Matheus C. Fernandes
fernandes@g.harvard.edu
CS181-S17

Assignment #5
Due: 5:00pm April 14, 2016

Collaborators:
Bolei Deng
Callin Switzer
Joe Sedlak

Homework 5: Graphical Models and MDPs

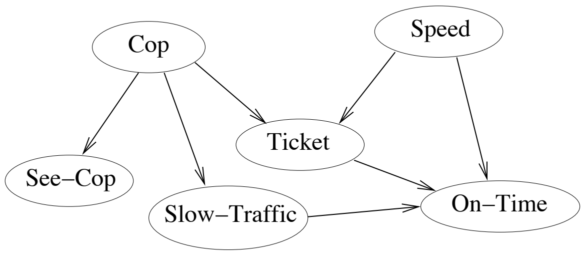
Introduction

There is a mathematical component and a programming component to this homework. Please submit your PDF and Python files to Canvas, and push all of your work to your GitHub repository. If a question requires you to make any plots, please include those in the writeup.

Bayesian Networks [7 pts]

Problem 1

In this problem we explore the conditional independence properties of a Bayesian Network. Consider the following Bayesian network representing an Uber driver taking a passenger to the airport. Each random variable is binary (true/false).



The random variables are:

- Cop: is there a State trooper present on the highway?
- See-Cop: has the driver seen the trooper?
- Slow-Traffic: is the traffic moving slowly?
- Ticket: does the driver get a speeding ticket?
- Speed: does the passenger demand that the driver speeds?
- On-Time: does the passenger get to the airport on time?

For each of these questions, use the method of d-separation, and show your working.

1. Is $I(\text{Cop}, \text{Speed})$? If NO, give intuition for why.
2. Is $I(\text{Cop}, \text{Speed} \mid \text{Ticket})$? If NO, give intuition for why.
3. Is $I(\text{See-Cop}, \text{On-Time})$? If NO, give intuition for why.
4. Is $I(\text{Cop}, \text{On-Time} \mid \text{Slow-Traffic})$? If NO, give intuition for why.
5. Modify the network to model the setting where a trooper must still be present for a driver to get a ticket, but tickets are delivered electronically and the driver does not need to stop if caught.
6. For this modified network, what are two random variables, X, such that if either was known we would have $I(\text{See-cop}, \text{On-Time} \mid X)$? Give intuition for your answer.

Solutions

Problem 1

Yes, cop is conditionally independent from speed since the only connection is speed and from d-separation we know that this setting introduces conditional independence.

Problem 2

No, because conditioning on ticket means that we have a ”explaining away” situation. Thus, there is a path from speed to cop through ticket. In other words, if we know whether there is a ticket or not, we can relate speed to whether there is a cop there. So, if the uber driver got a ticket, that means that if the cop is there the driver was very likely speeding and vice versa.

Problem 3

No, because there are two connections between cop and on-time, through ticket and slow-traffic. Here, being on time is dependent on whether the uber driver gets a ticket which ultimately depends on whether there is a cop. Likewise, being on time depends on the slow-traffic which depends on whether the cop is there on not.

Problem 4

No, if slow traffic is known, that does not block all paths from on-time to see-cop. A path still exists from on-time to ticket to cop to see-cop. Intuitively, this means that of the driver saw the cop, when a cop is there, and he does or does not get a ticket, he will either make it on time or not. Therefore, from this chain of events, there is a dependence between on-time and see-cop.

Problem 5

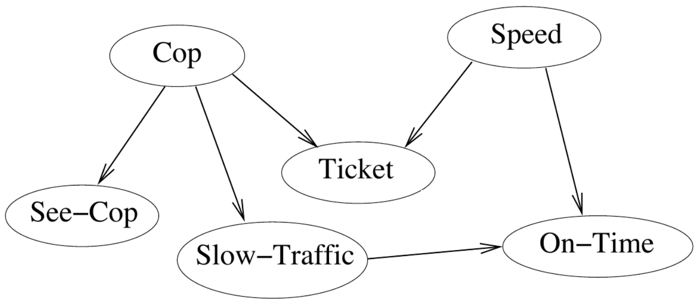


Figure 1: Modified the network to model the setting where a trooper must still be present for a driver to get a ticket, but tickets are delivered electronically and the driver does not need to stop if caught.

Problem 6

X = Slow traffic and cop

If we know there is slow traffic, whether we see cop or not traffic will dictate whether the uber is on-time or not. On the other case, if we know cop, then whether the driver sees the cop or not, it will not tell us whether the uber will be on time or not. That is because we conditionally know the cop and thus this will tell us whether there will be slow traffic ot not and will also relate whether the driver gets a ticket. This will be independent of seeing the cop, as the seeing the cop does not influence the other parts.

