Problem 1: Value Iteration

a. answer

for value iteration

$$V_{k+1} = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$$

for 0 iterations

$$V_{opt}(-2) = 0,\ V_{opt}(-1) = 0,\ V_{opt}(0) = 0, V_{opt}(1) = 0, V_{opt}(2) = 0$$

for 1 iterations

$$V_{opt}(-2) = 0$$

$$\begin{gathered} V_{opt}(-1) = \max\{T(-1,-1,-2)[R[-1,-1,-2] + V_0(-2)] + \\ T(-1,-1,0)[R[-1,-1,0] + V_0(0)], \\ T(-1,1,0)[R[-1,1,0] + V_0(0)] + \\ T(-1,1,-2)[R[-1,1,-2] + V_0(-2)]\} \end{gathered}$$

so

$$V_{opt}(-1) = \max\begin{Bmatrix} 0.8 * 20 + 0.2 * (-5), \\ 0.3 * (-5) + 0.7 * 20 \end{Bmatrix} = 0.8 * 20 + 0.2 * (-5) = 15$$

$$\begin{gathered} V_{opt}(0) = \max\{T(0,-1,-1)[R[0,-1,-1] + V_0(-1)] + \\ T(0,-1,1)[R[0,-1,1] + V_0(1)], \\ T(0,1,1)[R[0,1,1] + V_0(1)] + \\ T(0,1,-1)[R[0,1,-1] + V_0(-1)]\} \end{gathered}$$

so

$$V_{opt}(0) = \max\begin{Bmatrix} 0.8 * (-5) + 0.2 * (-5), \\ 0.3 * (-5) + 0.7 * (-5) \end{Bmatrix} = -5$$

$$\begin{gathered} V_{opt}(1) = \max\{T(1,-1,0)[R[1,-1,0] + V_0(0)] + \\ T(1,-1,2)[R[1,-1,2] + V_0(2)], \\ T(1,1,2)[R[1,1,2] + V_0(2)] + \\ T(1,1,0)[R[1,1,0] + V_0(0)]\} \end{gathered}$$

so

$$V_{opt}(1) = \max\begin{Bmatrix} 0.8 * (-5) + 0.2 * 100, \\ 0.3 * 100 + 0.7 * (-5) \end{Bmatrix} = 0.3 * 100 + 0.7 * (-5) = 26.5$$

$$V_{opt}(2) = 0$$

Therefore

$$V_{opt}(-2) = 0,\ V_{opt}(-1) = 15,\ V_{opt}(0) = -5, V_{opt}(1) = 26.5, V_{opt}(2) = 0$$

for 2 iterations

$$V_{opt}(-2) = 0$$

$$V_{opt}(-1) = \max\{T(-1,-1,-2)[R[-1,-1,-2] + V_1(-2)] + T(-1,-1,0)[R[-1,-1,0] + V_1(0)], T(-1,1,0)[R[-1,1,0] + V_1(0)] + T(-1,1,-2)[R[-1,1,-2] + V_1(-2)]\}$$

so

$$V_{opt}(-1) = \max\begin{Bmatrix} 0.8 * 20 + 0.2 * [(-5) + (-5)], \\ 0.3 * [(-5) + (-5)] + 0.7 * 20 \end{Bmatrix} = 0.8 * 20 + 0.2 * [(-5) + (-5)] = 14$$

$$V_{opt}(0) = \max\{T(0,-1,-1)[R[0,-1,-1] + V_1(-1)] + T(0,-1,1)[R[0,-1,1] + V_1(1)], T(0,1,1)[R[0,1,1] + V_1(1)] + T(0,1,-1)[R[0,1,-1] + V_1(-1)]\}$$

so

$$V_{opt}(0) = \max\begin{Bmatrix} 0.8 * [(-5) + 15] + 0.2 * [(-5) + 26.5], \\ 0.3 * [(-5) + 26.5] + 0.7 * [(-5) + 15] \end{Bmatrix} = 0.3 * [(-5) + 26.5] + 0.7 * [(-5) + 15] = 13.45$$

