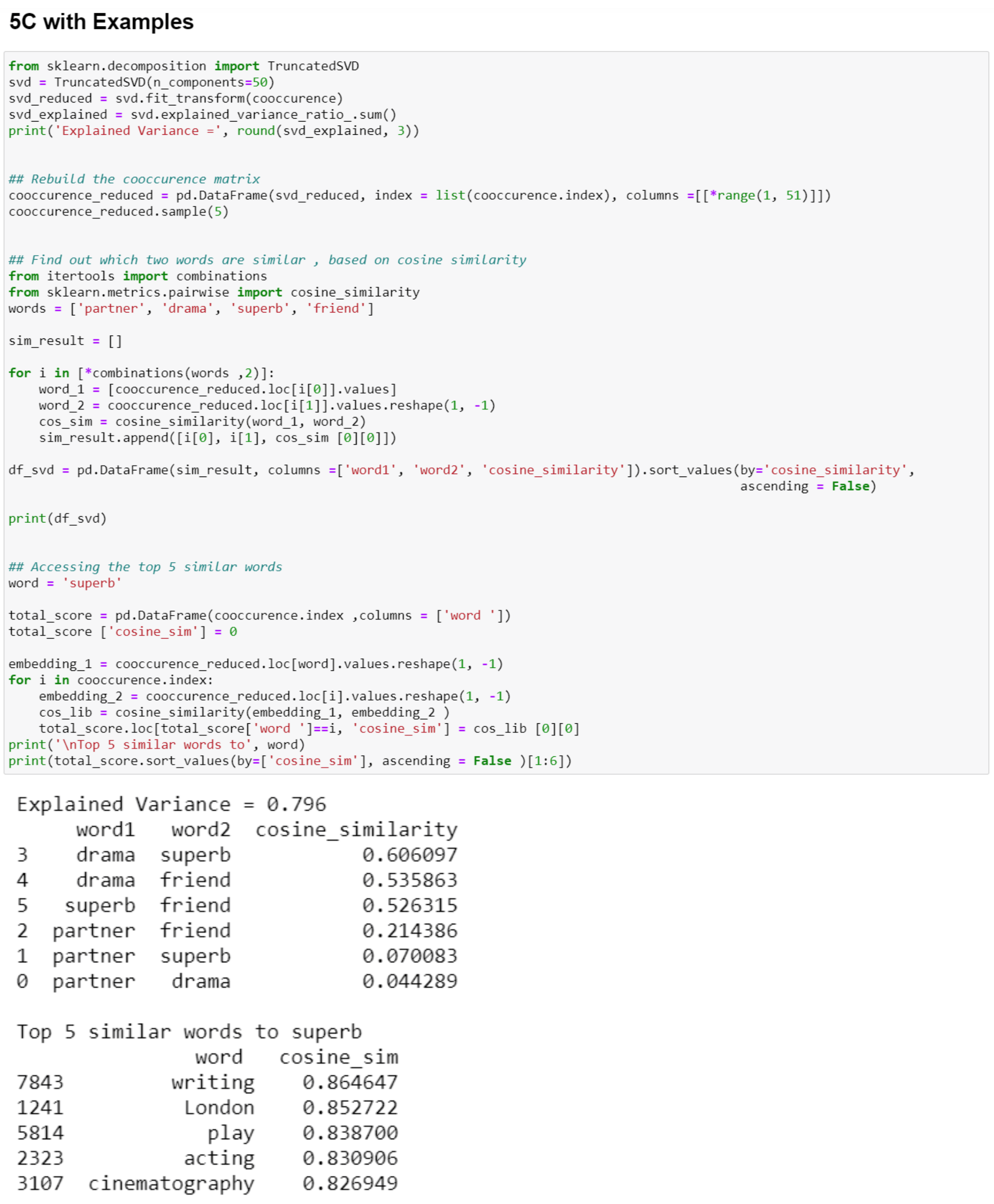
(b) Create the co-occurrence matrix for all the remaining words (after stop words are eliminated), where the window of co-occurrence is 5 on either side of the word.

```
In [85]: ## Create a wordlist for Co-occurence Matrix

         general_wordlist = []
         stopWords = set(stopwords.words('english'))
         for k in range(len(sentences)):
             words = nltk.RegexpTokenizer(r'\w+').tokenize(sentences[k])
             wordlist_wo_stopwords =[]
             for w in words:
                 if w not in stopWords:
                     wordlist_wo_stopwords.append(w)
             general_wordlist.append(wordlist_wo_stopwords)
```

```
In [87]: ## Create Co-occurence Matrix

         from collections import defaultdict
         def co_occurrence(sentences,window_size):
             d = defaultdict(int)
             vocab = set()
             for text in sentences:

                 ## iterate over sentences
                 for i in range(len(text)):
                     token = text[i]
                     vocab.add(token) # add to vocab
                     next_token = text[i+1: i+1+window_size]
                     for t in next_token:
                         key = tuple( sorted([t, token]))
                         #print(key)
                         d[key] += 1

             ## formulate the dictionary into dataframe
             vocab = sorted(vocab) # sort vocab
             df = pd.DataFrame(data=np.zeros((len(vocab), len(vocab)), dtype =np.int16),index=vocab, columns=vocab)
             for key,value in d.items():
                 df.at[key[0], key[1]] = value
                 df.at[key[1], key[0]] = value
             print(df)
             return df

         cooccurence = co_occurrence(general_wordlist,5)
         0  000  1  10  100  101  11  117  12  13th  ...  z  zapped  zero  \
0        0    0  0   2    0    0   0    0   0     0  ...  0       0     0   
000      0    0  0   1    0    0   1    0   0     0  ...  0       0     0   
1        0    0  0   2    0    0   1    0   0     0  ...  0       0     0   
10       2    1  2   3    0    0   0    0   0     0  ...  0       0     0   
100      0    0  0   0    0    0   0    0   0     0  ...  0       0     0   
...     ..  ... ..  ..  ...  ...  ..  ...  ..   ...  ... ..     ...   ...   
zombies  0    0  0   0    0    0   0    0   0     0  ...  0       0     0   
zone     0    0  0   0    0    0   0    0   0     0  ...  0       0     0   
zoom     0    0  0   0    0    0   0    0   0     0  ...  0       0     0   
zooms    0    0  0   0    0    0   0    0   0     0  ...  0       0     0   
Êxtase   0    0  0   0    0    0   0    0   0     0  ...  0       0     0   

         zip  zombie  zombies  zone  zoom  zooms  Êxtase  
0          0       0        0     0     0      0       0  
000        0       0        0     0     0      0       0  
1          0       3        0     0     0      0       0  
10         0       0        0     0     0      0       0  
100        0       0        0     0     0      0       0  
...      ...     ...      ...   ...   ...    ...     ...  
zombies    0       1        0     0     0      0       0  
zone       0       0        0     0     0      0       0  
zoom       0       0        0     0     0      0       0  
zooms      0       0        0     0     0      0       0  
Êxtase     0       0        0     0     0      0       0  

[7880 rows x 7880 columns]
```

