Beating Blackjack - A Reinforcement Learning Approach

Joshua Geiser and Tristan Hasseler
Stanford University

As a popular casino card game, many have studied Blackjack closely in order to devise strategies for improving their likelihood of winning. This research seeks to develop various learning algorithms for Blackjack play and to verify common strategic approaches to the game. Each of the three learning algorithms presented below, Value Iteration, Sarsa, and Q-Learning, resulted in policies with win rates far superior than a random policy when taking into account the agent's hand. By additionally accounting for the dealer's up-card, the agent was able to further improve win percentage. These results were then directly compared with the frequently used "ideal" Blackjack strategy charts and are shown to be nearly identical, thus verifying that the agent was able to properly learn an optimized policy for Blackjack play.

I. Introduction

Blackjack is a popular card game that is frequently played in casinos around the world. A typical game of blackjack consists of one or more players who play against the dealer (sometimes referred to as "the house" in casino play). The goal of the game is to have a hand of cards that collectively sum to 21. During each turn a player can elect to receive another card (known as a "hit") or to stop receiving additional cards (known as a "stand"). If the player's hand goes over 21, this is referred to as a "bust" and the player loses the hand. Assuming the player opts to "stand" before busting, the dealer's hand is dealt out until their hand sums to at least 17 or they bust. If neither the player nor dealer busts, the winner is chosen by whose hand is closer to 21 (tie goes to the dealer). An Ace can count as a 1 or an 11 towards a player's hand, all face cards represent a value of 10, and all other cards represent a value equal to their numeric value. In addition to hitting/standing actions, a player typically also has options to "split" hands (if their hand contains a pair), "double-down" bets on high-potential hands, or "surrender" on low-potential hands. On the other hand, the dealer follows a fixed policy of hitting only until their hand sums to 17 or greater (or they bust), and standing thereafter.

Card and board games have long been used to train and evaluate various learning algorithms [1],[2]. Blackjack is one such example of this, and has been frequently studied due to its interesting relationship with probability theory. Due to its relatively small state and action spaces as well as its well defined rules and rewards systems, Blackjack is often thought of as a test-bed for modern reinforcement learning, deep learning, and neural network algorithms [3]. Additionally, some "Basic" strategy tables based on the dealer's up-card and player's hand have been built for specific Blackjack variations [4]. These tables (one example of which is shown in **Figure 2)** are frequently used by Blackjack enthusiasts to increase their expected win rate. It is then easy to evaluate a blackjack strategy produced by a new algorithm by comparing it to these "Basic" strategies along with traditional expert strategy analysis [5]. The overarching goal of this project was to verify some of this existing research on optimal policies for Blackjack play.

II. Theory

Presented in this section is a discussion of the various reinforcement learning algorithms [6] used in this project.

A. Model-based Methods

Model-based methods can be used to learn the underlying reward and transition models from an agent's interaction with the environment. Taking a maximum-likelihood approach, we have that

$$T(s'|s,a) \approx \frac{N(s,a,s')}{N(s,a)}, \tag{1}$$

$$R(s,a) \approx \frac{\rho(s,a)}{N(s,a)} \tag{2}$$

where $N(s,a,s')$ is a matrix that counts the number of times a transition from $s \rightarrow s'$ occurred from taking action a. Similarly, $N(s,a)$ is simply the total number of times state s and action a have been observed: $N(s,a) = \sum_{s'} N(s,a,s')$.

To avoid division by zero, if $N(s,a) = 0$ we set $R(s,a) = T(s'|s,a) = 0$. Additionally, $\rho(s,a)$ is the running total sum of rewards received when taking action a in state s.

Once R and T have been estimated, we can use any number of solution methods to solve the problem. For this project, Value Iteration was used to solve the model-based problem and generate an optimal policy:

$$U_{k+1}(s) = max_a \left(R(s,a) + \gamma \sum_{s'} T(s'|s,a) U_k(s') \right) \tag{3}$$

The Bellman equation above is iterated until convergence. Once the converged value function $U^*(s)$ has been computed, the policy can simply be extracted by taking the greedy policy:

$$\pi^*(s) = argmax_a(U^*(s)) \tag{4}$$

B. Model-free Methods

Another valid approach is to estimate the Q function directly as opposed to learning R and T. A simple example of such a method is Q-learning, which leverages the incremental update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma max_{a'} \left[Q(s',a') \right] - Q(s,a) \right) \tag{5}$$

Here, the learning rate, α, is a hyper-parameter that must be chosen based on the problem. Another closely related model-free method is Sarsa, which involves the same incremental Q-update as Q-learing, but with the actual next state and action as opposed to maximization over all possible next actions:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma Q(s',a') - Q(s,a) \right) \tag{6}$$

The policy is then extracted as the greedy policy which maximizes Q(s,a) for a given state.

