Hidden Markov Model

Barry Yang
CS 76: Artificial Intelligence

January 25, 2018

To Run

Unarchive desired .zip file and place files into navigable folder. Open terminal and `cd` to folder with files. Run `python3 sensor_problem.py` to view the solution output. The solution will output a series of matrices, with each element in the matrix corresponding to the probability the robot is in that particular location. For convenience, the correct route as well as the robot's sensed colors are also printed. Because the sensor readings are randomly, each run will be different.

To change the maze parameter, navigate to `mazes/maze0.maz` to change the maze. Capital letters "R," "G," "B," "Y" must be used to denote a floor. All other maze building constraints are retained from lab 2. To change the timestep parameter, open up `python3 sensor_problem.py` and navigate to the bottom of the file. Under `main`, change the "5" in `test_problem = sensor_problem(test_maze, 5)` to whatever timestep is desired.

Literature Review

For an extension, I read Lawrence R. Rabiner's paper on speech recognition[1] in order to identify an application of hidden Markov models. The paper addressed the problem of speech recognition given interfering signals.

In general, the authors sought to learn how a better sensor could be produced. The paper examined signal decomposition as an efficient technique to decompose simultaneous signals. They discussed two school of ideas: the first is that speech recognition techniques often involve pre-processing based on signals and the second is that the signal and the noise could be processed simultaneously via decomposition. The idea is a particular sensor could be broken down simultaneously and combined to form a coherent signal or observation probability = $P(\text{observation} M1 \veebar M2)$ where $\veebar$ is any operation. Ultimately, the authors argued that the latter method is the superior method, as a HMM can dynamically model varying signals.

The components of the signal are speech models of words and a concurrent set of noise (the author used pink and white noise as treatment for an experiment). They also combined the signal with multiple signal-noise-ratios to

produce different treatments. They method they produced signals is to pass a stream of speech through a filter which produces background noise, and analyzed the signal based on their coherence.

I liked this paper, but thought that their experimental methods were too "sanitized." In real life, noise is seldom as constant as white noise. Nevertheless, their experimental results proved superior to that of other speech recognition noise interference techniques, so it has been important to make speech recognition the power it is today.

Introduction

In the HMM assignment, a robot is in a maze with the same physics as the mazeworld assignment. The robot does not know where it is, but can use a sensor (with chance of error) to read the color it is standing on. Given a list of sensor readings, we can figure out a probability distribution of where the robot might be using a hidden Markov model.

The structure of my solution is composed of a `sensor_problem` class which uses a `robot` object (in addition, the `Maze` class is modified slightly to support colored tiles). The robot class's purpose is to create a sequence of plausible sensor readings, given the robot's initial location, sensor errors, and geography of the maze. The `sensor_problem` class uses the sequence given by the robot in order to create probability distributions of where the robot may be for each location in the maze for a given timestep. These classes are further discussed in detail in the following sections.

For my solution, I implemented the forward-filtering method found on page 579 from the textbook.

Robot

As stated before, the `Robot` class produces a sequence of plausible sensor readings. The primary method is `make_readings()` which produces this sequence to be used by the `sensor_problem` class. In this class, the robot class keeps track of the sequence of "real" locations, in order to simulate erred readings.

`make_readings()` adds a sensed color as long as the time step runs. For each timestep, `sense_color()` runs which appends the sensed colors to the readings list. This method picks a random number from 0 to 1 and compares it against a threshold, which is set to .88. If the random value is less than .88, then the "sensor" gave a valid reading, and the real color is appended. Otherwise, if the random value is over the threshold, then a random choice is returned out of the three remaining "wrong" colors. This is done to simulate a wrong reading.

After the color is appended, the robot makes a random move in one of the four cardinal directions. However, if a move is blocked (because of a wall or a border) then the robot stays in place. This process is done by the method

`move()` which updates the robot's current location. `move()` picks a random move from `get_successors()`, which returns a list of hypothetical moves by a robot. The array returned by `get_successors()` will always be size four, as each index represents a movement in one of the four cardinal directions.

