

COMS W4701: Artificial Intelligence, Summer 2024

Homework 4

Instructions: Compile all written solutions for this assignment in a single, typed PDF file. Coding solutions may be directly implemented in the provided Python file(s). **Do not modify any filenames or code outside of the indicated sections.** Submit all files on Gradescope in the appropriate assignment bins, and make sure to **tag all pages for written problems**. Please be mindful of the deadline and late policy, as well as our policies on citations and academic honesty.

Problem 1: Wandering Robot (28 points)

A robot is wandering around a room with some obstacles, labeled as # in the grid below. It can occupy any of the free cells labeled with a letter, but we are uncertain about its true location and so we keep a belief distribution over its current location. At each timestep, it may either stay in its current cell, or move to an adjacent cardinal cell, all with uniform probability. The robot also makes an observation after each transition, returning what it sees in a random cardinal direction. Possibilities include observing #, “wall”, or “empty” (for a free cell).

A	B	#
C	D	E
#	F	#

We provide you with a Jupyter notebook with the transition matrix and observation probability vectors defined as NumPy arrays. We highly recommend that you complete the subproblems in the same notebook. You can convert the notebook file to a PDF and attach it to your writeup in place of showing work. Please clearly print and indicate your final answers in the outputs, and/or transcribe them to your writeup.

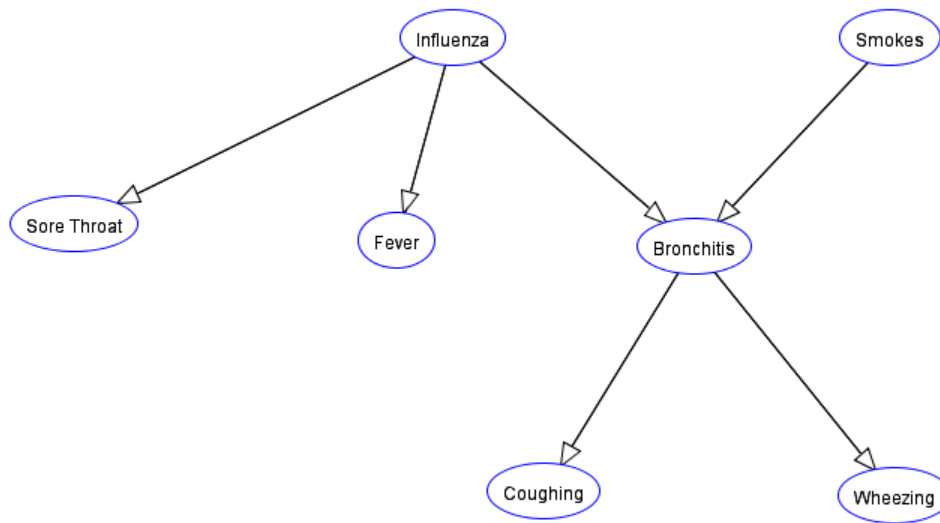
1. (4 pts) Suppose the robot wanders around for a long time without making any observations. What is the stationary distribution π over the robot’s predicted location?¹
2. (4 pts) The robot’s sensors just started working. Take the stationary distribution that you found above to be $\Pr(X_0)$. Starting from this belief state, the robot makes one transition and observation $e_1 = \text{“wall”}$, followed by a second transition and observation $e_2 = \#$. What are the belief distributions $\Pr(X_1 \mid e_1)$ and $\Pr(X_2 \mid e_1, e_2)$?
3. (4 pts) Compute the joint distribution $\Pr(X_1, X_2 \mid e_1, e_2)$. You can just list the joint probability values that are nonzero.

¹`numpy.linalg.eig` in Python returns a 1D array of eigenvalues and a 2D array where each column is a corresponding eigenvector. Remember that eigenvectors may not sum to 1 by default.

4. (4 pts) The robot suspects that the model described above is no longer accurate. We will try to update it by running through an iteration of the Baum-Welch algorithm. Compute (in order) the probability arrays β_2 , β_1 , and β_0 .
5. (4 pts) Compute the distribution of state occurrences in each timestep: γ_0 , γ_1 , γ_2 . What is the new initial state distribution $\Pr(\hat{X}_0)$?
6. (4 pts) Compute the new set of observation probabilities $\Pr(\hat{e}|X)$ for the “wall” and # observation values. Briefly explain why some state(s) may have very inaccurate or extreme estimates.
7. (4 pts) Finally, compute the two “expected” transition frequency matrices ξ_0 and ξ_1 . Use these to compute an updated model transition matrix that includes all $\Pr(X_t|\hat{X}_{t-1})$. Remember to normalize each row.

