Assignment 2: Reinforcement Learning

Setup

We set up an Info class to store the state/action/state/probability information we would be receiving. This is initialized with Strings corresponding to starting location, direction, result, and probability, with the option to set a utility and reward. All of these properties also have getter methods. In the main class, an arrraylist of Info objects was created to store the data we would be receiving and a scanner was initialized. A while loop will run while the scanner has a next line to read. The scanner reads the line and splits every word separated by a "/" into a String array. The strings in this array are used to initialize a new Info object and add it into our list of data. The loop ends once the start location is "Close" and the result is "In" because that should be the last piece of information in the list.

Then we calculated a "cumulative probability" of sorts for each state/action/state. We created a for loop to run through the data list that starts at position 1, and continues if both the start location and the action is the same as the info in the position before it. The probability of this Info is then set to be the sum of its original probability and all of the ones before it within the start/action pair, since the probabilities within each state/action pair add up to 1.

Golf Simulator

We created a GolfSimulator function that takes in a starting location and action as parameters and will return a resulting position based on the transition probabilities given. A random double was generated and this will determine which result is picked. First, a for loop runs for the data list to narrow down the Info in the list that matches the start and action provided as arguments. This takes us to the first state/action Info in the data. If the double generated is less than or equal to the probability for that Info, then that resulting location is returned. Else, we check the condition for the next Info in the list. This is done in a while(true) loop, but since we are using "cumulative" probabilities, the values are ascending and the highest is 1 so we know the loop must terminate and the function must return a result.

pickAction

We created a pickAction helper method that will randomly pick a possible action to take given the starting location as an argument. A random double is generated again and there are only a few options here. If the starting location is the fairway or ravine, there is a ⅓ chance of shooting "at", "past", or "left". So if the double is less than 0.33, "at" is returned, if it is less than 0.66, "past" is returned, else "left" is returned. Results for the rest of the states were set up in the same manner.

Model Free Learning

For model-free learning, we chose a largely exploitative strategy. 10% of the time, the model will randomly select the next action, and 90% of the time the model chooses the action that currently has the highest utility. Utililty is updated when a golf game is completed. During the game simualtion, we keep an array of the moves that occured during that game, as well as a count of the number of total turns taken. The utility value is then updated for each state/action pair that occured in the game, calculated by updating the inverse of average of the number of turns taken in games where that state/action pair was used. When updating the utilities, the number of turns is discounted with a constant discount value of 0.9 to bias the long-term outcome rather than immediate rewards. Utility is normalized accross all state/action pairs to be between 0 and 1. The higher the utility, the more consequential the state/action pair is to reducing the overall score.

Model Based Learning

The first step is to generate new transition probabilities. A new array list newProbs was created to store these. A for loop for each Info in the original data ran to add elements to newProbs with the same arguments, but probability for each Info was set to 0. Then a for loop was created. An integer variable "turns" is set to 0, and this will be used to count the number of turns which will translate to rewards. Since we always start a game from the Fairway, pickAction was called to generate an action from there, and these two arguments were provided as arguments to call GolfSim. A for loop then loops through newProbs to update the transition probability for that Info by finding the matching start, action, and result from GolfSim. For each state/action/state set, we keep track of the number of times that triplet occurred. A while loop is started to run under the condition that the result does not equal "In", because we need to keep playing until the ball goes in the hole. Turns is incremented, the result becomes the start, and the same steps as above are implemented again: pickAction, GolfSim, and updating the probability. The probability is calulated by taking the number of total times the model was in the first state/action pair of the state/action/state triplet, and dividing the number of times the action resulted in the result state of the triplet. Once the for loop are done, we update newProbs to find the average rewards. For each Info in newProbs, the state/action/state/probability line is printed out. Then we call a helper method maxUtilityBellman to calculate utility values.

maxUtilityBellman

This is a helper function that calculates the utility values using the Bellman equation and determines the optimal policy. It takes in an array list of Info and a double discount [factor] as parameters. Doubles util, prevUtil, and sum, and String policy were declared at the beginning. Since we want to start with the first value in the list, the sum is initialized to be the probability * reward of that first Info at position 0. prevUtil is 0 and the policy is just the direction of that first Info. Then a for loop starts from i = 1 to run the length of the list. If the start and action of the Info is the same as the one before it, we are at a place to recalculate the sum, which equals the current sum + the probability * reward of that current Info. Once the action is no longer the same, we are at a point to calculate the utility, which equals 1 + the discount provided in argument * sum. If this util is greater than prevUtil, it becomes the new prevUtil value to beat, and the policy is set to be the direction for this Info (actually the Info at i-1 because we are one ahead). This means that a utility value will be calculated for each state/action pair. Once the start location is no longer the same as the previous, it is time to print the policy. The prevUtil is reset to 0, the sum is reset to the probability * reward at that Info, and the policy is reset to be the direction at that Info.

Questions

1) Changing the values for exploration and exploitation did not ultimately impact the outcome of the policies since we had a high degree of iterations, however, we did observe that the difference in utility value for the generally suboptimal locations (especially the ravine) were less significant when the model was predominantly exploitative. This is an intuitive result, because the exploitative model will quickly favor actions that drive it away from the ravine due to the ravine's almost universally negative impact on the overall number of turns taken. However, this means that there is simply less robust testing behind the most appropriate policy when one is actually in the ravine. The more exploratory model will develop the robustness of each policy more evenly across states.

2) We decided to stop learning when the maximum difference found in probability/utility values 75% of the way through the max iterations and 99% through the max iterations was marginal. In other words, we stopped learning when the values stopped changing. For probability, the threshold difference was 10%, and for utility, it was 1%.

3) Lowering the initial discount values did not substaintially impact the policy outcomes for the model-based learned, but for the model-free learner, which was highly exploitative, lowering the discount value compounded the aforementioned effect of reducing the frequency with which the ravine was visited, because when looking at short-term utility, the ravine is highly unfavorable. This did not change the winning policies, but narrowed the margin of victory for the winning policies in the ravine.

4) Changing the epsilon values also did not ultimately impact the resulting policies, but the difference in probabilities was noticable. With a higher threshold required for stopping iterations, the calculated transition probabilities grew much closer to the input probabilities. The impact was most profound when using an exploratory model. The model-free learner was not profoundly impacted by a change in epsilon values, most likely because it is a highly exploitative model and the state/action pairs are examined in a greedy manner.

Contributions

Maina -set up Info class, GolfSim and pickAction helper methods, initial model based learning and maxUtilityBellman

Danielle - wrote model-free learning function, cleaned up and added epsilon and exploration/exploitation values to model-based learning