III. Modeling

A few assumptions were made in order to simplify the state and action models. First, the agent is assumed to have an action space consisting of only two actions: hit (draw another card) or stand (pass on drawing any more cards). Although casino blackjack typically includes options to double down, split hands, or surrender, reducing the action space to these two options simplified overall model complexity.

Additionally, two different state parameterizations were modeled to evaluate impacts of model complexity on performance. Both state parameterizations included a win state (agent won the current round), lose state (dealer won the current round), and terminal state (end of current round).

State parameterization 1 included states based solely on the score of the agent's current hand. This served as a baseline model for evaluating how an agent would perform against the dealer without taking into account the dealer's up-card. This parameterization consisted of 21 total states (18 states for hand values ranging from 4 to 21 and the 3 additional win/lose/terminal states).

State parameterization 2 included both knowledge of the agent's hand and the dealer's up-card in the state mapping. Since the agent's hand could range between 18 values and the dealer's up-card could range between 10 different numerical values, this state parameterization included 183 total states (again including the win/lose/terminal states). **Figure 1** provides a visual depiction of these state mapping methodologies.

For both state parameterizations, if the agent acted to "hit", it would be transported to its next state based on the agent's new hand total and the dealer's up-card. If the new card caused the agent to "bust" (score > 21), the agent would be transported directly to the lose state. If at any time the agent elects to "stand", the dealer's hand is played out and the agent is transported to either the win state or the lose state depending on the outcome. From the win and lose states, the agent always travels directly to the terminal state.

The reward model was fairly simple for state parameterization 1. The agent would receive a large positive reward (+10) in the win state, and alternatively a large negative reward (-10) in the lose state. The agent would receive 0 reward in all other states. State parameterization 2 included the same rewards for the win/lose states, but additionally included a small positive reward (+1) when the agent "hit" on low hand values (score $\leq$ 11) or "stood" on large hand values (score $\geq$ 17). Alternatively, a small negative reward (-1) was given when the opposite actions were chosen. This modeling decision was introduced due to observations of the agent's trained policy in state parameterization 1 (as discussed in the Results & Discussion Section) as well as the low frequencies with which some states were visited.

State Parameterization 1		
		Dealer Upcard
		n/a
Player's Hand	4	3
	5	4
	6	5
	7	6
	8	7
	9	8
	10	9
	11	10
	12	11
	13	12
	14	13
	15	14
	16	15
	17	16
	18	17
	19	18
	20	19
	21	20

State Parameterization 2										
		Dealer Upcard								
		A	2	3	4	5	6	7	8	9	10
Player's Hand	4	3	21	39	57	75	93	111	129	147	165
	5	4	22	40	58	76	94	112	130	148	166
	6	5	23	41	59	77	95	113	131	149	167
	7	6	24	42	60	78	96	114	132	150	168
	8	7	25	43	61	79	97	115	133	151	169
	9	8	26	44	62	80	98	116	134	152	170
	10	9	27	45	63	81	99	117	135	153	171
	11	10	28	46	64	82	100	118	136	154	172
	12	11	29	47	65	83	101	119	137	155	173
	13	12	30	48	66	84	102	120	138	156	174
	14	13	31	49	67	85	103	121	139	157	175
	15	14	32	50	68	86	104	122	140	158	176
	16	15	33	51	69	87	105	123	141	159	177
	17	16	34	52	70	88	106	124	142	160	178
	18	17	35	53	71	89	107	125	143	161	179
	19	18	36	54	72	90	108	126	144	162	180
	20	19	37	55	73	91	109	127	145	163	181
	21	20	38	56	74	92	110	128	146	164	182

Fig. 1 State Parameterization 1 (left) and State Parameterization 2 (right)

IV. Results & Discussion

The following results were obtained by training various algorithms on simulation data from many successive games of an agent following a random policy. This exploration phase consisted of 10,000 runs for training models using state parameterization 1, and 100,000 runs for training with state parameterization 2. The random policy had an average win rate of 28%.

We note that because the algorithms were trained on previously obtained simulation data, the reinforcement learning algorithms presented here were a subset of batch reinforcement learning. Due to the small action space, it was observed that following a purely stochastic exploration policy (i.e. randomly hit or stand on a given turn) during the batch generation phase still provided a sufficiently representative training dataset where most state-action pairs were visited.

