EDA132, Assignment 2: Probabilistic reasoning

Kristoffer Lundgren
kem01klu@student.lu.se
810622-2717

Axel Nyström
mat04any@student.lu.se
850825-8590

May 12, 2016

1 Introduction

This assignment in the course "Applied Artificial Intelligence, EDA132" at LTH, Sweden, during the spring semester of 2016 is about probabilistic reasoning, where the position of a simulated robot is to be estimated through inference from a temporal model. Using a hidden Markov model together with a simulated noisy sensor it is possible through filtering to get the best possible estimate of the current position of the robot.

2 Theory

The probabilities that are interesting to our problem consist of the probabilities for the entire state space, given all evidence up to and including this point, where some sort of causality is assumed between transitions between two states, that is

$$\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) = f(\mathbf{e}_{t+1}, \mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t})), \tag{1}$$

which defines a recursive relation for the transitions between two states. By dividing up the evidence, using Bayes' rule as well as the Markov assumption[1] we get

$$\begin{aligned}\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) &= \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}, \mathbf{e}_{t+1}) \\ &= \alpha\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1}, \mathbf{e}_{1:t})\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}) \\ &= \alpha\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1})\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t}).\end{aligned} \tag{2}$$

By conditioning on the current state $\mathbf{X}_t$, it is possible to obtain a one-step prediction:

$$\begin{aligned}\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1}) &= \alpha\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1}) \sum_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t, \mathbf{e}_{1:t}) P(\mathbf{x}_t|\mathbf{e}_{1:t}) \\ &= \alpha\mathbf{P}(\mathbf{e}_{t+1}|\mathbf{X}_{t+1}) \sum_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t) P(\mathbf{x}_t|\mathbf{e}_{1:t}).\end{aligned} \tag{3}$$

[1] That is, the probabilities for a given state depends on only the previous state, $\mathbf{P}(\mathbf{X}_t|\mathbf{X}_{0:t-1}) = \mathbf{P}(\mathbf{X}_t|\mathbf{X}_{t-1})$.

Thinking of the filtered estimate $\mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t})$ as a message $\mathbf{f}_{1:t}$, equation (3) can now instead be written as

$$\mathbf{f}_{1:t+1} = \alpha \text{Forward}(\mathbf{f}_{1:t}, \mathbf{e}_{t+1}) \tag{4}$$

where the process begins with $\mathbf{f}_{1:0} = \mathbf{P}(\mathbf{X}_0)$.

If we introduce two matrices, $\mathbf{T}_{ij} = P(X_t = j|X_{t-1} = i)$ and $\mathbf{O}_t$ whose diagonal consists of $P(e_t|X_t = i)$, equation (4) can now be implemented as

$$\mathbf{f}_{1:t+1} = \alpha \mathbf{O}_{t+1} \mathbf{T}^T \mathbf{f}_{1:t}. \tag{5}$$

3 The model

The robot moves on an $m * n$ rectangular grid, taking one step at a time in whatever direction it is facing. If it's not possible to move forward (ie if the robot is blocked by a wall), it will change it's heading to a new random heading such that it's path is clear. There is also a 30% chance that the robot changes it's direction even if it's not being blocked by a wall.

The sensor readings are perturbed by random noise: with a probability of 10% it reports the correct location of the robot, with 40% probability it reports a location whose max-norm distance to the correct location is 1 and with 40% probability it reports a location at a distance of 2. With 10% probability the sensor fails to produce a reading at all. In our interpretation and subsequent implementation of the model, a sensor reading outside the bounds of the room will also be reported as a failed reading, so if the robot is close to the edges or corners of the room, the sensor is more likely to fail.

4 Implementation

The program in the form of a runnable .jar-file as well as the source code can be found at `https://github.com/Tipulidae/RobotLocalisation`. To run the program, type "java -jar robot w h", where the optional parameters w and h represents the width and height of the room, respectively (default value is 10).

EstimatorInterface.java Interface that supplies the necessary methods for the RobotLocalizationViewer class to work. Implemented in `RobotCommandCenter.java`. This interface was supplied as part of the assignment.