$$V_{opt}(1) = \max\{T(1,-1,0)[R[1,-1,0] + V_1(0)] + T(1,-1,2)[R[1,-1,2] + V_1(2)], T(1,1,2)[R[1,1,2] + V_1(2)] + T(1,1,0)[R[1,1,0] + V_1(0)]\}$$

so

$$V_{opt}(1) = \max\begin{Bmatrix} 0.8 * [(-5) + (-5)] + 0.2 * 100, \\ 0.3 * 100 + 0.7 * [(-5) + (-5)] \end{Bmatrix} = 0.3 * 100 + 0.7 * [(-5) + (-5)] = 23$$

$$V_{opt}(2) = 0$$

Therefore

$$V_{opt}(-2) = 0,\ V_{opt}(-1) = 14,\ V_{opt}(0) = 13.45, V_{opt}(1) = 23, V_{opt}(2) = 0$$

b. policy extraction

$$\pi^*(s) = \ argmax_a \sum_{s'} P(s'|s,a)[V^*(s') + R(s,a,s')]$$

after the second iteration

$$V_{opt}(-2) = 0,\ V_{opt}(-1) = 14,\ V_{opt}(0) = 13.45, V_{opt}(1) = 23, V_{opt}(2) = 0$$

Therefore

$$\pi_{opt}(-1) = argmax_a \begin{Bmatrix} 0.8 * (0 + 20) + 0.2 * (13.45 - 5), \\ 0.3 * (13.45 - 5) + 0.7 * (0 + 20) \end{Bmatrix} = -1$$

$$\pi_{opt}(0) = argmax_a \begin{Bmatrix} 0.8 * (14 - 5) + 0.2 * (23 - 5), \\ 0.3 * (23 - 5) + 0.7 * (14 - 5) \end{Bmatrix} = 1$$

$$\pi_{opt}(1) = argmax_a \begin{Bmatrix} 0.8 * (13.45 - 5) + 0.2 * (0 + 100), \\ 0.3 * (0 + 100) + 0.7 * (13.45 - 5) \end{Bmatrix} = 1$$

Therefore

$$\pi_{opt}(-1) = -1, \qquad \pi_{opt}(0) = 1, \qquad \pi_{opt}(1) = 1$$

Problem 2: Transforming MDPs

a. answer

I think it is not always the case that $V_1(s_{start}) \geq V_2(s_{start})$. The counterexample is that,

Assume that, for the original MDP, the current state of agent is 0. If the agent goes to 1 with the probability 0.9, the reward the agent gets is 1, while if the agent goes to -1 with the probability 0.1, the reward the agent gets is 10. For us, we absolutely go to 1 to get 10 reward, but for the agent, the transition prob is 0.1 for -1 and 0.9 for 1, which means it often goes to 1 even if it wants to get to -1.

However, when the noise added, the probability for going to -1 rises. In the case, the agent can sometimes get to -1 with more than 0.5 prob.

The counterexample code is in *submission.py*

b. answer

For value iteration,

$$V_{k+1} = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$$

The reason why the normal MDP needs to visit the specific state many times is that the agent doesn't have a specific order over states.

If we have an acyclic MDP, which means the agent will not reach one state two times, we can get the specific order over states for agent. In this case, when we calculate V_{k+1}, we just utilize the order to compare its successor node rather than explore the whole search space.

c. answer

$$T'(s,a,s') = \gamma T(s,a,s')$$

$$T'(s,a,o) = 1 - \sum_{s'} T'(s,a,s') = 1 - \gamma \sum_{s'} T(s,a,s') = 1 - \gamma$$

$$R'(s,a,s') = \frac{R(s,a,s')}{\gamma}$$

$$R'(s,a,o) = 0$$

Problem 3: Peeking Blackjack

a. answer

We need to consider three conditions, take, peek and quit. The succAndProbReward code is in *submission.py*

b. answer

In the case of BlackjackMDP (cardValues = [1,2,3,4,50], multiplicity = 1, threshold = 20, peekCost = 1), the agent needs to peek everytime to avoid taking 50. The peekingMDP code is in *submission.py*

