Hidden Markov Model

Note that not all code can be provided in this report. Also, some code snippets have places where code is omitted, as, in these cases, the omitted material is not as relevant to the accompanying explanations. Sometimes code goes past the page breaks, but this, unfortunately, cannot be helped for a particularly lengthy method.

1 Introduction

In this report, we tackle the blind robot problem we once visited in our Maze World assignment, with a probabilistic twist. That is, we now are trying to find the robot's likeliest location based only upon a sequence of sensory details. Each square of the maze will have its own color, allowing the robot to report its color at each point in its path. However, the robot has a broken sensor, meaning it does not work a certain amount of the time. We make use of filtering to calculate the likelihood of the robot being in a certain place, given the chance that it was in one of the previous states. The idea of the first-order hidden markov model is that we can try having a future state (a set of probabilities of being at a certain location) depend only on the previous state.

2 Structure of Maze

We have two classes: Maze and BlindRobot. The former will take care of everything that is algorithmic. This includes solving for the various probabilities and finding the likeliest place of the robot given the color sensory details. To get these details, we use the BlindRobot class. It takes a Maze object as input and traverses a random path within this maze, all the while providing sensory details, some of which may be wrong at a certain rate.

2.1 Probabilistic Theory

We do not use explicit matrices to perform out calculations, so actually understanding the meaning behind the operations is essential. Both approaches are isomorphic, but it may be difficult to see the state transition and sensor models right away.

The way we figure out the probability of a blind robot being in a certain square given the previous distribution and the sensed color reading is by first finding the probabilities of having moved north, south, east, or west. By using the color reading, we know that there is a certain probability that we are now at the sensed color. Thus, we want to calculate probabilities of moving in a certain direction when the sensor is right and when it is wrong.

2.2 Update

As said before, we find the actual probabilities of moving in a certain direction in thie step. We also, however, handle the actual updating of probabilities in the state to the new ones. The basic gist of this is that, to compute the probability of being in a position, we need to see that the probability that the previous position was to the left of the the current position multiplied by the probability of moving to the right, ..., etc. We do for every direction and sum. This amounts to the same sort of calculation, but is much different in its implementation.

```
1   public void update(int color){
2       double [] probMovesCorrect = new double [4]; //N, S, E, W
3       double [] probMovesIncorrect = new double [4]; //N, S, E, W
4       double [] [] updatedProbs = new double[probs.length][probs[0].length];
5
6       //Find the probabilities of moving in a certain direction
7       for(int i = 0; i < coloring.length; i++){
8           for(int j = 0; j < coloring[0].length; j++){
9
10              if(coloring[i][j] != -1){
11                  boolean hasNorth = false, hasSouth = false, hasEast = false, hasWest =
                        false;
12                  int surrounding = 0;
13                  /* HANDLING CASE WHERE SENSOR IS CORRECT */
14                  //get number of surrounding color
15                  if((coloring[i][j] == color && (i == 0 || coloring[i - 1][j] == -1)) || (i !=
                        0 && coloring[i - 1][j] == color)){//Not north-most or south of wall
16                      surrounding++;
17                      hasNorth = true;
18                  }
19          ... Similar for other directions
20
21                  //updating probs
22                  if(surrounding > 0){
23                      if(hasNorth){
24                          probMovesCorrect[0] += probs[i][j]/surrounding;
25                      }
26              ...
27                  }
28                  hasNorth = false;
29                  hasSouth = false;
30                  hasEast = false;
31                  hasWest = false;
32
33                  surrounding = 0;
34                  /* HANDLING CASE WHERE SENSOR IS NOT CORRECT */
35
36                  if((coloring[i][j] != color && (i == 0 || coloring[i - 1][j] == -1)) || (i !=
                        0 && coloring[i - 1][j] != color)){//Not north-most or south of wall
37                      surrounding++;
38                      hasNorth = true;
39                  }
40
41              ...
42                  if(surrounding > 0){
```

```
43                                          if(hasNorth){
44                                                  probMovesIncorrect[0] += probs[i][j]/surrounding;
45                                          }
46
47                                      ....
48                                  }
49                              }
50                          }
51                  }
52
53                  //Update probabilities
54                  for(int i = 0; i < coloring.length; i++){
55                          for(int j = 0; j < coloring[0].length; j++){
56                                  if(coloring[i][j] != -1){
57                                          //Moving up
58                                          if(((i == 0 || coloring[i - 1][j] == -1) && coloring[i][j] == color)){
59                                                  updatedProbs[i][j] += (1 - errorRate) * probs[i][j] *
                                                        probMovesCorrect[0];
60                                          } else if(i != 0 && coloring[i - 1][j] == color){//Not north-most or
                                                south of wall
61                                                  updatedProbs[i - 1][j] += (1 - errorRate) * probs[i][j] *
                                                        probMovesCorrect[0];
62                                          } else if (((i == 0 || coloring[i - 1][j] == -1) && coloring[i][j] != color)){
63                                                  updatedProbs[i][j] += errorRate * probs[i][j] * probMovesIncorrect
                                                        [0];
64                                          } else if(i != 0 && coloring[i - 1][j] != color && coloring[i - 1][j] != -1)
                                                {
65                                                  updatedProbs[i - 1][j] += errorRate * probs[i][j] *
                                                        probMovesIncorrect[0];
66                                          }
67
68                                          //Move to from above
69                                      ...
70                                          //Move to from west
71                                          ...
72                                          //Move to from east
73                                      ...
74                                  }
75                          }
76                  }
77
78          probs = normalize(updatedProbs);
79
80          }
```

