COMP 560 A2 Short Response

Sunnie Kwak, Jacob Gersfeld, Kinsey Ness

Model-Based Approach

For the Model-based RL approach, we first read the data from the text file and stored it in a dictionary called `data`, organized as follows: {s : {a : {s' : given P(s'| s, a) }}}. This made later data retrieval easy. We also created two extra dictionaries to store the transition probabilities that were structured similarly to the `data` dictionary: one holding the frequency of each s, a, s' triplet occurring during our learning process (structured {s : {a : {s' : frequency }}}) and one holding the transition probabilities of each s, a, s' triplet (structured {s : {a : {s' : calculated P(s'| s, a) }}}). The dictionary holding the frequencies was just used to easily calculate the transition probabilities by taking frequency(s,a,s')/total frequency(s,a). The `utilities` dictionary stored the utility values of each state ({s: U(s)}).

Every iteration of the learning process starts with "Fairway" as the starting state. The exploration rate value determines whether or not the system would explore or exploit by choosing a random floating-point value between 0 and 1. If this random value is less than the exploration rate, the system would choose to explore. Else, it would choose to exploit. The exploration rate is initially set to 1 and decreases more and more over time. This was done by decreasing the exploration rate by the exploration decay rate and also increasing the exploration decay rate by a very small amount every iteration. With the values we assigned to these variables, the ratio of the number of explorations to the number of exploitations was roughly 2:1, with exploration happening mostly in the first ⅔ of the iterations and exploitation happening in the last ⅓ of iterations. With the exploration value being 1 or close to 1, all state spaces were explored, and the resulting policy showed that for most runs, it was best to shoot At the pin on the Fairway and the Ravine. When the exploration value started off closer to 0, the system chose to exploit more, and many state spaces were left unexplored. Therefore, the resulting policies would not be consistent. I found that having a low exploration rate resulted in a policy that would switch off from At pin to Past pin as the optimal action for both Fairway and Ravine. We set our initial exploration rate to 1 because we wanted to make sure all state spaces were explored before determining and choosing optimal actions.

If the system chooses to explore, it would choose a random action from the given current state. With this action, it would use the given probabilities to determine which state it would end up in. Using the `transitions` dictionary, the frequency of s, a, s' occurring is updated (increased by 1). The `transitions` dictionary helps calculate the transition probabilities to store in the `transition_prob` dictionary. The utility values for states is also updated with the updated transition probabilities. s' is returned and is set to the current state in the `main` method.

If the system chooses to exploit, it first calculates the utilities of each action given the current state and will choose to take the action with the best utility. The utility values are calculated with the Bellman equation. The reward is set to 1 for each stroke taken, meaning that when we choose the best utility, we will actually want to choose the minimum value, since for golf, we want to yield the lowest number of strokes possible. The discount value is set at .7, but this can also be changed. We found that when the discount value was set closer to 0, the utility values of all states were similar and close to 1 (the reward), meaning that more weight was given to the immediate reward. When the discount value was closer to 1, the utility values were more varied and more weight was given to later reward. I found that

when the discount was very close to 1 (.9 and up), the policy for Fairway was more likely to be shoot Past the pin rather than At the pin, whereas discount values .8 and below seemed to favor At the pin over Past the pin for Fairway. As the discount value got closer to 0, the policy was more consistent in outputting At the pin as the optimal action to take for Fairway. Each time the utility values were updated, the loop for computing the utility values would run until the values started to converge, since running the utility value calculation once might not yield proper values. Once the program determined that the utility values were beginning to converge (in our code, +/- .0005 off from the previous iteration), it would stop looping through and return the action that yielded the minimum utility value. Then, a new state s' is chosen again based on the returned action and given environment probabilities, and the transition probabilities are updated again for this s, a, s' triplet. Again, s' is returned and is set to the current state in the `main` method.

This loop continues until the state becomes "In". The epsilon value will determine how many times this learning process runs, with each process starting at state "Fairway" and ending with "In". The epsilon value is set to 10000 because we found that it yields consistent, accurate results without taking too long; although, we also found that 100000 was also an acceptable epsilon value. If the epsilon value was set too low, accuracy was compromised, for the system was unable to explore all possible states and actions. We found that even with epsilon set to 1000, not all s, a, s' triplets were explored, but with epsilon set to 10000, all state spaces were consistently explored. If the epsilon was set too high, the code would run for too long. The program stops running once the process of getting from Fairway -> In has been run "epsilon" number of times.

To print the P(s'| s, a) of each triplet, we simply looped over our dictionary holding the transition probabilities for all s, a, s' triplets. To print the policy, we ran the function that calculates utility and returns the optimal action for each state.

Model-Free Approach

Our model-free approach uses Q-learning to determine the optimal action to take at a given state. To implement Q learning, we created a Q matrix that represents the Q value for each state/action pair. The Q value represents the utility value of each action. The basis of our model-free approach is the same as our approach to model-free: our data structures and the explore/exploit calculations are the same.

The process our program goes through starts the agent at the starting state "Fairway" for each episode. An episode (hole) consists of steps (strokes) that the agent (golfer) takes. At each step, the program determines whether to explore or exploit the environment. It then chooses an action and end state for that step. This process repeats until the agent reaches the end goal "In".

When choosing an action based on the current state an agent is in, if our randomization process has chosen exploration, our program will randomly choose an action. If exploitation was selected, our program will take the action with the highest Q value currently in the table and either reinforce that state/action pair or dispel it.

Once the action for a given step is chosen, our model-free approach chooses s' based on the probabilities provided in the input. This s' is chosen using numpy's random choice function, which takes in the possible resulting states with weighted probabilities to properly choose an s'.

The Q value is then updated in the Q matrix with the Bellman's equation for Q values:
$Q(s, a) = R(s) + \gamma \sum P(s' | s, a) \max Q(s', a')$. In this equation, R(s) represents the reward for getting to a given state. To get the most accurate results, we assigned a value of 0.5 for getting to state "Close" because being in that state gives the agent the greatest chance to get to the goal state, "In". We assigned the state "In" a reward of 1, and all other states are assigned a reward of 0. Having reward values set up in this way allows the agent to learn which states should be targeted over others.

Once the Q values are updated after each step, the current state is updated to be s', and the process is repeated.

Once the agent has completed 10,000 episodes, the program completes and prints out the utility values for each state/action/state triplet, along with the optimal policy for each state. For 10,000 episodes, the exploration decay rate was set to .0001 (1/10000) to ensure that the program explores plenty before it begins to exploit the dataset. A learning rate of 0.1 also seemed to suit the program well, as each iteration will have a mild to moderate impact without impacting it too intensely for less favorable/rare outcomes. The policy function uses the highest Q value at each state in the table to recommend to the agent the best action to take in any given state. The utility uses the Q values with weighted probabilities to determine a U(s, a, s') value for each state/action/s' triplet.

Model-based RL was mostly done by Sunnie Kwak; Model-free RL was mostly done by Jacob Gersfeld & Kinsey Ness. All members worked on the write-up (Sunnie Kwak- Model-based section, Jacob Gersfeld & Kinsey Ness - Model-free section).