Problem 4: Learning to Play Blackjack

a. answer

Off policy strategy,

$$Q(s,a) = Q(s,a) + \alpha(\ R(s)\gamma \max_{a} Q(s',a') - Q\ (s,a)\ )$$

The incorporateFeedback code is in *submission.py*

b. answer

The trail code is as follows,

```
def simulate_QL_over_MDP(MDP, featureExtractor):
  # NOTE: adding more code to this function is totally optional, but it will probably be useful
  # to you as you work to answer question 4b (a written question on this assignment).  We suggest
  # that you add a few lines of code here to run value iteration, simulate Q-learning on the MDP,
  # and then print some stats comparing the policies learned by these two approaches.
  # BEGIN_YOUR_CODE
  # pass
  RL = QLearningAlgorithm(MDP.actions, MDP.discount, featureExtractor, explorationProb=0)
  util.simulate(MDP, RL, numTrials=30000, maxIterations=1000, verbose=False, sort=False)
  MDP.computeStates()
  RL_policy = {}
  for state in MDP.states:
    RL_policy[state] = RL.getAction(state)
  val = util.ValueIteration()
  val.solve(MDP)
  val_policy = val.pi
  sum_ = []
  for key in RL_policy:
    if RL_policy[key] == val_policy[key]:
      sum_.append(1)
    else:
            sum_.append(0)
    print(float(sum(sum_))/len(RL_policy))
    return RL_policy, val_policy

smallMDP_RL_policy, smallMDP_val_policy = simulate_QL_over_MDP(smallMDP, identityFeatureExtractor)
largeMDP_RL_policy, largeMDP_val_policy = simulate_QL_over_MDP(largeMDP, identityFeatureExtractor)
```

Question 1: How does the Q-learning policy compare with a policy learned by value iteration (i.e., for how many states do they produce a different action)?

	Q-learning policy	value iteration
state: action	(0, None, None): 'Take',	(0, None, None): 'Take',
	(1, 1, (1, 2)): 'Take',	(1, 1, (1, 2)): 'Take',
	(0, None, (2, 2)): 'Take',	(0, None, (2, 2)): 'Take',
	(7, None, (0, 1)): 'Take',	(7, None, (0, 1)): 'Quit',
	(1, None, (1, 2)): 'Take',	(1, None, (1, 2)): 'Take',
	(6, None, (1, 1)): 'Take',	(6, None, (1, 1)): 'Quit',
	(6, 1, (1, 1)): 'Take',	(6, 1, (1, 1)): 'Quit',
	(7, None, None): 'Take',	(7, None, None): 'Take',
	(6, 0, (1, 1)): 'Take',	(6, 0, (1, 1)): 'Quit',
	(2, None, (0, 2)): 'Take',	(2, None, (0, 2)): 'Take',
	(2, 1, (0, 2)): 'Take',	(2, 1, (0, 2)): 'Take',
	(10, None, (2, 0)): 'Take',	(10, None, (2, 0)): 'Quit',
	(11, None, None): 'Take',	(11, None, None): 'Take',
	(12, None, None): 'Take',	(12, None, None): 'Take',
	(5, None, None): 'Take',	(5, None, None): 'Take',
	(0, 0, (2, 2)): 'Take',	(0, 0, (2, 2)): 'Take',
	(10, 0, (2, 0)): 'Take',	(10, 0, (2, 0)): 'Quit',
	(6, None, None): 'Take',	(6, None, None): 'Take',
	(10, None, None): 'Take',	(10, None, None): 'Take',
	(1, None, None): 'Take',	(1, None, None): 'Take',
	(5, 1, (2, 1)): 'Take',	(5, 1, (2, 1)): 'Take',
	(5, None, (2, 1)): 'Take',	(5, None, (2, 1)): 'Take',
	(1, 0, (1, 2)): 'Take',	(1, 0, (1, 2)): 'Take',
	(5, 0, (2, 1)): 'Take',	(5, 0, (2, 1)): 'Take',
	(2, None, None): 'Take',	(2, None, None): 'Take',
	(0, 1, (2, 2)): 'Take',	(0, 1, (2, 2)): 'Take',
	(7, 1, (0, 1)): 'Take'	(7, 1, (0, 1)): 'Quit'