2.3 Normalization

After this update step, the probabilities will sum up to something less than 1. This is because we are multiplying the probabilities, in general, by numbers that sum up to much less than one. Thus, we are required to scale the probabilities to 1 after every iteration of the algorithm.

```
1           public double [][] normalize(double [][] matrix){
```

```
2               double total = 0;
3
4               for(double [] arr : matrix){
5                       for(double d : arr){
6                               total += d;
7                       }
8               }
9
10              for(int i = 0; i < matrix.length; i++){
11                      for(int j = 0; j < matrix[0].length; j++){
12                              matrix[i][j] /= total;
13                      }
14              }
15
16              return matrix;
17      }
```

2.4 maxesContain

This is a specifically constructed way of assessing how correct the algorithm is performing. It looks at the top x positions by probability magnitude and checks whether they contain the given position. It is actually quite remarkable how the numbers jump up from finding how often the predicted robot position is in the top 1 maximum to the top 2 maxima.

```
1   public boolean maxesContain(int [] arr, int top){
2               ArrayList<double []> coords = new ArrayList<>();
3               for(int i = 0; i < probs.length; i++){
4                       for(int j = 0; j < probs[0].length; j++){
5                               double [] array = new double[3];
6                               array[0] = i;
7                               array[1] = j;
8                               array[2] = probs[i][j];
9
10                              coords.add(array);
11                      }
12              }
13              coords.sort(new Comparator<double []>(){
14
15                      @Override
16                      public int compare(double[] o1, double[] o2) {
17                              return (int)(-100000*((o1)[2] - (o2)[2]));
18                      }
19
20              });
21
22              for(int i = 0; i < top; i++){
23                      if(coords.get(i)[0] == arr[0] && coords.get(i)[1] == arr[1])
24                              return true;
25              }
26
27              return false;
28      }
```

2.5 Main

While we discuss this part of the program in this section, it really is in a class of its own (though it appears in the same .java file as Maze).

```
1  public static void main(String [] args){
2                  int [][] coloring = {{1,2,3,4},{4,3,2,−1},{1,2,4,3},{4,−1,3,2}};
3                  Maze maze = new Maze(coloring, 4);
4                  maze.printMaze();
5                  BlindRobot bp = new BlindRobot(maze);
6                  bp.runPath(100, 0, 0);
7
8                  int corrCount = 0;
9                  int totalCount = 0;
10                 double max = 0;
11                 for(int i = 0; i < bp.sensedPath.size(); i++){
12                         maze.update(bp.sensedPath.get(i));
13                         System.out.println("Sensed color: " + bp.sensedPath.get(i));
14                         System.out.println("MAZE (Robot at): (" + bp.realPath.get(i)[0] + ", " + bp.realPath.get(
                               i)[1] + ")");
15                         System.out.println(maze.probs[bp.realPath.get(i)[1]][bp.realPath.get(i)[0]]);
16
17                                 //if(maze.getMax()[0] == bp.realPath.get(i)[0] && maze.getMax()[1] == bp.
                                       realPath.get(i)[1]){
18                                 if(maze.maxesContain(bp.realPath.get(i), 2)){
19                                         corrCount++;
20                                         System.out.println("yes");
21                                 }
22                                 totalCount++;
23                         maze.printMaze();
24                 }
25
26                 System.out.println(corrCount + "/" + totalCount);
27                 System.out.println(max);
28         }
```

