Probabalistic Reasoning

Carter J. Bastian

March 1, 2016

Introduction

In this assignment, a Hidden Markov Model was used to integrate sensor information into a Mazeworld-like problem proposed by Rob Schapire. Further, three different techniques were used to elicit information about the most probable state at various times: filtering for forward-oriented prediction, smoothing for retrospective analysis, and best-path analysis for a holistic view of the problem's sequence.

Setting up the Problem

Much of the code to set up mazes was borrowed from Mazeworld's `Maze.java`file. However, some slight extensions were made. First, when a maze is generated, each space is also given a color (via the `colorIn()`function). Second, in addition to the `getPath()`function, which returns a random path of a specified length, there are also `getColorPath()`and `getCorrectColorPath()`functions, which return the (randomly-generated) set of observations and the factually-correct set of observations.

Notice that, when solving the problem, the program only ever has access to the set of colors observed along the path, not the correct set of colors for the path. Further, this set up is reflected in the first section of the program's output. A pictoral representation of the maze, the colors, and a full listing of the path is displayed before results are shown.

Designing the Models

In designing the models, there were three considerations to be had. First, the method of representation for the sensor model had to be designed. Second, the analogous method of representation for the transition model had to be made explicit. Finally, the meaning and structure of a "message", be it forward, backward, or viterbi, had to be defined.

As far as the sensor model, I chose to implement it as a set of four diagonal matrices (one for each type of evidence). For example, on the diagonal for the red sensor model, `rSensorModel`, would be the probabilities of seeing the color red given that the problem is in each possible state (in this case, S_0 through S_{15}.

As such, in each type of solution, at each step, the sensor model used would be based on the evidence observed. A partial listing of the initialization of these matrices is provided below:

```
/* Build the sensor Models */
// Sensor model for red
c = 'r';
model = new double[stateCount][stateCount]; // Initialized to zeros
for (state = 0; state < stateCount; state++) {
  XY = Maze.StatetoXY(state);
  if (m.getColor(XY[0], XY[1]) == c) // This state is the given color
    model[state][state] = 0.88;
  else // The state is not the given color
```

```
        model[state][state] = 0.04;
    }
    rSensorModel = model;
```

Second, the transition model was also implemented as a matrix of doubles. In this case, the element of the array at row i and column j would be the probability of moving from state S_i to state S_j at any given iteration of the problem. Below is a listing of the for loop used to build this transition model

```
/* Build the transition Model */
// For each state in the system
transitionModel = new double[stateCount][stateCount];
for (state = 0; state < stateCount; state++) {
  legalMoves = 0;
  moveProb = 0.0;
  XY = Maze.StatetoXY(state); // Get the (x,y) coordinates of this state

  // Find out how many available actions there are
  for (int[] action : Maze.actions)
    if (m.isLegal(XY[0] + action[0], XY[1] + action[1]))
      legalMoves += 1;

  if (legalMoves == 0) // Edge case of state being on a wall surrounded by walls
    moveProb = 0.0;
  else
    moveProb = 1.0 / legalMoves;

  // Fill in with probability of transitioning from state -> nextState
  for (int[] action : Maze.actions) {
    if (m.isLegal(XY[0] + action[0], XY[1] + action[1])) {
      nextState = Maze.XYtoState(XY[0] + action[0], XY[1] + action[1]);
      transitionModel[state][nextState] = moveProb;
    }
  }
}
```

Notice also that, when this transition model is built, it's transpose is also built and saved for the sake of the Filtering algorithm's equation for a next sequential message. This is built as follows:

```
/* Build its transpose (for Filtering equation) */
// This loop has god-awful locality. Sean Smith would kill me.
transitionTranspose = new double[stateCount][stateCount];
for (int y = 0; y < stateCount; y++)
  for (int x = 0; x < stateCount; x++)
    transitionTranspose[y][x] = transitionModel[x][y];
```

Finally, the representation of a message was set to be an array of doubles such that, at element i, the element in the message array is the probability at the current time that the problem is in state S_i. The initial forward message was initialized to be a vector of equal probabilities, and the initial backward message was initialized to be a vector of all 1.0. The initialization of these functions is listed below:

```
/* Build the backward initial condtion */
backwardInitialCondition = new double[stateCount];
Arrays.fill(backwardInitialCondition, 1.0);

/* Build the forward initial condition */
```

```
initProb = 1.0 / stateCount;            // All states are equi-probable
forwardInitialCondition = new double[stateCount];
Arrays.fill(forwardInitialCondition, initProb);
```

Notice first that this set up of the design elements entailed in the project is implemented in the `setUp()` function of the `SchapireProblem` sub-class in the `SchapireDriver.java` file. Also note that all of these models are implemented as state-variables in this sub-class and that the results after their creation are listed in the program's output.

The output from this set up is after the problem set up in the program's output.

Filtering

HMM filtering was used to predict the probability of each possible state at each step in the process. Specifically, at each step in the problem, the message was computed to be the normalization of the chosen sensor model, multiplied by the transpose of the transition matrix, multiplied by the forward message at the previous iteration.