Hidden Markov Models [5 pts]

Problem 2

In this problem, you will derive the expressions for using the α- and β- values computed in the forward-backward algorithm on HMMs for the inference tasks of predicting the next output $\mathbf{x}_{n+1}$ in a sequence, and calculating the probability of a sequence of states $\mathbf{s}_t, \mathbf{s}_{t+1}$.

Recall that for output sequence $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_n)$, the α-values are defined, for all t, and all $\mathbf{s}_t \in \{S_k\}_{k=1}^c$, as:

$$\alpha_t(\mathbf{s}_t) = p(\mathbf{x}_1, \dots, \mathbf{x}_t, \mathbf{s}_t).$$

Similarly, the β-values are defined, for all t, and all $\mathbf{s}_t \in \{S_k\}_{k=1}^c$ as:

$$\beta_t(\mathbf{s}_t) = \begin{cases} p(\mathbf{x}_{t+1}, \dots, \mathbf{x}_n \mid \mathbf{s}_t) & \text{, if } 1 \le t < n \\ 1 & \text{otherwise.} \end{cases}$$

You will also find it useful to recall that, for all t, and all $\mathbf{s}_t \in \{S_k\}_{k=1}^c$,

$$\alpha_t(\mathbf{s}_t)\beta_t(\mathbf{s}_t) = p(\mathbf{x}_1, \dots, \mathbf{x}_n, \mathbf{s}_t).$$

1. Show that
$$p(\mathbf{x}_{n+1} \mid \mathbf{x}_1, \dots, \mathbf{x}_n) \propto \sum_{\mathbf{s}_n, \mathbf{s}_{n+1}} \alpha_n(\mathbf{s}_n) p(\mathbf{s}_{n+1} \mid \mathbf{s}_n) p(\mathbf{x}_{n+1} \mid \mathbf{s}_{n+1})$$

2. Show that
$$p(\mathbf{s}_t, \mathbf{s}_{t+1} \mid \mathbf{x}_1, \dots, \mathbf{x}_n) \propto \alpha_t(\mathbf{s}_t) p(\mathbf{s}_{t+1} \mid \mathbf{s}_t) p(\mathbf{x}_{t+1} \mid \mathbf{s}_{t+1}) \beta_{t+1}(\mathbf{s}_{t+1})$$

Solutions

Problem 1

For this part we take the definition above and use this as our beginning.

$$\begin{aligned} p(\mathbf{x}_{n+1}|\mathbf{x}_1,...,\mathbf{x}_n) &= \sum_{\mathbf{s}_n,\mathbf{s}_{n+1}} p(\mathbf{x}_{n+1},\mathbf{s}_n,\mathbf{s}_{n+1}|\mathbf{x}_1,...,\mathbf{x}_n) \\ &= \sum_{\mathbf{s}_n,\mathbf{s}_{n+1}} p(\mathbf{x}_{n+1}|\mathbf{s}_{n+1},\mathbf{s}_n,\mathbf{x}_1,...,\mathbf{x}_n)p(\mathbf{s}_{n+1}|\mathbf{s}_n,\mathbf{x}_1,...,\mathbf{x}_n)p(\mathbf{s}_n|\mathbf{x}_1,...,\mathbf{x}_n) \end{aligned}$$

For the first term, because we know that $\mathbf{x}_{n+1}$ only depends on $\mathbf{s}_{n+1}$, we can rewrite it as

$$= \sum_{\mathbf{s}_n,\mathbf{s}_{n+1}} p(\mathbf{x}_{n+1}|\mathbf{s}_{n+1})p(\mathbf{s}_{n+1}|\mathbf{s}_n,\mathbf{x}_1,...,\mathbf{x}_n)p(\mathbf{s}_n|\mathbf{x}_1,...,\mathbf{x}_n)$$

Likewise, given the markovian assumption, $\mathbf{s}_{n+1}$ only depends on $\mathbf{s}_n$

$$\begin{aligned} &= \sum_{\mathbf{s}_n,\mathbf{s}_{n+1}} p(\mathbf{x}_{n+1}|\mathbf{s}_{n+1})p(\mathbf{s}_{n+1}|\mathbf{s}_n)p(\mathbf{s}_n|\mathbf{x}_1,...,\mathbf{x}_n) \\ &= \sum_{\mathbf{s}_n,\mathbf{s}_{n+1}} p(\mathbf{x}_{n+1}|\mathbf{s}_{n+1})p(\mathbf{s}_{n+1}|\mathbf{s}_n)\frac{p(\mathbf{s}_n,\mathbf{x}_1,...,\mathbf{x}_n)}{\underbrace{p(\mathbf{x}_1,...,\mathbf{x}_n)}_{\text{constant}}} \\ &\propto \sum_{\mathbf{s}_n,\mathbf{s}_{n+1}} p(\mathbf{x}_{n+1}|\mathbf{s}_{n+1})p(\mathbf{s}_{n+1}|\mathbf{s}_n)p(\mathbf{s}_n,\mathbf{x}_1,...,\mathbf{x}_n) \\ &= \sum_{\mathbf{s}_n,\mathbf{s}_{n+1}} p(\mathbf{x}_{n+1}|\mathbf{s}_{n+1})p(\mathbf{s}_{n+1}|\mathbf{s}_n)\alpha_n(\mathbf{s}_n) \end{aligned}$$