Maze

Small modifications were made to the maze file.

```
R#GB
B#YG
GBRY
YGBR
```

Floors for mazes are now valid via the letters "R," "G," "B," "Y". # signs and syntax for defining a robot's starting location remains the same.

sensor_problem

The `sensor_problem` class takes in the sequence of colors produced by the robot and, with knowledge about the layout of the maze, produces a series of matrices that display the probabilistic distribution for each location.

Theory

This problem uses the hidden Markov model in order to build the probabilistic distribution for each location of the maze for a given timestep. For a 4×4 maze, the state of the robot is a single random variable X, which can be 16 different values (the probability that the robot is in that location of the maze, out of 16 possible locations). For example, X_1 represents the probability that the robot is located at the first state.

This problem makes use of two models in order to build the probabilistic solution. The first model is the transition model T which is produced from the maze's layout. Specifically, T determines the probability distribution of the current state variables given the past state variables. A cell in T can be represented as $P(X_j|X_i)$, or the probability of X_j given X_i. For a 4×4 maze, T is 16×16 as there are 16 possible values that could turn into 16 other possible values (for this question, many cells will be 0 as many cells are not directly adjacent to each other).

The other model used is the sensor model O. O determines how likely evidence at time t, or e_t, appears given a state $X_t = i$. This can be be represented as $P(e_t|X_t = i)$, or the probability of e_t given $X_t = i$. O is diagonal matrix, where values in a diagonal represent the "evidence" displayed in state i. For example, in the sensor problem, which reads correct colors .88 percent of the time and wrong colors .04 of the time for other wrong colors, if state 1 is on a green tile, then $P(e_t|X_t = 1) = 0.88$ and other values that are not green would have a probability of 0.04. Thus, for the sensor model, four different matrices are needed in order to identify a correct color.

These two models could be used to compute the probabilities. From the textbook, this is the act of filtering, or the task of completing the belief state given all evidence to date (this is also known as state estimation). I use the forward equation, described as $f_{1:t+1} = \alpha\text{FORWARD}(f_{1:t+1}, e_{t+1})$, where FORWARD implements the update. . From the textbook, $f_{1:t}$ can be thought of as a "message" that is propagated forward along the sequence, modified by each transition and updated by each new observation. The forward equation for a HMM is specified by $f_{1:t+1} = \alpha O_{t+1} T^T f_{1:t}$, where O_{t+1} is the observation for the state, T^T is the transpose of the transition matrix, and $f_{1:t}$ all previous updates. For the first iteration of the sensor robot, f_0 is evenly distributed along all possible legal state values (to reflect that the robot currently does not initially know where it is).

Code

Again, `sensor_problem` takes in a path given by the robot and displays the distribution of probabilities and renders the output.

Prior to any calculation, a state map is created which maps an x-y coordinate to state (represented by number) and a boolean value. This boolean value is "True" if a wall is found at the state or "False" if no wall is found at a state. These states represent the probability the robot is located at the x-y coordinate for a given timestep. Thus, states that have a wall (marked as True) automatically are given a probability of 0 as a robot can never be occupying a space held by a wall.

Transition Matrix Implementation

The transition matrix is represented by a 2D-list which is then converted to a matrix for matrix multiplication. The theory behind the transition matrix is discussed in the "Theory" section of the paper. To create a transition matrix for a 4×4 maze, a 16×16 matrix is initialized. For each possible coordinate, the program gets the state and the wall value. If there is no wall at the location, then a list of legal moves is found for that location (the legal moves range from 1 to 5—the robot is always able to stay at its current location, or can move north, south, east, or west if there is no obstacle).

The number of legal moves can be used to find $T_{i,i}$, or the probability that the robot stays in its current location. This probability is (5−number of legal moves)/4. This is because if there are five possible legal moves, then the robot will definitely move from its current location, so the probability is 0. If there are 4 possible legal moves, that means there is one wall, so the robot has three locations it can move to, or one location it can stay in—in this case the probability of staying is .25. Similarly, for three walls, the robot can only move away from its location if it chooses that direction, so there is a .75 probability it says (the robot will stay in the same location three out of four movements).

The program then iterates through the legal moves for that location to fill in $T_{i,j}$, where i and j are adjacent. For each legal move, unless the states are the

same, the probability is always 0.25, as a movement into a location is always chosen out of four possible movements.

After the transition matrix is filled, the 2D-list is converted to a matrix and the transpose of that matrix is returned (as the transpose of the transition matrix is used in the forward location).