(c) Apply SVD and obtain word embeddings of size 50.

Then, please generate word embeddings of size 50 using Word2Vec.pynb (uploaded in Lectures 5-6) on the same “200Reviews.csv” dataset. Please show comparison on few examples to understand which method works better.



5C with Examples

```python
from sklearn.decomposition import TruncatedSVD
svd = TruncatedSVD(n_components=50)
svd_reduced = svd.fit_transform(cooccurence)
svd_explained = svd.explained_variance_ratio_.sum()
print('Explained Variance =', round(svd_explained, 3))


## Rebuild the cooccurence matrix
cooccurence_reduced = pd.DataFrame(svd_reduced, index = list(cooccurence.index), columns =[[*range(1, 51)]])
cooccurence_reduced.sample(5)


## Find out which two words are similar , based on cosine similarity
from itertools import combinations
from sklearn.metrics.pairwise import cosine_similarity
words = ['partner', 'drama', 'superb', 'friend']

sim_result = []

for i in [*combinations(words ,2)]:
    word_1 = [cooccurence_reduced.loc[i[0]].values]
    word_2 = cooccurence_reduced.loc[i[1]].values.reshape(1, -1)
    cos_sim = cosine_similarity(word_1, word_2)
    sim_result.append([i[0], i[1], cos_sim [0][0]])

df_svd = pd.DataFrame(sim_result, columns =['word1', 'word2', 'cosine_similarity']).sort_values(by='cosine_similarity',
                                                                                                ascending = False)

print(df_svd)


## Accessing the top 5 similar words
word = 'superb'

total_score = pd.DataFrame(cooccurence.index ,columns = ['word '])
total_score ['cosine_sim'] = 0

embedding_1 = cooccurence_reduced.loc[word].values.reshape(1, -1)
for i in cooccurence.index:
    embedding_2 = cooccurence_reduced.loc[i].values.reshape(1, -1)
    cos_lib = cosine_similarity(embedding_1, embedding_2 )
    total_score.loc[total_score['word ']==i, 'cosine_sim'] = cos_lib [0][0]
print('\nTop 5 similar words to', word)
print(total_score.sort_values(by=['cosine_sim'], ascending = False )[1:6])
```

```
Explained Variance = 0.796
     word1   word2  cosine_similarity
3    drama  superb           0.606097
4    drama  friend           0.535863
5   superb  friend           0.526315
2  partner  friend           0.214386
1  partner  superb           0.070083
0  partner   drama           0.044289

Top 5 similar words to superb
                word  cosine_sim
7843         writing    0.864647
1241          London    0.852722
5814            play    0.838700
2323          acting    0.830906
3107  cinematography    0.826949
```