■

Problem 2

$$\begin{aligned} p(\mathbf{s}_t,\mathbf{s}_{t+1}|\mathbf{x}_1,...,\mathbf{x}_n) =& \frac{p(\mathbf{s}_t,\mathbf{s}_{t+1},\mathbf{x}_1,...,\mathbf{x}_n)}{\underbrace{p(\mathbf{x}_1,...,\mathbf{x}_n)}_{\text{constant}}} \\ \propto& p(\mathbf{s}_t,\mathbf{s}_{t+1},\mathbf{x}_1,...,\mathbf{x}_n) \\ =& p(\mathbf{x}_1,...,\mathbf{x}_n|\mathbf{s}_t,\mathbf{s}_{t+1})p(\mathbf{s}_t,\mathbf{s}_{t+1}) \end{aligned}$$

Here, we can subdivide the above expression into the parts that matter, as some conditional probabilities will be 0, give the Markovian assumptions. Thus, we know the independences of the different states from the other variables except for the previous states, therefore we get the following breakdown:

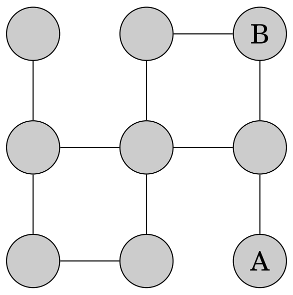
$$\begin{aligned} p(\mathbf{x}_1,...,\mathbf{x}_n|\mathbf{s}_t,\mathbf{s}_{t+1})p(\mathbf{s}_t,\mathbf{s}_{t+1}) =& p(\mathbf{x}_1,...,\mathbf{x}_t|\mathbf{s}_t)p(\mathbf{x}_{t+1}|\mathbf{s}_t)p(\mathbf{x}_{t+2},...,\mathbf{x}_n|\mathbf{s}_{t+1})p(\mathbf{s}_t,\mathbf{s}_{t+1}) \\ =& p(\mathbf{x}_1,...,\mathbf{x}_t|\mathbf{s}_t)p(\mathbf{x}_{t+1}|\mathbf{s}_t)p(\mathbf{x}_{t+2},...,\mathbf{x}_n|\mathbf{s}_{t+1})p(\mathbf{s}_{t+1}|\mathbf{s}_t)p(\mathbf{s}_t) \\ =& p(\mathbf{x}_1,...,\mathbf{x}_t|\mathbf{s}_t)p(\mathbf{s}_t)p(\mathbf{x}_{t+1}|\mathbf{s}_t)p(\mathbf{x}_{t+2},...,\mathbf{x}_n|\mathbf{s}_{t+1})p(\mathbf{s}_{t+1}|\mathbf{s}_t) \\ =& p(\mathbf{x}_1,...,\mathbf{x}_t,\mathbf{s}_t)p(\mathbf{x}_{t+1}|\mathbf{s}_t)p(\mathbf{x}_{t+2},...,\mathbf{x}_n|\mathbf{s}_{t+1})p(\mathbf{s}_{t+1}|\mathbf{s}_t) \\ =& p(\mathbf{x}_1,...,\mathbf{x}_t,\mathbf{s}_t)p(\mathbf{s}_{t+1}|\mathbf{s}_t)p(\mathbf{x}_{t+1}|\mathbf{s}_t)p(\mathbf{x}_{t+2},...,\mathbf{x}_n|\mathbf{s}_{t+1}) \\ =& \alpha_t(\mathbf{s}_t)p(\mathbf{s}_{t+1}|\mathbf{s}_t)p(\mathbf{x}_{t+1}|\mathbf{s}_t)\beta_{t+1}(\mathbf{s}_{t+1}) \end{aligned}$$

■

Markov Decision Processes [7 pts]

Problem 3

In this problem we will explore the calculation of the *MDP value function V* in a 2D exploration setting, without time discounting and for a finite time horizon. Consider a robot navigating the following grid:



The robot moves in exactly one direction each turn (and must move). The robot's goal is to maximize its score in the game after T steps. The score is defined in the following way:

- If the robot attempts to move off the grid, then the robot loses a point (-1) and stays where it is.
- If the robot moves onto node A, it receives 10 points, and if it moves onto node B, it receives 5 points. Otherwise, it receives 0 points.