Sensor Matrix Implementation

A dictionary of matrices is used to store sensor matrices, where the key is a color mapping to the color's particular sensor matrix. The sensor matrices can be generated in the beginning of the program prior to any calculation as the tiles of the matrices do not change color as time progresses. For four colors, there are four matrices that need to be stored.

Each matrix is created in the same way as the transition matrix—a 16×16 matrix is initialized with every element at 0, and each state is examined. Because the matrix is a diagonal matrix, only values $T_{i,i}$ (where i ranges across all the states) are filled in as non-zero values. If the state is a wall, then that state is ignored. Let us assume that the program is currently building the matrix for the color green. If the state is not a wall, then the color of the state is examined. If the state is green then $T_{i,i}$ is set to 0.88 (the error threshold). This is to reflect that if the state is green, then the "evidence" (sensed color value) has an 0.88 chance of being correct. The remaining .12 error is split among the other three colors. So, if the state is not green, then $T_{i,i}$ is set to 0.04. Once the matrix finishes building, it is converted to a matrix and added to the dictionary.

Filtering

After the transition and sensor matrices are built and a path is given, the sensor problem can be solved. The `solve` method creates a starting distribution and runs the distribution through the filter method in order to produce a solution (a list of probability distributions). The list of solutions is then rendered to a readable format (which is commented in the code).

`starting_distribution()` creates a 1×16 matrix which represents all the possible initial probabilities for every starting space. This is 1/number of legal starting states. If there are no walls, the number of legal starting states is the number of starting states. However, because a robot can never occupy a wall, the probability at that space is always 0. Thus, for 16 possible starting spaces and 1 wall, the initial probability for every position is $1/(16-1)$.

The formula for filtering is $f_{1:t+1} = \alpha O_{t+1} T^T f_{1:t}$. Essentially, this means that the vector of starting locations should be multiplied by the transpose of the transition matrix and then the appropriate sensor matrix, given the current timestep. This result is then repeated for the next color in the path, with the vector of starting locations replaced by the last result. For each step, the distribution is stored in a list of solutions to be rendered.

Experimentation

In general, the probability distributions are accurate given a random path. Below is an example of one run of `python3 sensor_problem.py` with no mistakes in the sensor. However, the program does become confused when plausible routes can be made (because of adjacent colors) despite an incorrect real route.

```
Barrys-MacBook-Pro:lab6 barryyang$ python3 sensor_problem.py
Maze:
      R#GB
      B#YG
      GBRY
      Y1BR

sensed path: ['G', 'B', 'R', 'Y', 'G']
correct path: [(1, 0), (2, 0), (2, 1), (2, 2), (3, 2)]
no mistakes in sensors

 t = 0 (starting values)
3 |R: 0.07143||W: 0.00000||G: 0.07143||B: 0.07143|
2 |B: 0.07143||W: 0.00000||Y: 0.07143||G: 0.07143|
1 |G: 0.07143||B: 0.07143||R: 0.07143||Y: 0.07143|
0 |Y: 0.07143||G: 0.07143||B: 0.07143||R: 0.07143|
       0           1           2           3

 t = 1 (robot begins sensing)
3 |R: 0.02267||W: 0.00000||G: 0.49871||B: 0.02267|
2 |B: 0.02267||W: 0.00000||Y: 0.02267||G: 0.49871|
1 |G: 0.49871||B: 0.02267||R: 0.02267||Y: 0.02267|
0 |Y: 0.02267||G: 0.49871||B: 0.02267||R: 0.02267|
       0           1           2           3

 t = 2
3 |R: 0.00245||W: 0.00000||G: 0.02819||B: 0.62024|
2 |B: 0.33709||W: 0.00000||Y: 0.02819||G: 0.01532|
1 |G: 0.01532||B: 0.62024||R: 0.00245||Y: 0.01532|
0 |Y: 0.02819||G: 0.01532||B: 0.33709||R: 0.00245|
       0           1           2           3

 t = 3
3 |R: 0.30689||W: 0.00000||G: 0.02854||B: 0.05200|
2 |B: 0.02802||W: 0.00000||Y: 0.00300||G: 0.02750|
1 |G: 0.04053||B: 0.02646||R: 0.89173||Y: 0.00144|
0 |Y: 0.00352||G: 0.04053||B: 0.01447||R: 0.31836|
       0           1           2           3

 t = 4
3 |R: 0.02752||W: 0.00000||G: 0.00325||B: 0.00464|
2 |B: 0.01170||W: 0.00000||Y: 0.60675||G: 0.00244|
1 |G: 0.00286||B: 0.02899||R: 0.00132||Y: 0.79070|
0 |Y: 0.05623||G: 0.00247||B: 0.03670||R: 0.01893|
       0           1           2           3

 t = 5
3 |R: 0.00277||W: 0.00000||G: 0.39984||B: 0.00044|
2 |B: 0.00158||W: 0.00000||Y: 0.01805||G: 0.90888|
1 |G: 0.06457||B: 0.00105||R: 0.04304||Y: 0.02392|
0 |Y: 0.00346||G: 0.08049||B: 0.00175||R: 0.02545|
       0           1           2           3
```