Problem 2: Diagnostic Bayes Net (28 points)

The following Bayes net is the “Simple Diagnostic Example” from the Sample Problems of the Belief and Decision Networks tool on AIspace. All variables are binary.

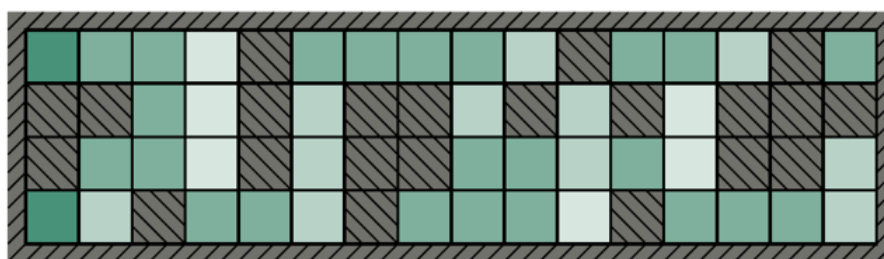


1. (6 pts) We are interested in computing the distribution $\Pr(\text{Fever} \mid \text{Wheezing} = \text{True})$. Find a minimal analytical expression for this distribution (or its unnormalized version) in terms of the Bayes net CPTs, and then rewrite each term as a factor explicitly showing the variable dependencies. What is the maximum size of the intermediate factor if all marginalization is done at the end?
2. (6 pts) We employ variable elimination to solve the query above. Consider the following two variable orderings (you may only need to sum over a subset of each):
 - (a) influenza, smokes, sore throat, fever, bronchitis, coughing, wheezing
 - (b) wheezing, coughing, bronchitis, fever, sore throat, smokes, influenza

For each of these orderings, rewrite your analytical expression by splitting up the sum over each variable, and factor out the maximal number of terms from each summation. Remember that sums are evaluated from right to left, or from inside out. What is the size of the largest intermediate factor in each?

3. (16 pts) Follow the more efficient ordering from above to numerically compute $\Pr(\text{Fever} \mid \text{Wheezing} = \text{True})$ using the applet parameters. You can use the applet to verify that this comes out to $(0.27, 0.73)$, but you should carry out the computations yourself, either manually or using Python. In addition to the final result, show each intermediate factor resulting from each variable summation step.

Problem 3: Grid World Localization (44 points)



You will be implementing grid world localization and HMM learning (see Section 14.3.2 in AIMA for a similar example). An agent is moving around and gathering observations on a grid, where each cell may be passable or not passable (e.g., a wall). We do not know exactly where the agent is, so we will have a belief state over each cell of the grid. For simplicity, the belief state will also include the blocked cells, which will always have probability 0. You will be completing the `Gridworld_HMM` class to implement the environment model, as well as filtering and smoothing algorithms.

3.1: Transition Probabilities (8 points)

The transition model is such that the agent may either stay in the current cell or move to an adjacent free cell, all with uniform probability. Thus, if the set S includes all adjacent free neighbors of state x in addition to x itself, then $\Pr(x'|x) = \frac{1}{|S|}$ for all $x' \in S$.

Write the function `initT()`, which should return a $n \times n$ NumPy array T , where n is equal to `grid.size` and $T_{ij} = \Pr(x_j|x_i)$. We recommend that you populate this array one row (state) at a time. You may use the `neighbors()` helper function, which returns a list of all adjacent free cells as well as the given cell. Be sure to check that the rows of T sum to 1.

3.2: Observation Probabilities (8 points)

From any state x , the agent can make an observation e , which is an integer value between 0 and 15 (inclusive). The 4-bit representation of e indicates whether each of the four adjacent neighbors is a blocked (1) or free (0) cell. The order of the bits is north, east, south, west (NESW). For example, a correct observation for the lower-right corner cell in the figure would be the value 6, with bit representation 0110 indicating free cells north and west and blocked cells east and south.