1. Model this as a Markov decision process: define the states S, actions A, reward function $r : S \times A \mapsto \mathbb{R}$, and transition model $p(s' \mid s, a)$ for $s', s \in S$ and $a \in A$.
2. Consider a *random policy* π, where in each state the robot moves uniformly at randomly in any of its available directions (including off the board). For every position on the grid calculate the value function, $V_t^\pi : S \mapsto \mathbb{R}$, under this policy, for $t = 2, 1$ steps left to go. You can find LaTeX code for the tables in the solution template. Note that you should have 2 tables, one for each time horizon.
3. Now assume that the robot plays an *optimal policy* π_t^* (for t time steps to go). Find the optimal policy in the case of a finite time horizon of $t = 1, 2$ and give the corresponding MDP value functions $V_t^* : S \mapsto \mathbb{R}$, under this optimal policy. You can indicate the optimal policy for each time horizon on the corresponding V_t^* table via arrows or words in the direction that the robot should move from that state.
4. Now consider the situation where the robot does not have complete control over its movement. In particular, when it chooses a direction, there is a 80% chance that it will go in that direction, and a 10% chance it will go in the two adjacent (90° left or 90° right) directions. Explain how this changes the elements S, A, r, and $p(s' \mid s, a)$ of the MDP model. Assume the robot uses the same policy π_t^* from the previous question (now possibly non-optimal), and write this as π_t, and tie-break in favor of N, then E, then S then W. Give the corresponding MDP value functions $V_t^\pi : S \mapsto \mathbb{R}$, for this policy in this partial control world, for $t = 2, 1$ steps left to go. Is the policy still optimal?

Solution

Part 1

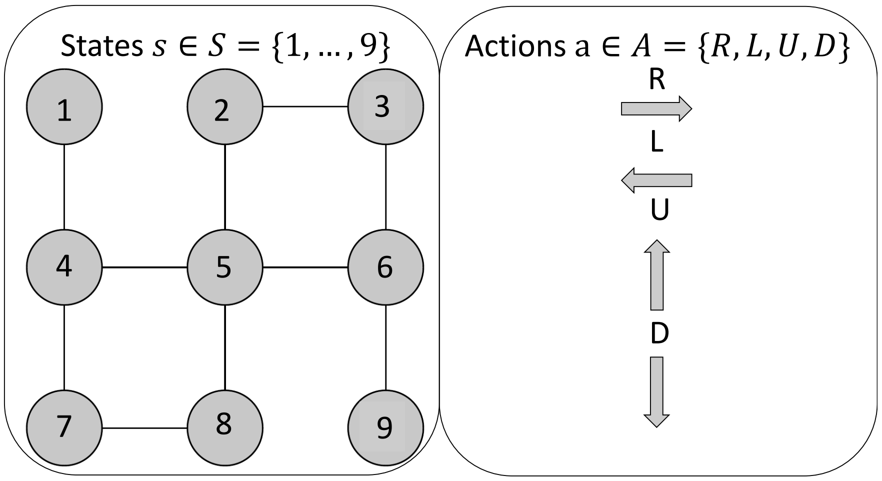

States $s \in S = \{1, \ldots, 9\}$
1
2
3
4
5
6
7
8
9
Actions a $\in A = \{R, L, U, D\}$
R
L
U
D

Figure 2: States and actions for model

Now for the reward function $r : S \times A \mapsto \mathbb{R}$,

s	$a = R$	$a = L$	$a = U$	$a = D$
1	-1	-1	-1	0
2	5	-1	-1	0
3	-1	0	-1	0
4	0	-1	0	0
5	0	0	0	0
6	-1	0	5	10
7	0	-1	0	-1
8	-1	0	0	-1
9	-1	-1	0	-1

Table 1: Reward function, $r : S \times A \mapsto \mathbb{R}$

For the transition

s, a	$s' = 1$	$s' = 2$	$s' = 3$	$s' = 4$	$s' = 5$	$s' = 6$	$s' = 7$	$s' = 8$	$s' = 9$
1,R	1	0	0	0	0	0	0	0	0
1,L	1	0	0	0	0	0	0	0	0
1,U	1	0	0	0	0	0	0	0	0
1,D	0	0	0	1	0	0	0	0	0
2,R	0	0	1	0	0	0	0	0	0
2,L	0	1	0	0	0	0	0	0	0
2,U	0	1	0	0	0	0	0	0	0
2,D	0	0	0	0	1	0	0	0	0
3,R	0	0	1	0	0	0	0	0	0
3,L	0	1	0	0	0	0	0	0	0
3,U	0	0	1	0	0	0	0	0	0
3,D	0	0	0	0	0	1	0	0	0
4,R	0	0	0	0	1	0	0	0	0
4,L	0	0	0	1	0	0	0	0	0
4,U	1	0	0	0	0	0	0	0	0
4,D	0	0	0	0	0	0	1	0	0
5,R	0	0	0	0	0	1	0	0	0
5,L	0	0	0	1	0	0	0	0	0
5,U	0	1	0	0	0	0	0	0	0
5,D	0	0	0	0	0	0	0	1	0
6,R	0	0	0	0	0	1	0	0	0
6,L	0	0	0	0	1	0	0	0	0
6,U	0	0	1	0	0	0	0	0	0
6,D	0	0	0	0	0	0	0	0	1
7,R	0	0	0	0	0	0	0	1	0
7,L	0	0	0	0	0	0	1	0	0
7,U	0	0	0	1	0	0	0	0	0
7,D	0	0	0	0	0	0	1	0	0
8,R	0	0	0	0	0	0	0	1	0
8,L	0	0	0	0	0	0	1	0	0
8,U	0	0	0	0	1	0	0	0	0
8,D	0	0	0	0	0	0	0	1	0
9,R	0	0	0	0	0	0	0	0	1
9,L	0	0	0	0	0	0	0	0	1
9,U	0	0	0	0	0	1	0	0	0
9,D	0	0	0	0	0	0	0	0	1