A. State Parameterization 1

1. Value Iteration

Using first a model-based approach to reinforcement learning, the reward and transition models, $R(s,a)$ and $T(s'|s,a)$ were estimated using **Eqs. 1-2**. It is interesting to note that while no reward or transition model knowledge was assumed, either one could have been directly entered by hand. The reward model was devised exactly by the framing of the problem (receive a +10 reward for being in the win state and a -10 reward for being in the lose state). Further, the transition probability from one score to another could have easily been coded using simple probability of a random card draw without replacement. However, we were drawn to the generality of reinforcement learning to estimate R and T directly from the training dataset. We also wished to see how accurately a maximum likelihood approach would estimate the already-known transition and rewards models. Using this methodology, it was found that the maximum likelihood approach exactly recovered the reward model and accurately estimated the transition model after 10,000 datapoints were collected.

Once the transition and reward models had been estimated, the Value Iteration algorithm was used to iterate on a value function for each state until convergence using **Eq. 3**. Once the value function converged, the policy for a given state was selected as the greedy action using **Eq. 4**. A discount factor of $\gamma = 0.5$ was selected based on the short (usually 1-4 turns) horizon needed for Blackjack. The result was a policy that chose to hit if the agent's score was ≤ 14 and to stand if the agent's score was ≥ 15. This fixed policy resulted in an average win rate of 41.2% after playing 20,000 games against the dealer - a noted increase over the 28% win rate of a purely random policy.

2. Sarsa

In addition to model-based value iteration, we also implemented two separate model-free reinforcement learning algorithms, the first of which was Sarsa. The incremental update in the Sarsa algorithm was implemented using **Eq. 6**. This equation was iterated 5-10 times to ensure convergence of the resulting Q-function. Here, the next state, s' was given as part of the batch learning tuples generated beforehand. The next action a' was taken as the action performed in

the next observed turn during a game. Because the exploration strategy employed in the batch generation phase was observed to effectively fill the state-action space, it was reasoned that the next action a' would serve as a good proxy for the maximization over all actions that is required for Q-learning. Similar to the above method, a discount factor of $\gamma = 0.5$ was selected. Further, a learning rate of $\alpha = 0.01$ was found to provide good results after iterating through a number of different values. The resulting policy using Sarsa was slightly more conservative than the above method, instead deciding to hit if the agent's score was ≤ 13 and to stand if the agent's score was ≥ 14. This more conservative policy showed a slightly improved average win rate of 41.9% after playing 20,000 games against the dealer. This win rate is still comparable to the previous method, however, and shows that the two fundamentally different approaches both provide reasonable policies that improve over the baseline.

3. Q-Learning

Q-Learning was another model-free reinforcement learning algorithm that was applied to train an agent. Various hyperparameter values were tested, however the final trained model used a discount factor of $\gamma = 0.5$ and learning rate $\alpha = 0.01$. This led to an optimized policy where the agent would elect to hit if its current hand value was ≤ 15, or to stand for hand values ≥ 16. Similar to Value Iteration and Sarsa, this policy led to an average win percentage of approximately 41.4%. This further verified that all three algorithms successfully improved the performance of the agent by about 13.5% when compared with the random policy.

B. State Parameterization 2

All three reinforcement learning algorithms discussed above were re-run using state mapping 2, which not only tracked the agent's score but also the dealer's upcard. We hypothesized this additional information would help generate a policy with a higher win rate as opposed to solely tracking the agent's score. Thus, it is better to also reason about what your opponent's score is before selecting an action for yourself.

Running value iteration with this new state mapping (still with a discount factor of $\gamma = 0.5$) generated a policy that had a resulting average win rate of 42.4% after playing 20,000 games against the dealer - an increase of approximately 1.2% over the previous value iteration policy. Similarly, the policy generated by Sarsa (still using $\alpha = 0.01$, $\gamma = 0.5$) provided an average win rate of 42.5%, an increase of approximately 0.6% over the previous Sarsa policy. Finally, re-running Q-learning with the extra knowledge of the dealer's up-card allowed the agent to increase its win percentage to 42.5% from 41.4%.

These results support our initial hypothesis that including the dealer's upcard in the state will improve the win rate. Though these 0.6-1.2% increases from state parameterization 1 seem minor, any slight advantage in a game like blackjack is quite significant in the long term. These final win percentages seem like reasonable values, as our expected win percentage should be less than 50% due to the inherent house advantage in casino blackjack.