From the table, we can see that there are 7 states that they produce a different action.

Question 2: How does the policy learned in this case (largeMDP) compare to the policy learned by value iteration?

	Value Iteration	policy overlap	compare
smallMDP	5	0.741	For small MDP, the search space is small, so Q_{opt} can be obtained accurately. In this case, Q-learning is better than Value Iteration
largeMDP	15	0.668	For large MDP, the search space is big, so Q_{opt} cannot be obtained accurately. In this case, Q-learning is worse than Value Iteration

Question 3: What went wrong?

For large MDP, the search space is big, so Q_{opt} cannot be obtained accurately. The reason is that the partial updating doesn't enforce consistency among all the value of Q in this RL-model.

c. answer

The blackjackFeatureExtractor code is in *submission.py*

d. answer

```
# Problem 4d: What happens when the MDP changes underneath you?!

# Original mdp
originalMDP = BlackjackMDP(cardValues=[1, 5], multiplicity=2, threshold=10, peekCost=1)

# New threshold
newThresholdMDP = BlackjackMDP(cardValues=[1, 5], multiplicity=2, threshold=15, peekCost=1)

def compare_changed_MDP(original_mdp, modified_mdp, featureExtractor):
    # NOTE: as in 4b above, adding more code to this function is completely optional, but we've adde
d
    # this partial function here to help you figure out the answer to 4d (a written question).
    # Consider adding some code here to simulate two different policies over the modified MDP
    # and compare the rewards generated by each.
    # BEGIN_YOUR_CODE
    val = util.ValueIteration()
    val.solve(original_mdp)
    val_policy = val.pi
    RL1 = util.FixedRLAlgorithm(val_policy)
    sum1 = sum(util.simulate(modified_mdp, RL1, numTrials=30000, maxIterations=1000, verbose=
False, sort=False))
    print(sum1)
    RL2 = QLearningAlgorithm(modified_mdp.actions, modified_mdp.discount, featureExtractor, explorationProb=0)
    sum2 = sum(util.simulate(modified_mdp, RL2, numTrials=30000, maxIterations=1000, verbose=False, sort=False))
    print(sum2)
    # pass
    # END_YOUR_CODE

compare_changed_MDP(originalMDP, newThresholdMDP, blackjackFeatureExtractor)
```

Question 1: simulate your policy on newThresholdMDP (also defined for you in submission.py) by calling simulate with an instance of FixedRLAlgorithm that has been instantiated using the policy you computed with value iteration. What is the expected reward from this simulation?

the expected reward from this simulation is 6.83668

Question 2: Now try simulating Q-learning directly on newThresholdMDP with blackjackFeatureExtractor and the default exploration probability. What is your expected reward under the new Q-learning policy?

the expected reward from this simulation is 9.60092

Question 3: Provide some explanation for how the rewards compare, and why they are different.

	expected reward
Fixed-RL-Algorithm	6.83668
Q-learning	9.60092

newThresholdMDP uses the policy of originalMDP to carry out simulation, but newThresholdMDP and originalMDP are different MDP, so originalMDP's policy is not the optimal policy for newThresholdMDP probably, but the Q-learning method is to obtain the optimal policy of newThresholdMDP through reinforcement learning, so the expected reward of Q-learning method is larger than the expected reward of Fixed RL Algorithm method.