Table 2: Transition model $p(s'|s, a)$

Problem 2

s	$V_1^\pi(s)$
1	-3/4
2	3/4
3	-1/2
4	-1/4
5	0
6	14/4
7	-1/2
8	-1/2
9	-3/4

Table 3: Value function for t=1

s	$V_2^\pi(s)$
1	3/4(-1-3/4)+1/4(-1/4)=-1.375
2	1/2(-1+3/4)+1/4(-1/2)=-1/4
3	1/2(-1-1/2)+1/4(14/4)+1/4(3/4)=0.3125
4	1/4(-1-1/4)+1/4(-3/4)+1/4(3/4)=-0.625
5	1/4(-1/4)+1/4(3/4)+1/4(14/4)+1/4(-1/2)=0.015625
6	1/4(-1+14/4)+1/4(-1/4)+1/4(-1/2)=0.4375
7	1/2(-1-1/2)+1/4(-1/2)+1/4(-1/4)=-0.9375
8	1/2(-1-1/2)+1/4(-1/2)=-0.375
9	3/4(-1-3/4)+1/4(14/4)=-0.4375

Table 4: Value function for t=2

NOT DONE!

Problem 3

s	$V_1^*(s)$	a_1
1	0	D
2	5	R
3	0	D
4	0	R
5	0	R
6	10	D
7	0	R
8	0	U
9	0	U

Table 5: Value function under optimal policy for t=1

s	$V_2^*(s)$	a_1, a_2
1	0	D,R
2	5	R,D
3	10	D,D
4	0	R,R
5	10	R,D
6	10	D,U
7	0	R,U
8	0	U,R
9	10	U,D

Table 6: Value function under optimal policy for t=2

Problem 4

This does not change the elements of A, A, r, however it will change the probabilities of each element of $p(s'|s,a)$, such that it becomes NOT DONE!

s, a	$s' = 1$	$s' = 2$	$s' = 3$	$s' = 4$	$s' = 5$	$s' = 6$	$s' = 7$	$s' = 8$	$s' = 9$
1,R	.9	0	0	.1	0	0	0	0	0
1,L	.9	0	0	.1	0	0	0	0	0
1,U	1	0	0	0	0	0	0	0	0
1,D	.2	0	0	.8	0	0	0	0	0
2,R	0	.1	.8	0	.1	0	0	0	0
2,L	0	.9	0	0	.1	0	0	0	0
2,U	0	.9	.1	0	0	0	0	0	0
2,D	0	.1	.1	0	.8	0	0	0	0
3,R	0	0	.9	0	0	.1	0	0	0
3,L	0	.8	.1	0	0	.1	0	0	0
3,U	0	.1	.9	0	0	0	0	0	0
3,D	0	.1	.1	0	0	.8	0	0	0
4,R	.1	0	0	0	.8	0	.1	0	0
4,L	.1	0	0	.8	0	0	.1	0	0
4,U	.8	0	0	.1	.1	0	0	0	0
4,D	0	0	0	.1	.1	0	.8	0	0
5,R	0	.1	0	0	0	.8	0	.1	0
5,L	0	.2	0	.8	0	0	0	.1	0
5,U	0	.8	0	.1	0	.1	0	0	0
5,D	0	0	0	.1	0	.1	0	.8	0
6,R	0	0	.1	0	0	.8	0	0	.1
6,L	0	0	.1	0	.8	0	0	0	.1
6,U	0	0	.8	0	.1	.1	0	0	0
6,D	0	0	0	0	.1	.1	0	0	.8
7,R	0	0	0	.1	0	0	.1	.8	0
7,L	0	0	0	.1	0	0	.9	0	0
7,U	0	0	0	.8	0	0	.1	.1	0
7,D	0	0	0	0	0	0	.9	.1	0
8,R	0	0	0	0	.1	0	0	.9	0
8,L	0	0	0	0	.1	0	.8	.1	0
8,U	0	0	0	0	.8	0	.1	.8	0
8,D	0	0	0	0	0	0	.1	.9	0
9,R	0	0	0	0	0	.1	0	0	.9
9,L	0	0	0	0	0	.1	0	0	.9
9,U	0	0	0	0	0	.8	0	0	.2
9,D	0	0	0	0	0	0	0	0	1