By taking into account both the player's summed hand as well as the dealer's up-card, the resulting policies could easily be compared to existing blackjack strategy charts. Consider, for example, the policy obtained from Q-learning in **Figure 2** below.

Figure 2 compares the Q-Learning policy with the ideal strategies for both hard and soft hand totals. As can be readily seen, the optimized policy appears to follow the ground-truth reference strategy almost identically (assuming hard hand totals). The trained agent only appears to deviate from the optimal strategy on some of the borderline cases with lower dealer hand values (depicted by bold font). However, since the assumptions in our state parameterization prevent knowledge of whether we have a hard or soft hand total, this accounts for the slight deviation in our policy from the "Hard Totals" table.

See **Table 1** below for a summary of the win rates obtained from each of the policies discussed above.

V. Conclusions

Three separate reinforcement learning algorithms were implemented to generate Hit-Stand policies for the game of Blackjack. One model-based method (maximum likelihood with Value Iteration), and two model-free methods (Sarsa and Q-learning), were implemented. For each of these algorithms, two different state parameterizations were introduced. First, a simple state mapping that only tracked the agent's hand was implemented. Using this mapping, Value Iteration, Sarsa, and Q-learning produced policies that resulted in win rates of 41.2%, 41.9%, and 41.4%, respectively after playing 20,000 games against the dealer. Second, a more complex state mapping that tracked both the agent's hand as well as the dealer's up-card was implemented. In this case, Value Iteration, Sarsa, and Q-learning produced policies that

Fig. 2 Policy Comparison between Q-Learning (left) and Reference Strategy Tables [7] (right)

Table 1 Win rate after 20,000 games for each policy

Policy	State Mapping 1 (agent's hand)	State Mapping 2 (agent's hand + dealer's upcard)
Random Policy	28%	28%
Value Iteration	41.2%	42.4%
Sarsa	41.9%	42.5%
Q-Learning	41.4%	42.5%

resulted in win rates of 42.4%, 42.5%, and 42.5%, respectively. The win rate was thus observed to improve on the order of 1-1.5% using this new state mapping.

These results support the hypothesis that including knowledge of the opponent's hand leads to higher win rates for the agent. Compared to a baseline win rate of 28% for a purely random policy, each algorithm provided a significant improvement to the agent's play. Further, it was found that each of the three algorithms converged to similar policies with very similar win rates - especially for the second state mapping. These policies were compared to the "solved" policy employed by expert Blackjack players, and it was seen that our policies were extremely close to these expert policies. Because of this, we are confident that each learning algorithm presented in this project provided an effective solution to the problem.

For future work, it would be interesting to further explore the effect of "counting cards" (i.e. reasoning about what the agent might expect to draw next based on which cards have already been played). It would also be interesting to apply more exotic extensions of reinforcement learning algorithms such as Q-λ to see if any extra win percentage can be achieved. Finally, additional future work could also extend the action space of the agent, allowing it to place bets, double-down, or perform other advanced Blackjack moves.

References

[1] Yakowitz, S., and Kollier, M., "Machine learning for optimal blackjack counting strategies," *Journal of Statistical Planning and Inference*, 1992. https://doi.org/10.1016/0378-3758(92)90001-9.

[2] Fogel, D. B., Hays, T. J., Hahn, S. L., and Quon, J., "A self-learning evolutionary chess program," *Proceedings of the IEEE*, 2004. https://doi.org/10.1109/JPROC.2004.837633.

[3] Perez-Uribe, A., and Sanchez, E., "Blackjack as a test bed for learning strategies in neural networks," *IEEE International Conference on Neural Networks - Conference Proceedings*, 1998. https://doi.org/10.1109/ijcnn.1998.687170.

[4] Bond, N., "Basic Strategy and Expectation in Casino Blackjack," *Organizational Behavior and Human Performance*, Institute of Electrical and Electronics Engineers Inc., 1974, pp. 413–428. https://doi.org/10.1016/0030-5073(74)90061-0.

[5] Baldwin, R. R., Cantey, W. E., Maisel, H., and McDermott, J. P., "The Optimum Strategy in Blackjack," *Journal of the American Statistical Association*, 1956. https://doi.org/10.1080/01621459.1956.10501334.

[6] Kochenderfer, M. J., *Decision Making Under Uncertainty: Theory and Application*, MIT Press, 2015.

[7] Jones, C., "Blackjack Basic Strategy Chart," , 2020. URL https://www.blackjackapprenticeship.com/blackjack-strategy-charts/.