For the sake of simplicity, the matrix algebra methods are not listed (`matrixMultiply()`, `normalize()`, etc.). However, a full implementation of the filtering algorithm is listed below:

```
/**
 * Implements Markov Chain Filtering to produce probability distributions
 */
public void solveFiltering() {
  double[][] sensorModel = null;
  double[] forwardMessage;
  int[] XY;

  // Initialize with the initial probability distribution
  forwardMessage = forwardInitialCondition;

  for (int step = 0; step < randomPath.length; step++) {
    // Select the appropriate sensor model based on the evidence
    switch (evidence[step]) {
      case 'r':   sensorModel = rSensorModel;
                  break;
      case 'b':   sensorModel = bSensorModel;
                  break;
      case 'g':   sensorModel = gSensorModel;
                  break;
      case 'y':   sensorModel = ySensorModel;
                  break;
    }

    // f_1:t+1 = alpha * SensorModel_t+1 * TransitionTranspose * f_1:t
    forwardMessage = normalize(
                       matrixMultiply(sensorModel,
                         matrixMultiply(transitionTranspose, forwardMessage)));

    // Step one is the starting point for viterbi optimal-path finding
    if (step == 0)
      viterbiMessages[0] = forwardMessage;
```

```
      System.out.println("\nFiltered Distribuition at step " + (step+1) + ": ");
      System.out.println("(Real location is state " + randomPath[step] + ")");
      printMessage(forwardMessage, mazeDimension);

    }
  }
```

The results with this method were quite good. For example, in the sample problem (run by seeding the random number generator in `Maze.java`to the value 0), the final state was S_{12}. In the last iteration of running the filtering algorithm, the probability it found of the problem being in state S_{12} was 0.797.

However, especially towards the earlier iterations, some non-trivial error was observed. For example, in the second iteration, the problem was actually in state S_{15}. However, the forward message at this iteration showed gave state S_{15} a probability of .387 while it gave S_{14} a probability of .394. As such, it did provide us with an incorrect prediction.

Smoothing

The smoothing method, which was implemented as the forward-backwards algorithm, simply used the results from forward filtering in combination with an analogous message calculated in the backward-orientation. This backward message is the multiplication of the transition model with the matrix product of the selected sensor model and the backwards message at the previous iteration.

To then find the smoothed distribution of the data, one takes the normalization of element-wise multiplication of the forward and backwards messages. A partial implementation of the backwards-segment of this algorithm is listed below:

```
  // Backwards part of forward-backwards
  for (int step = randomPath.length - 1; step >= 0; step--) {
    distribution = normalize(messageMultiply(fVals[step], backwardMessage));

    // Print the results at this step
    System.out.println("\nSmoothed Distribuition at step " + (step+1) + ": ");
    System.out.println("(Real location is state " + randomPath[step] + ")");
    printMessage(distribution, mazeDimension);

    // Select the appropriate sensor model based on the evidence
    switch (evidence[step]) {
      case 'r':   sensorModel = rSensorModel;
                  break;
      case 'b':   sensorModel = bSensorModel;
                  break;
      case 'g':   sensorModel = gSensorModel;
                  break;
      case 'y':   sensorModel = ySensorModel;
                  break;
    }

    backwardMessage = matrixMultiply(transitionModel,
                        matrixMultiply(sensorModel,backwardMessage));
  }
```

Notice that this incorporation of future evidence into previous estimations resulted in improved results in the earlier iterations of the problem. For example, the incorrect prediction of being in S_{14} at iteration two was changed such that the smoothed estimate gave the prediction of being in S_{15} (the correct state) a probability

of .504, while it gave the prediction of being in S_{14} a probability of .416. This is a marked improvement from the incorrect prediction produced by filtering.

Finding the Most-Likely Path

Finally, in order to find the most-likely path of states through the problem, the Viterbi algorithm was used. Notice that this algorithm is exactly like forward prediction, except that instead of summing the probabilities over each state, the maximization was taken instead. As such, the the viterbiMessages (tracking the probabilities) and viterbiBackchains (linking each optimal path to it's preceding state) were created. A full implementation of this follows:

```
// Setup the path lengths
Arrays.fill(viterbiBackchains[0], -1);

// Loop through each step in the path
for (int step = 1; step < randomPath.length; step++) {
  double[] viterbiMessage = new double[stateCount];

  // Loop through each possible current state to fill out the viterbi message
  for (currState = 0; currState < stateCount; currState++) {
    maxState = -1;
    maxVal = 0.0;

    // Loop through each possible last state
    for (lastState = 0; lastState < stateCount; lastState++) {
      // Get the probability of an optimal path to the last state
      // times the probability of transitioning from the last state to the current
      tempVal = viterbiMessages[step - 1][lastState] *
                          transitionModel[lastState][currState];

      if (tempVal > maxVal) {
        maxVal = tempVal;
        maxState = lastState;
      }
    }

    // Update this index in the viterbiMessage
    viterbiMessage[currState] = maxVal;
    viterbiBackchains[step][currState] = maxState;
  }
  // Having the probabilities correctly in the viterbiMessage,
  // Find and apply the sensor model, normalize, and save in the global variable

  // Select the appropriate sensor model based on the evidence
  switch (evidence[step]) {
    case 'r':   sensorModel = rSensorModel;
                break;
    case 'b':   sensorModel = bSensorModel;
                break;
    case 'g':   sensorModel = gSensorModel;
                break;
    case 'y':   sensorModel = ySensorModel;
```

```
                    break;
            }

            viterbiMessages[step] = normalize(
                                        matrixMultiply(sensorModel,
                                          viterbiMessage));
        }
```

After creating the set of viterbimessages, I looped through the set op probabilities for the optimal paths and found that was most likely. From there, I backchained to find the full path.

The results of this method were quite successful. For example, unlike forward-oriented filtering, the second state in the sample problem was correctly guessed. Further, the optimal path was found with probability 0.841, a higher probability than that found for the most-likely state in either filtering or smoothing.