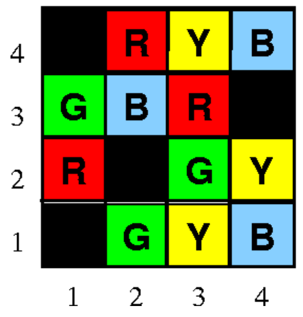
Table 7: Transition model $p(s'|s, a)$

The Viterbi Algorithm [15 pts]

Problem 4

In this problem, you will use Hidden Markov Models to reconstruct state sequences in data after learning from complete-data labeled sequences.

We consider a simple robot that moves across colored squares. At each time step, the robot attempts to move up, down, left or right, where the choice of direction is made at random. If the robot attempts to move onto a black square, or to leave the confines of its world, its action has no effect and it does not move at all. The robot can only sense the color of the square it occupies. Moreover, its sensors are only 90% accurate, meaning that 10% of the time, it perceives a random color rather than the true color of the currently occupied square. The robot begins each walk in a randomly chosen colored square.



In this problem, state refers to the location of the robot in the world in $x : y$ coordinates, and output refers to a perceived color (`r`, `g`, `b` or `y`). Thus, a typical random walk looks like this:

```
3:3 r
3:3 r
3:4 y
2:4 b
3:4 y
3:3 r
2:3 b
```

Here, the robot begins in square 3:3 perceiving red, attempts to make an illegal move (to the right), so stays in 3:3, still perceiving red. On the next step, the robot moves up to 3:4 perceiving yellow, then left to 2:4 perceiving blue (erroneously), and so on. By learning on this data, you will build an HMM model of the world. Then, given only sensor observations (i.e., a sequence of colors), the Viterbi code will re-construct an estimate of the actual path taken by the robot through its world.

The data for this problem is in `robot_no_momentum.data`, a file containing 200 training sequences (random walks) and 200 test sequences, each sequence consisting of 200 steps. We are also providing data on a variant of this problem in which the robot's actions have "momentum" meaning that, at each time step, with 85% probability, the robot continues to try to move in the direction of the last move. So, if the robot moved (successfully) to the left on the last move, then with 85% probability, it will again attempt to move left. If the robot's last action was unsuccessful, then the robot reverts to choosing an action at random. Data for this problem is in `robot_with_momentum.data`.

[Acknowledgment: thanks Rob Schapire for allowing us to use his robot dataset for this homework.]

1. **Learning a HMM model from labeled data.**

   Recall that a Hidden Markov Model is specified by three sets of probabilities: the initial probabilities of starting in each state ($\boldsymbol{\theta}$), the probabilities of transitioning between each pair of hidden states ($\mathbf{T}$), and the probabilities of each output in each state ($\boldsymbol{\pi}$). Your job will be to compute estimates of these probabilities from data. We are providing you with training data consisting of one or more sequences of state-observation pairs, i.e., sequences of the form

   $$\mathbf{s}_1, \mathbf{x}_1, \mathbf{s}_2, \mathbf{x}_2, \cdots, \mathbf{s}_n, \mathbf{x}_n$$

   For this problem, we will assume that the states are observed while training (complete data assumption). Given these sequences, you need to estimate the probabilities that define the HMM.

   Note: We will make one modification to complete-data maximum-likelihood estimation described in Lecture 17 for all parameters. Instead of estimating using MLE we will use a method known as *Laplace smoothing*, where an additional pseudo-count is added to each class. For instance, instead of using $\hat{\theta}_k = \frac{N_{1,k}}{N}$ for the estimate of the starting state, we will use $\hat{\theta}_k = \frac{N_{1,k}+1}{N+c}$, where there are c states in total. This ensures that no sequences have zero probability. (This method can also be shown to correspond to a Bayesian approach, with the pseudocount arising from a Dirichlet prior on the Categorical distribution.)

   Task: First, familiarize yourself with the provided code. Your first programming task is to fill in the `learn_from_labeled_data` function in `hmm.py`

   Testing: Check the code passes `test_learn_from_labeled_data` in `test_hmm.py`.

2. **Computing the most likely sequence of hidden states (short sequences).**

   Now that we have learned a model, we can use the Viterbi algorithm to compute the most likely sequence of hidden states given a sequence of observations.

   The second part of the dataset consists of test sequences of state-observation pairs. The Viterbi code accepts as input just the observation part of each of these sequences, and from this, will compute the most likely sequence of states to produce such an observation sequence. The state part of these sequences is provided so that you can compare the estimated state sequences generated by the algorithm to the actual state sequences. Note that these two sequences will not necessarily be identical, even for a correct implementation of the Viterbi algorithm.