VI. Appendix A - Contributions

A. Josh

I worked on the setup for simulating Blackjack games with random and fixed policies, as well as training with the Q-Learning algorithm. I wrote the Abstract, Modeling Section, and parts of the Introduction and Results & Discussion Sections.

B. Tristan

I worked on implementing the model-based algorithm (max likelihood with Value Iteration). I also implemented the Sarsa algorithm. For the paper, I wrote the Theory section, the Conclusion section, and parts of the Introduction and Results & Discussion.

VII. Appendix B - Code

A. Blackjack Code

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Oct 30 17:18:17 2020

@author: joshuageiser
"""

import os
import numpy as np
import pandas as pd
import random

class Params():
    def __init__(self):
        # 'input', 'random_policy', 'fixed_policy'
        self.action_type = 'fixed_policy'

        # Only used for 'random_policy' or 'fixed_policy' input
        self.num_games = 20000
```

```
        # Filepath to fixed policy file (only used for 'fixed_policy' input)
        self.fixed_policy_filepath = os.path.join(os.getcwd(), 'Sarsa_Policy_2.policy')

        # Which state mapping algorithm to use (1 or 2)
        self.state_mapping = 2

        return

'''
State Mapping 1: state = players_hand - 1

State 0 - lose state
State 1 - win state
State 2 - terminal state
State 3 - players hand sums to 4
State 4 - players hand sums to 5
State 5 - players hand sums to 6
State 6 - players hand sums to 7
...
State 19 - players hand sums to 20
State 20 - players hand sums to 21

-------------------------------------------------------------------------------

State Mapping 2: state = (players_hand - 1) + (18 * (dealers_hand-1))

State 0 - lose state
State 1 - win state
State 2 - terminal state
State 3 - players hand sums to 4, dealers hand is 1
State 4 - players hand sums to 5, dealers hand is 1
...
State 19 - players hand sums to 20, dealers hand is 1
State 20 - players hand sums to 21, dealers hand is 1
State 21 - players hand sums to 4, dealers hand is 2
State 22 - players hand sums to 5, dealers hand is 2
...
State 181 - players hand sums to 20, dealers hand is 10
State 182 - players hand sums to 21, dealers hand is 10

'''

class BlackJack_game():
    def __init__(self, params):

        # 1 = Ace, 2-10 = Number cards, Jack/Queen/King = 10
        self.deck = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10]*4
        random.shuffle(self.deck)

        # Player and dealer hands
        self.player = self.draw_hand()
        self.dealer = [self.draw_card()]

        # State, Action, Reward, Next State arrays
        self.sarsp = []
        self.sarsp_arr = np.array([], dtype='int').reshape(0,4)
```

```
        # Various other parameters
        self.action_type = params.action_type # 'input', 'random_policy', 'fixed_policy'
        self.verbose = (params.action_type == 'input')
        self.num_games = params.num_games
        self.fixed_policy_filepath = params.fixed_policy_filepath
        self.policy = self.load_policy()
        self.state_mapping = params.state_mapping

        # Probably do not need to change these
        self.lose_state = 0
        self.win_state = 1
        self.terminal_state = 2

        # Also do not need to change these
        self.lose_reward = -10
        self.win_reward = 10
        return

    # Reset deck, player/dealer hands, and sarsp for a new game
    def reset(self):
        self.player = self.draw_hand()
        self.dealer = [self.draw_card()]
        self.sarsp = []

        self.deck = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10]*4
        random.shuffle(self.deck)
        return

    # Draw random card from deck
    def draw_card(self):
        return self.deck.pop()

    # Dras random hand (2 random cards from deck)
    def draw_hand(self):
        return [self.draw_card(), self.draw_card()]

    # Does this hand have a usable ace?
    def usable_ace(self, hand):
        return 1 in hand and sum(hand) + 10 <= 21

    # Return current hand total
    def sum_hand(self, hand):
        if self.usable_ace(hand):
            return sum(hand) + 10
        return sum(hand)

    # Is this hand a bust?
    def is_bust(self, hand):
        return self.sum_hand(hand) > 21

    # What is the score of this hand (0 if bust)
    def score(self, hand):
        return 0 if self.is_bust(hand) else self.sum_hand(hand)

    # Return True if the player won or False if the dealer won
    def player_won(self, player, dealer):
        if self.is_bust(player):
            return False
```

```
        elif self.is_bust(dealer):
            return True
        elif self.sum_hand(player) > self.sum_hand(dealer):
            return True
        else:
            return False