2.6 Example Runs

The first example was the one on the class site.
Length 101 Path, top 1:

```
1  0.07 (1)  0.07 (2)  0.07 (3)  0.07 (4)
2  0.07 (4)  0.07 (3)  0.07 (2)  #
3  0.07 (1)  0.07 (2)  0.07 (4)  0.07 (3)
4  0.07 (4)  #  0.07 (3)  0.07 (2)
5  Sensed color: 1
6  MAZE (Robot at): (0, 0)
7  0.25038799793067773
8  yes
9  0.25 (1)  0.04 (2)  0.04 (3)  0.05 (4)
10 0.04 (4)  0.04 (3)  0.05 (2)  #
11 0.22 (1)  0.05 (2)  0.04 (4)  0.05 (3)
```

```
0.05 (4)  #  0.04 (3)  0.05 (2)
...
MAZE (Robot at): (2, 2)
0.6720473232962887
yes
0.00 (1)  0.01 (2)  0.00 (3)  0.01 (4)
0.08 (4)  0.00 (3)  0.01 (2)  #
0.00 (1)  0.01 (2)  0.67 (4)  0.07 (3)
0.00 (4)  #  0.05 (3)  0.09 (2)
67/101
```

Note that theway to read this is to see that 67/101 refers the number of times the robot location is the top 1 prediction.
Similarly, Length 101 Path, top 2:

```
...
...
MAZE (Robot at): (0, 1)
0.04019328616661506
yes
0.05 (1)  0.42 (2)  0.01 (3)  0.00 (4)
0.04 (4)  0.01 (3)  0.15 (2)  #
0.02 (1)  0.29 (2)  0.00 (4)  0.00 (3)
0.01 (4)  #  0.00 (3)  0.00 (2)
94/101
0.0
```

The hashtags refer to wall locations (zero probability of robot being there) and the numbers enclosed in paranthesis refer to different colors.

3 Structure of BlindRobot

The aim of BlindRobot is to traverse a random path of a given length that reports colors with an accuracy proportional to the errorRate.

3.1 runPath

The goal is to maintain two lists: one of actual robot locations and another of color sensors. With the former, we can actually assess how often we are correct, and, with the latter, we can use as input for our HMM problem.

```
        sensedPath = new ArrayList<>();
                realPath = new ArrayList<>();

                int [] coords = {x, y};
                realPath.add(coords);

                ArrayList<Integer> colors = new ArrayList<>();
                ArrayList<Integer> colorsPath = new ArrayList<>();
                colorsPath.add(maze.colorAt(x, y));
```

```
10
11              for(int i = 1; i <= maze.numColors; i++){
12                      colors.add(i);
13              }
14
15              for(int i = 0; i < length; i++){
16                      ArrayList<int []> reachable = getReachable(coords[0], coords[1]);
17
18                      coords = reachable.get((int)(Math.random() * reachable.size()));
19                      realPath.add(coords);
20
21                      int color = maze.colorAt(coords[0], coords[1]);
22                      colors.remove(Integer.valueOf(maze.colorAt(coords[0], coords[1])));
23
24                      if(Math.random() < errorRate){
25                              colorsPath.add(colors.get((int)Math.random() * (colors.size())));
26                      } else {
27                              colorsPath.add(color);
28                      }
29                      colors.add(color);
30              }
31
32              sensedPath = colorsPath;
```