   Task: Implement the Viterbi algorithm to fill in the `most_likely_states` function in `hmm.py`

   Testing: Confirm that your code passes `test_viterbi_simple_sequence` in `test_hmm.py`.

3. **Computing the most likely sequence of hidden states (long sequences).**

   In practice, the method described in the lecture notes will not actually work for longer sequences. This issue arises due to numerical underflow to to the repeated multiplication of probabilities. We can avoid this problem, by instead computing the most likely sequence in log-space, i.e.

   $$\arg\max_{\mathbf{s}_1 \dots \mathbf{s}_n} \log p(\mathbf{s}_1, \dots \mathbf{s}_n, \mathbf{x}_1, \dots, \mathbf{x}_n).$$

   Mathematically, this is the same, but will avoid the numerical issues.

   Task: Change the code in `most_likely_states` to use logs of the probabilities instead of the probabilities directly.

   Testing: Check the code now also passes `test_viterbi_long_sequence` in `test_hmm.py`.

4. **Experimenting with data.**

   Finally run `viterbi.py` on the two robot data files, `robot_no_momentum.data` and `robot_with_momentum.data` and examine the results, exploring how, why and when it works. To get the the breakdown of the error, set `debug=True` in the code. Sample command-line prompt:

```
$ python viterbi.py robot_small.data -v
```

   You should write up *briefly* what you found. Your write-up should include any observations you have made about how well HMMs work on this problem and why. Your observations should be quantitative (for instance, the number of errors was x) as well as anecdotal (situations where the approach failed).

   Although this write-up is quite open ended, **you should be sure to discuss the following**:

   - What probabilistic assumptions are we making about the nature of the data in using a hidden Markov model?
   - How well do those assumptions match the actual process generating the data?
   - And to what extent was or was not performance hurt by making such realistic or unrealistic assumptions?

Solution

Problem 1

Check!

Problem 2

Check!

Problem 3

Check!

Problem 4

The method predominantly works because, as we know, the Viterbi algorithm factors in not only the observation of the sequence but also the probability of the next step given the previous step. This can be pictured as making the best use possible of all the learning data instead of just learning how it behaves based on the observations or on the transition separately. This boils down the the Markovian assumption which dictates that we can work locally as a given state only depends on the previous state and no other state $p(s_{n+1}|s)$. This assumption, of course, introduces error to the probabilistic model of data, but more so in the case without momentum. That is because the the next step the robot will take is less predictable given the current state. The case with momentum, we see that the algorithm performs better than the case without momentum. That is because the next state more heavily depends the current state. The following is the performance breakdown:

```
No Momentum: 0.1888100 = 7524/4000 <--> 6012.332ms
With Momentum: 0.133375 = 5335/4000 <--> 6011.893ms
```

This method could work better, if there was less of a random motion between the different states. If there was a stepping mechanism that was defined more clearly than using the random or with a percent momentum then we would be able to achieve better performance. However, in general, I believe that the Markovian model is adequate for this problem, given that the sensing error will not allow the performance to greatly improve beyond using a different method that does not contain the Markovian assumption. Furthermore, if we consider that the sensors are only 90% accurate, we see that we are doing very well in comparison to even the sensor accuracy.