    # Map the current player's hand to a state index
    def hand_to_state(self, player, dealer):
        if self.state_mapping == 1:
            return self.sum_hand(player) - 1
        elif self.state_mapping == 2:
            return (self.sum_hand(player) - 1) + (18 * (dealer[0] - 1))

    # Get reward based off of current state and action (may get rid of this
    # function, not really being used at the moment)
    def get_reward(self, state, action, player, dealer):
        if self.state_mapping == 1:
            return 0
        else:
            if ((self.sum_hand(player) <= 11 and action == 1) or
                (self.sum_hand(player) >= 17 and action == 0)):
                    return 1
            elif ((self.sum_hand(player) <= 11 and action == 0) or
                (self.sum_hand(player) >= 17 and action == 1)):
                return -1
            else:
                return 0

    # Load policy from input .policy file into self.policy
    def load_policy(self):

        # Policy not needed if a user is playing or a random policy is being used
        if self.action_type in ['random_policy', 'input']:
            return None

        # Read policy file and extract policy
        f = open(self.fixed_policy_filepath, 'r')
        data = f.read()
        data = data.split()
        policy = [int(x) for x in data]

        return policy

    # Print data about the current player's/dealer's hands
    # This only used for 'input' mode where user is playing a single blackjack game
    def print_iter(self):
        if not self.verbose:
            return

        print(f'Player hand: {self.player}\t\t sum: {self.sum_hand(self.player)}')
        print(f'Dealer hand: {self.dealer}\t\t sum: {self.sum_hand(self.dealer)}')
        return

    # Get action depending on if user is playing, or if a random/fixed policy
    # is being used
    def get_action(self, state):
        if self.action_type == 'input':
```

```
            action = int(input('Hit (1) or Pass (0): '))
        elif self.action_type == 'random_policy':
            action = np.random.randint(2)
        elif self.action_type == 'fixed_policy':
            action = self.policy[state]
        return action

    # Play a single game of BlackJack!
    def play_game(self):

        # Only for 'input' mode
        if self.verbose:
            print('New Game!\n')

        # Iterate through game
        done = False
        while(not done):

            # Only for 'input' mode
            self.print_iter()

            # Current state/action/reward
            state = self.hand_to_state(self.player, self.dealer)
            action = self.get_action(state)
            reward = self.get_reward(state, action, self.player, self.dealer)

            if action:  # hit: add a card to players hand and return
                self.player.append(self.draw_card())
                if self.is_bust(self.player):
                    done = True
                else:
                    done = False
            else:  # stick: play out the dealers hand, and score
                while self.sum_hand(self.dealer) < 17:
                    self.dealer.append(self.draw_card())
                done = True

            # Add a row to sarsp as long as we still have more iterations
            # through the while loop
            if(not done):
                sp = self.hand_to_state(self.player, self.dealer)
                self.sarsp.append([state, action, reward, sp])

        # Only for 'input' mode
        self.print_iter()

        # Check if player won
        player_won_bool = self.player_won(self.player, self.dealer)

        # Set next state to win state or lose state based on if player won/lost
        if player_won_bool:
            sp = self.win_state
        else:
            sp = self.lose_state
        self.sarsp.append([state, action, reward, sp])

        # Add a row with 0 action, win/loss reward, and terminal state for next state
        state = sp
```

```
            if player_won_bool:
                reward = self.win_reward
            else:
                reward = self.lose_reward
            self.sarsp.append([state, np.random.randint(2), reward, self.terminal_state])

            # Only for 'input' mode
            if self.verbose:
                print(f'Player won?: {player_won_bool}')

            # Append current run data to full sarsp_arr
            self.sarsp_arr = np.vstack((self.sarsp_arr, np.array(self.sarsp)))

            return

        # Output CSV file of runs if a random_policy was used
        def output_sarsp_file(self):
            filename = f'random_policy_runs_mapping_{self.state_mapping}.csv'

            output_filepath = os.path.join(os.getcwd(), filename)
            header = ['s', 'a', 'r', 'sp']
            pd.DataFrame(self.sarsp_arr).to_csv(output_filepath, header=header, index=None)
            return

        # Print win/loss stats if a random or fixed policy was used
        def print_stats(self):
            num_wins = np.count_nonzero(self.sarsp_arr[:,0] == self.win_state)
            num_lose = np.count_nonzero(self.sarsp_arr[:,0] == self.lose_state)

            print(f'Number of games: {self.num_games}')
            print(f'Number of wins: {num_wins}')
            print(f'Number of losses: {num_lose}')
            print(f'Win Percentage: {num_wins / self.num_games : .3f}')

            return

        # Simulate (num_games) games of BlackJack!
        def play_games(self):

            # Iterate through num_games
            for i in range(self.num_games):
                self.play_game()
                self.reset()