LocalizationDriver.java Class supplied as part of the assignment, drives the updates of the RobotLocalizationViewer class.

Main.java Class supplied as part of the assigment, which constitutes the main program that runs the robot simulation.

Position.java Describes a position as an x,y-tuple. Contains some helper methods for computing the maxnorm and taxinorm, and some others. Also contains a custom NULL-position, for convenience.

Heading.java constitutes an enum which allows us to represent headings as either NORTH, EAST, SOUTH or WEST.

Robot.java describes the actual robot and its methods, which includes walking, changing heading as well as getting a reading of the environment through the sensor.

RobotCommandCenter.java implements the EstimatorInterface and is mostly responsible for filtering the data into the requested format by the viewer from our internal representations.

Room.java describes the room and contains methods for answering various questions about the room, such as for example whether the robot is facing a wall, if it is inside the room, finding neighbors of a position and finding headings which don't face a well.

Sensor.java Simulates the noisy sensor readings. Each time the robot takes a step, the sensor makes a reading simulating the sensor model.

SensorFilter.java This class is doing the actual filtering using HMM. On initialization it creates the **O** matrix for all possible sensor readings, as well as the **T** matrix describing the transition model, since those are fixed in time. When a new reading is made by the Sensor, the SensorFilter updates it's **f** vector according to the theory, which then gives us the filtered probabilities for each state. There is probably room for a bit of optimization when it comes to all the nested loops (we noted, for instance, that **T** is symmetric), but we decided to leave that for another time.

RobotLocalizationViewer.java is a Viewer supplied as part of the assignment, which uses the Swing library for drawing graphics.

5 Result

As long as the dimensions of the room is kept small, the forward filtering algorithm is giving good results. After only a very few number of steps, the localisation estimate is correct 40% of the time and within 1 robot step from the correct location more than 85% of the time. This is illustrated in fig 1.

6 Discussion

Unfortunately the time complexity for each step is $O(S^2)$ where S is the size of the state space, i.e. $4 * m * n$. The space complexity, while fixed in time, is also $O(S^2)$. This means that even if the room size grows just a little, the time necessary to perform the filtering becomes quite long, and the space needed to store the matrices easily exceeds the RAM capability of the machine. For example, 2Gb of heap memory was insufficient for Java to run the program with a room size of $80 * 80$. Clearly the model doesn't scale particularly well. For larger rooms, particle filtering could possibly be employed as an approximation.

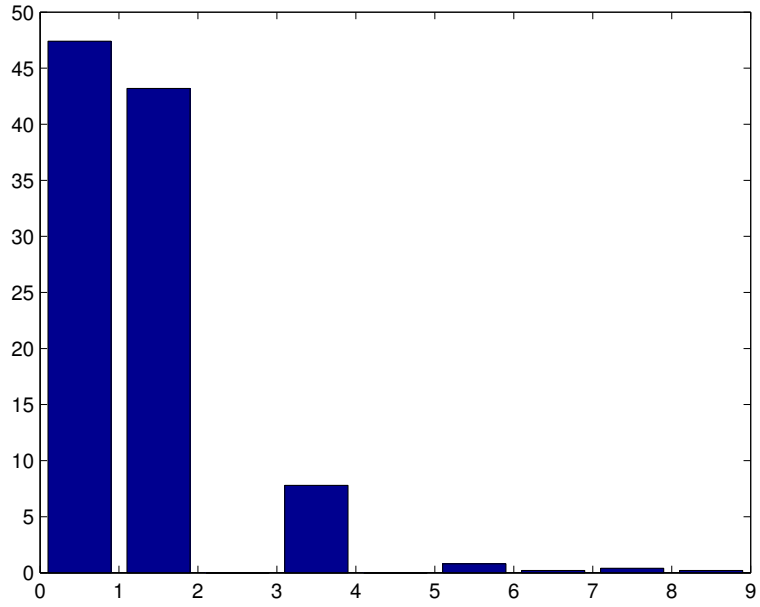

Figure 1: Histogram showing the L_1-norm distance between estimated and correct robot location. Y-axis is in %. The simulated room was 10x10 large, and the number of simulated steps was 500.