Below is an example of a sensor with a mistake. As expected, the accuracy of the probability distributions decrease, but do remain fairly accurate as more distributions are created.

```
[Barrys-MacBook-Pro:lab6 barryyang$ python3 sensor_problem.py
Maze (robot labeled as 1):
      R#GB
      B#YG
      GBRY
      Y1BR

sensed path: ['G', 'R', 'B', 'G', 'G']
correct path: [(1, 0), (1, 0), (1, 1), (0, 1), (0, 1)]
mistakes:
     Wrong color sensed at (1, 0)

 t = 0 (starting values)
3 |R: 0.07143||W: 0.00000||G: 0.07143||B: 0.07143|
2 |B: 0.07143||W: 0.00000||Y: 0.07143||G: 0.07143|
1 |G: 0.07143||B: 0.07143||R: 0.07143||Y: 0.07143|
0 |Y: 0.07143||G: 0.07143||B: 0.07143||R: 0.07143|
       0           1           2           3

 t = 1 (robot begins sensing)
3 |R: 0.02267||W: 0.00000||G: 0.49871||B: 0.02267|
2 |B: 0.02267||W: 0.00000||Y: 0.02267||G: 0.49871|
1 |G: 0.49871||B: 0.02267||R: 0.02267||Y: 0.02267|
0 |Y: 0.02267||G: 0.49871||B: 0.02267||R: 0.02267|
       0           1           2           3

 t = 2
3 |R: 0.45405||W: 0.00000||G: 0.23735||B: 0.23735|
2 |B: 0.12899||W: 0.00000||Y: 0.23735||G: 0.12899|
1 |G: 0.12899||B: 0.23735||R: 0.45405||Y: 0.12899|
0 |Y: 0.23735||G: 0.12899||B: 0.12899||R: 0.45405|
       0           1           2           3

 t = 3
3 |R: 0.03526||W: 0.00000||G: 0.02245||B: 0.43749|
2 |B: 0.43749||W: 0.00000||Y: 0.02501||G: 0.01732|
1 |G: 0.01732||B: 0.49385||R: 0.01732||Y: 0.02757|
0 |Y: 0.01732||G: 0.01732||B: 0.60657||R: 0.02757|
       0           1           2           3

 t = 4
3 |R: 0.01490||W: 0.00000||G: 0.30620||B: 0.02509|
2 |B: 0.02544||W: 0.00000||Y: 0.00225||G: 0.30620|
1 |G: 0.58295||B: 0.01497||R: 0.03163||Y: 0.00246|
0 |Y: 0.00190||G: 0.68499||B: 0.01835||R: 0.01891|
       0           1           2           3

 t = 5
3 |R: 0.00265||W: 0.00000||G: 0.53259||B: 0.02507|
2 |B: 0.02455||W: 0.00000||Y: 0.02446||G: 0.27973|
1 |G: 0.52054||B: 0.04974||R: 0.00144||Y: 0.01359|
0 |Y: 0.04812||G: 0.59958||B: 0.02853||R: 0.00222|
       0           1           2           3
```

References

[1] Varga, A. P. and Moore, R. K. *Hidden Markov model decomposition of speech and noise." Acoustics, Speech, and Signal Processing.* Acoustics, Speech and Signal Processing, 1990.