            # print(self.sarsp_arr)
            self.print_stats()

            # Output CSV file of runs if a random_policy was used
            if self.action_type == 'random_policy':
                self.output_sarsp_file()

            return

        # End BlackJack_game class ################################################

    def main():

        # Input parameters
```

```
    params = Params()
    assert (params.action_type in ['input', 'fixed_policy', 'random_policy']), "Action type
    must be 'input', 'fixed_policy', or 'random_policy'"

    # BlackJack_game object
    game = BlackJack_game(params)

    # Play one game if user is playing or simulate many if a random/fixed
    # policy is being used
    if params.action_type == 'input':
        game.play_game()
    else:
        game.play_games()

    return

if __name__ == "__main__":
   main()
```

B. Value Iteration Code

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Tue Nov  3 10:52:50 2020

@author: tristan
"""

import os
import pandas as pd
import numpy as np

#~~~~~~~~~~~~~~~~~~
state_mapping = 2
#~~~~~~~~~~~~~~~~~~

if state_mapping == 1:
    S = 21
    A = 2
    gam = 0.5
    maxIters = 100
    input_file = "random_policy_runs_mapping_1.csv"
    output_file = 'Value_Iteration_Policy_1.policy'
elif state_mapping == 2:
    S = 183
    A = 2
    gam = 0.5
    maxIters = 100
    input_file = "random_policy_runs_mapping_2.csv"
    output_file = 'Value_Iteration_Policy_2.policy'

df = pd.read_csv(input_file)

s_data = df['s']
a_data = df['a']
r_data = df['r']
sp_data = df['sp']

# Initialize action value function to all zeros
U = np.zeros((S))
u = np.zeros((A))

# Initialize transition function to all zeros
T = np.zeros((S, S, A))

# Initialize reward value function to all zeros
R = np.zeros((S, A))

# Initialize sum of rewards function to all zeros
rho = np.zeros((S, A))

# Initialize reward value function to all zeros
N = np.zeros((S, A, S))

# Initialize policy to all zeros
policy = np.zeros((S))
```

```
# Learn the model
for k in range(len(df)):
    s = s_data[k]
    a = a_data[k]
    r = r_data[k]
    sp = sp_data[k]

    N[s, a, sp] += 1
    rho[s, a] += r

    if(sum(N[s, a, :]) == 0):
        T[sp, s, a] = 0
        R[s, a] = 0
    else:
        T[sp, s, a] = N[s, a, sp] / sum(N[s, a, :])
        R[s, a] = rho[s, a] / sum(N[s, a, :])


#Value Iteration
for s in range(S):
    for i in range(maxIters):
        for a in range(A):
            u[a] = R[s,a] + gam*sum(T[:, s, a] * U[:])
            U[s] = max(u)
        if i == maxIters-1:
            policy[s] = np.argmax(u) #take greedy action once converged

# Get output file name and path
output_dir = os.getcwd()
output_file = os.path.join(output_dir, output_file)

# Open output file
DF = open(output_file, 'w')

# Iterate through each value in policy, writing to output file
for i in range(S):
    DF.write(f'{int(policy[i])}\n')

# Close output file
DF.close()
```

C. Sarsa Code

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Nov  1 14:01:04 2020

@author: tristan
"""

import os
import pandas as pd
import numpy as np

#~~~~~~~~~~~~~~~~~~
state_mapping = 1
#~~~~~~~~~~~~~~~~~~

if state_mapping == 1:
    S = 21
    A = 2
    gam = 0.5
    alpha = 0.01
    maxIters = 5
    input_file = "random_policy_runs_mapping_1.csv"
    output_file = 'Sarsa_Policy_1.policy'
elif state_mapping == 2:
    S = 183
    A = 2
    gam = 0.5
    alpha = 0.01
    maxIters = 5
    input_file = "random_policy_runs_mapping_2.csv"
    output_file = 'Sarsa_Policy_2.policy'

df = pd.read_csv(input_file)

s_data = df['s']
a_data = df['a']
r_data = df['r']
sp_data = df['sp']

# Initialize action value function to all zeros
Q = np.zeros((S, A))

for i in range(maxIters):
    print(i)
    for k in range(len(df)-1):
        s = s_data[k]
        a = a_data[k]
        r = r_data[k]
        sp = sp_data[k]
        ap = a_data[k+1]