However, the observations are also *noisy*; there is a ϵ probability that each bit may be independently wrong. If d is the bitwise discrepancy (XOR operation) between a possible observation e and the correct observation, then $\Pr(e|x) = (1 - \epsilon)^{4-d} \epsilon^d$. Following the example above, the probability of $e = 6$ is $(1 - \epsilon)^4$, while the probabilities of $e = 0$ and $e = 15$ are both $(1 - \epsilon)^2 \epsilon^2$.

Write the function `initO()`, which should return a $16 \times N$ array O where $O_{ij} = \Pr(e = i|x_j)$. Again, it may be easiest to populate the array one row (state) at a time. First find the “correct” observation value for each state. Then for each possible observation sequence with decimal values between 0 and 15, compute the discrepancy and insert the observation probability into the array².

3.3: Filtering and Smoothing (16 points)

Now that we have the model defined, we can start performing inference. We split these tasks into four functions. `forward()` and `backward()` perform a single step of each respective algorithm, given either the `alpha` or `beta` probability message and a single `observation`. Both can be easily done in one or two lines³. Then return the updated α or β vectors.

We can then use these two methods to implement `filtering()` and `smoothing()`, each given a list of `observations` and initial belief state `init`. Each method should return two $T \times N$ NumPy arrays, where T is the number of observations and N is equal to `grid.size`. `filtering()` returns an array of α vectors and the normalized $\Pr(X_t|e_{1:t})$ belief states; `smoothing()` returns an array of β vectors and the $\Pr(X_t|e_{1:T})$ belief states.

For `filtering()`, it should be straightforward to populate the array one row at a time, each one computed by calling `forward()` on the previous row. For `smoothing()`, you will want to actually call `filtering()` first; the returned array contains all the α vectors. Then populate a β array in the same format, but going from the last row to the first and calling `backward()` for each one. The smoothed belief states can then be computed by taking the elementwise product of α with β and then normalizing each row.

3.4: Learning via Baum-Welch (6 points)

For this last part, we will no longer assume a known observation model. The agent receives a sequence of measurements, which we will use to estimate a new set of observation probabilities using the Baum-Welch algorithm in the `baum_welch()` function.

You will not need to write any code here; instead, you should carefully read the provided implementation. In your writeup, provide 1-2 sentences to comment each of the indicated code snippets (there are four snippets of two lines each). You simply need to write (in English) what the code is doing; there is no need for further explanations or derivations.

3.5: Analysis (6 points)

When you are finished, you can test your implementation by invoking `python main.py`. It can run under three “modes”, specified by the argument `-m`, as described below:

²~ in Python implements bitwise XOR.

³@ in Python implements matrix multiplication, while * implements elementwise array multiplication.

- Mode 0 (default): Run both filtering and smoothing for $-t$ timesteps (default 50), $-n$ episodes (default 500), and a pre-specified set of ϵ values. When finished, plots showing the average localization error⁴ for each algorithm will be shown.
- Mode 1: Run one filtering episode for $-t$ timesteps on an environment with ϵ equal to $-e$ (default 0). An animation will pop up showing the agent's true location and the estimated belief distribution colored on the grid over time. Bright yellow corresponds to higher probabilities.
- Mode 2: Run Baum-Welch after generating a sequence of $-t$ observations using the given $-e$ value. The learned observation probabilities are then visualized using color intensity on a 16×64 grid, and the observation log likelihood is plotted as well.

After verifying that each mode above produces sensible results with your implementation, briefly address the following questions.

1. Inspect the results of mode 0 with no additional parameters, and do the same with a similar set of ϵ values for mode 1 (e.g., 0.0, 0.1, and 0.4). How does the value of ϵ affect the performance of each inference algorithm? Explain how the animations of mode 1 verify your hypotheses.
2. Looking at mode 0, which inference algorithm generally has better localization error over time and why? Also explain the differences in the shapes of the error curves over time. Please include the two localization plots in your writeup.
3. Run mode 2 for a few different values of ϵ (e.g., 0.0, 0.1, and 0.4) and take a look at the log likelihood curves of the Baum-Welch learning experiment. Give an intuitive explanation for what this measures (do not just repeat the definition $\log \Pr(e_{1:T})$), and explain whether the shape of the curve indicates whether the learning algorithm is successful. Please include one of the log likelihood plots in your writeup.

Coding Submission

Please submit the completed `gridworld_hmm.py` file under the HW4 Coding bin on Gradescope.

⁴This is computed as the total Manhattan distance between the agent's belief distribution and its true location (a distribution with a value of 1 in its true location and 0 elsewhere).