Problem 4 Code: The Viterbi Algorithm

```python
#!/usr/bin/env python

from util import *
from numpy import *
import numpy as np
from math import log
import copy
import sys


# Pretty printing for 1D/2D numpy arrays
MAX_PRINTING_SIZE = 30

def format_array(arr):
    s = shape(arr)
    if s[0] > MAX_PRINTING_SIZE or (len(s) == 2 and s[1] > MAX_PRINTING_SIZE):
        return "[  too many values (%s)   ]" % s

    if len(s) == 1:
        return  "[  " + (
            " ".join(["%.6f" % float(arr[i]) for i in range(s[0])])) + "  ]"
    else:
        lines = []
        for i in range(s[0]):
            lines.append("[  " + "  ".join(["%.6f" % float(arr[i,j]) for j in range(s[1])]) + "
    ]")
        return "\n".join(lines)



def format_array_print(arr):
    print format_array(arr)


def string_of_model(model, label):
    (initial, tran_model, obs_model) = model
    return """
    Model: %s
    initial:
    %s

    transition:
    %s

    observation:
    %s
    """ % (label,
            format_array(initial),
            format_array(tran_model),
            format_array(obs_model))


def check_model(model):
    """Check that things add to one as they should"""
    (initial, tran_model, obs_model) = model
    for state in range(len(initial)):
        assert((abs(sum(tran_model[state,:]) - 1)) <= 0.01)
        assert((abs(sum(obs_model[state,:]) - 1)) <= 0.01)
        assert((abs(sum(initial) - 1)) <= 0.01)


def print_model(model, label):
    check_model(model)
    print string_of_model(model, label)

def max_delta(model, new_model):
    """Return the largest difference between any two corresponding
    values in the models"""
    return max( [(abs(model[i] - new_model[i])).max() for i in range(len(model))] )


class HMM:
    """ HMM Class that defines the parameters for HMM """
    def __init__(self, states, outputs):
        """If the hmm is going to be trained from data with labeled states,
        states should be a list of the state names.  If the HMM is
        going to trained using EM, states can just be range(num_states)."""
        self.states = states
        self.outputs = outputs
        n_s = len(states)
        n_o = len(outputs)
        self.num_states = n_s
        self.num_outputs = n_o
        self.initial = zeros(n_s)
        self.transition = zeros([n_s,n_s])
        self.observation = zeros([n_s, n_o])

    def set_hidden_model(self, init, trans, observ):
        """ Debugging function: set the model parameters explicitly """
        self.num_states = len(init)
        self.num_outputs = len(observ[0])
        self.initial = array(init)
        self.transition = array(trans)
        self.observation = array(observ)

    def get_model(self):
        return (self.initial, self.transition, self.observation)

    def compute_logs(self):
        """Compute and store the logs of the model (helper)"""
        raise Exception("Not implemented")

    def __repr__(self):
        return """states = %s
        observations = %s
        %s
        """ % (" ".join(array_to_string(self.states)),
               " ".join(array_to_string(self.outputs)),
               string_of_model((self.initial, self.transition, self.observation), ""))


    # declare the @ decorator just before the function, invokes print_timing()
    @print_timing
    def learn_from_labeled_data(self, state_seqs, obs_seqs):
        """
        Learn the parameters given state and observations sequences.
        The ordering of states in states[i][j] must correspond with observations[i][j].
        Use Laplacian smoothing to avoid zero probabilities.
        Implement for (a).
        """

        num_states=self.num_states

        theta=ones((num_states))
        for i in range(len(state_seqs)):
            theta[state_seqs[i][0]]+=1
        theta=theta/sum(theta)
        self.initial=theta

        t=ones((num_states,num_states))
        for i in range(len(state_seqs)):
            for k in range(len(state_seqs[i])-1):
                t[state_seqs[i][k],state_seqs[i][k+1]]+=1
        for s in range(num_states):
            t[s,:]=t[s,:]/sum(t[s,:])
        self.transition=t

        pi=ones((num_states,num_states))
        for i in range(len(state_seqs)):
            for k in range(len(state_seqs[i])):
                pi[state_seqs[i][k],obs_seqs[i][k]]+=1

        for s in range(num_states):
            pi[s,:]=pi[s,:]/sum(pi[s,:])
        self.observation=pi

        # raise Exception("Not implemented")


    def most_likely_states(self, sequence, debug=True):
        """Return the most like sequence of states given an output sequence.
        Uses Viterbi algorithm to compute this.
        Implement for (b) and (c).
        """
        #this is being coded as according to the wikipedia page and notation
        #https://en.wikipedia.org/wiki/Viterbi_algorithm

        pi=self.initial
        A=self.transition
        B=self.observation

        num_states=self.num_states
        num_sequence=len(sequence)

        #initializing the T's
        T1 = zeros((num_states, num_sequence))
        T2 = zeros((num_states, num_sequence))

        for i in range(num_states):
            T1[i,0]=np.log(pi[i])+np.log(B[i,sequence[0]])
            # T2[i,0]=0 this is implied, but not computed for efficiency
        for i in range(1,num_sequence):
            for j in range(num_states):
                T1[j,i]=np.log(B[j,sequence[i]])+max(T1[:,i-1]+np.log(A[:,j]))
                T2[j,i]=argmax(T1[:,i-1]+np.log(A[:,j]))
        zI=argmax(T1[:,-1])
        X=zeros(num_sequence)
        X[-1]=zI

        for i in range(num_sequence-1,0,-1):
            zI=T2[int(zI),i]
            X[i-1]=int(zI)

        X=list(X.astype(int))

        return X

        # raise Exception("Not implemented")

def get_wikipedia_model():
    # From the rainy/sunny example on wikipedia (viterbi page)
    hmm = HMM(['Rainy','Sunny'], ['walk','shop','clean'])
    init = [0.6, 0.4]
    trans = [[0.7,0.3], [0.4,0.6]]
    observ = [[0.1,0.4,0.5], [0.6,0.3,0.1]]
    hmm.set_hidden_model(init, trans, observ)
    return hmm

def get_toy_model():
    hmm = HMM(['h1','h2'], ['A','B'])
    init = [0.6, 0.4]
    trans = [[0.7,0.3], [0.4,0.6]]
    observ = [[0.1,0.9], [0.9,0.1]]
    hmm.set_hidden_model(init, trans, observ)
    return hmm
```

Listing 1: Code for Problem 4