        Q[s, a] = Q[s, a] + alpha*(r + gam*Q[sp, ap] - Q[s, a]) #Sarsa
        #Q[s, a] = Q[s, a] + alpha*(r + gam*max(Q[sp, :]) - Q[s, a]) #Q-learn
```

```
policy = np.argmax(Q, axis=1)

# Get output file name and path
output_dir = os.getcwd()
output_file = os.path.join(output_dir, output_file)

# Open output file
DF = open(output_file, 'w')

# Iterate through each value in policy, writing to output file
for i in range(S):
    DF.write(f'{policy[i]}\n')

# Close output file
DF.close()
```

D. Q-Learning Code

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Oct 19 15:02:13 2020

@author: joshuageiser
"""

import os
import pandas as pd
import numpy as np
import time

class const():
    '''
    Class that contains various constants/parameters for the current problem
    '''
    def __init__(self):
        self.gamma = 0.5
        self.input_filename = 'random_policy_runs_mapping_2.csv'
        self.output_filename = 'QLearning_policy_mapping_2.policy'
        self.n_states = 183 # 21 for state mapping 1, 183 for state mapping 2
        self.n_action = 2
        self.alpha = 0.01

        self.lambda_ = 0.1


def update_q_lambda(Q_sa, N_sa, df_i, CONST):
    '''
    Note: not used in final implementation
    '''

    # Update visit count
    N_sa[df_i.s][df_i.a] += 1

    # Temporal difference residue
    diff = df_i.r + (CONST.gamma * max(Q_sa[df_i.sp])) - Q_sa[df_i.s][df_i.a]

    # Update action value function
    Q_sa += (CONST.alpha * diff * N_sa)

    # Decay visit count
    N_sa *= CONST.gamma * CONST.lambda_

    return

def train_q_lambda(input_file, CONST):
    '''
    Note: not used in final implementation
    '''

    # Read in input datafile
    df = pd.read_csv(input_file)
```

```
    # Initialize action value function to all zeros
    Q_sa = np.zeros((CONST.n_states, CONST.n_action))

    # Initialize counter function
    N_sa = np.zeros((CONST.n_states, CONST.n_action))

    # Iterate through each sample in datafile
    for i in range(len(df)):
        df_i = df.loc[i]
        update_q_lambda(Q_sa, N_sa, df_i, CONST)

    # Policy is the index of the max value for each row in Q_sa
    policy = np.argmax(Q_sa, axis=1)

    # Write policy to file
    write_outfile(policy, CONST)

    return


def update_q_learning(Q_sa, df_i, CONST):
    '''
    Perform Q-Learning update to action value function for a single sample
    '''

    # Temporal difference residue
    diff = df_i.r + (CONST.gamma * max(Q_sa[df_i.sp])) - Q_sa[df_i.s][df_i.a]

    # Update action value function
    Q_sa[df_i.s][df_i.a] += CONST.alpha * diff

    return

def train_q(input_file, CONST):
    '''
    Train a policy using Q-learning algorithm and input datafile containing
    sample data
    '''

    # Read in input datafile
    df = pd.read_csv(input_file)

    # Initialize action value function to all zeros
    Q_sa = np.zeros((CONST.n_states, CONST.n_action))

    # Iterate through each sample in datafile
    for i in range(len(df)):
        df_i = df.loc[i]
        update_q_learning(Q_sa, df_i, CONST)

    # Policy is the index of the max value for each row in Q_sa
    policy = np.argmax(Q_sa, axis=1)

    # Write policy to file
    write_outfile(policy, CONST)

    return
```

```
def write_outfile(policy, CONST):
    '''
    Write policy to a .policy output file
    '''

    # Get output file name and path
    output_dir = os.getcwd()
    output_file = os.path.join(output_dir, f'{CONST.output_filename}')

    # Open output file
    df = open(output_file, 'w')

    # Iterate through each value in policy, writing to output file
    for i in range(CONST.n_states):
        df.write(f'{policy[i]}\n')

    # Close output file
    df.close()

    return


def main():

    start = time.time()

    CONST = const()

    input_file = os.path.join(os.getcwd(), CONST.input_filename)

    train_q(input_file, CONST)
    #train_q_lambda(input_file, CONST)

    end = time.time()

    print(f'Total time: {end-start:0.2f} seconds')
    print(f'Total time: {(end-start)/60:0.2f} minutes')


    return

if __name__ == '__main__':
    main()
```