CSCI1410 Fall 2021
Assignment 4: Hidden Markov Models

Code Submission 1 Due Tuesday, October 19 at 11:59pm ET

Code Submission 2 Due Thursday, October 21 at 11:59pm ET

Code Due Wednesday, October 27 at 11:59pm ET

Writeup Due Thursday, October 28 at 11:59pm ET

Late Day Deadline Saturday, October 30 at 11:59pm ET

Introduction

Silly Premise:

George just bought the new iOS Atari game pack! Problem is, he spilled some of his favorite beverage Dr. Pepper, soaking his entire desk along with all his papers and his smartphone. He wants to play the game, but his phone's touchscreen has started acting wonky. You will be helping George filter his touchscreen data using Hidden Markov Models (HMMs) so he can control the little Atari dude with his finger.

Hidden Markov Models:

In this assignment, we will be taking a deeper look at HMMs to better understand their functionality and uses. In the first part, you will implement a generic HMM. In the second part you will use a HMM to filter noisy touchscreen data, and in the last part you will write about what you did in the second part.

1   Part 1: Generic HMM

In this part of the assignment, you will implement a basic HMM that supports filtering and prediction. Your HMM will take in a series of observations at sequential **timesteps** (i.e. discrete points in time: 0, 1, 2, ...), and compute a probability distribution over hidden states for the current timestep (filtering) and future timesteps (prediction).

1.1   Coding

You will write your code for this part in `hmm.py`, which contains the `HMM` class. The `HMM` class is instantiated with a function specifying the sensor model, a function specifying the transition model, and a preset number of hidden states.

You will implement the following functions:

1. `tell(observation)`:
   Takes in an **observation**, records it, and processes it in any way you find convenient. You will need to

keep track of the current timestep and increment it each time your HMM is "told" a new observation (i.e. each time this function is called).

2. `ask(time)`:
   Takes in a timestep that is greater than or equal to the current timestep and outputs the probability distribution over **states** at that timestep, informed by the observations so far. The probabilities should be represented as a list in which the ith element is the probability of state i.

   `ask(0)` **should return a uniform distribution**, and the first observation that the HMM is told occurs at time 1.

   Here is an example of calling `ask` from a HMM with 3 hidden states:

   ```
   [in:] ask(4)
   [out:] [0.4, 0.2, 0.4]
   ```

   The way to interpret this is that at time 4 (i.e. after the fourth observation), the hidden state is 0 with probability 0.4, 1 with probability 0.2, and 2 with probability 0.4.

Do not modify the input and output specifications of `ask` and `tell`.

To be clear, because `tell` does not output anything, you will not be graded on what it does in isolation. You will be graded based only upon how `tell` and `ask` work together.

1.2 Representations

- States are represented as 0-indexed natural numbers $0, 1, \ldots, N-1$, where N is the number of hidden states in the HMM.

- Observations are represented as upper-case letters 'A', 'B', ..., 'Z'. For simplicity of representation, you may assume that there are no more than 26 possible observations.

- Timesteps are represented as 0-indexed natural numbers $0, 1, \ldots$.

1.3 Assumptions

In all that follows, S_t is the random variable that gives the hidden state at time t, and o_t is the observation at time t. For this part of the assignment, you may make the following assumptions:

1. The tth observation given to your HMM occurs at timestep t. In other words, observations are never skipped, and they are always told in order. You may assume this in Part 2 as well.

2. **Markov Assumption.** The future is independent of the past, given the present.

$$\Pr(S_{t+1} \mid S_t, S_{t-1}, \ldots, S_1) = \Pr(S_{t+1} \mid S_t)$$

3. The present observation is independent of past observations, given the present state.

$$\Pr(o_t \mid S_t, o_{t-1}, o_{t-2}, \ldots, o_1) = \Pr(o_t \mid S_t)$$

You can use assumptions 2 and 3 to obtain the following equation, the derivation of which you can find in section 15.2.1 of the textbook. Note that α is the normalization factor for the probability distribution.

$$\Pr(S_{t+1} \mid o_{1,2,\ldots,t+1}) = \alpha \cdot \Pr(o_{t+1} \mid S_{t+1}) \sum_{s_t} \Pr(S_{t+1} \mid S_t = s_t) \cdot \Pr(S_t = s_t \mid o_1, o_2, \ldots, o_t) \tag{1}$$

You will find this equation *extremely* helpful in completing at least Part 1 of this assignment. You should stare at it for a while to figure out what it means. If, after enough staring, you need help understanding the equation, you should come to TA hours. You should take a stab at understanding the equation's derivation as well; doing so may be helpful in Part 2.

1.4   Testing Part 1

You can run `python unit_tests.py` to test your program on a few basic IO tests. There are also tests for part 2 within this file, so don't worry about failing those for now. You should write your own additional test cases to make sure that your generic HMM is behaving as expected. You won't be graded on your tests, but testing is critical to evaluate the correctness of your code. To test your HMM, you will need a transition model and an observation model.

For this reason, we give you some sample models in `hmm_runner.py`. This file contains functions with sample data within them:

1. `sensor_model(observation, state):`
   Returns the probability of **observation** being emitted when we are in **state**.

2. `transition_model(old_state, new_state):`
   Returns the probability of transitioning from `old_state` to `new_state`.

To run your HMM, you can run the file by executing `python hmm_runner.py` and entering in observations as you see fit. It will return the estimated distribution of the current time point. Note that you will have to begin your HMM implementation before it will return useful information.

There are two implemented models for you to test on: `suppliedModel` and `simpleModel`. Each contain a relevant sensor and transition model. By default, the `suppliedModel` is used in `hmm_runner.py` when you enter observations, but it can be easily changed to use `simpleModel` by running

$$\text{python hmm\_runner.py --simple.}$$

`simpleModel` only accepts the three observations 'A', 'B' and 'C' and as a result is much easier to hand simulate. `simpleModel` is notable since the number of states and observations are explicit, and the number of states and observations are different.

In order for you to quickly verify whether your solution is correct, we have also included a compiled binary that runs the TA soution called `ta_hmm_runner.bin`. Unfortunately, this compiled binary currently only works on the **department machines**. You can run this binary with the same flags as for `hmm_runner.py` to make sure that your HMM and the solution return the same values (values within 0.001 of the TA solution are sufficient for the autograder). Here is an example that runs the TA solution on the `simpleModel` (make sure to activate the course virtual environment first!):

$$\text{./ta\_hmm\_runner.bin --simple}$$

2   Part 2: Finger Tracking

2.1   The Problem

Now you get to apply the skills you learned implementing the generic HMM to help George filter his noisy touchscreen data. You will do this by filling in the stencil code for `touchscrenHMM` class in `touchscreen.py`.

Your goal is to write `filter_noisy_data`, which takes in the touchscreen's noisy reading of the location of George's finger and outputs a probability distribution over the true location of George's finger, considering the current noisy readings as well as all past noisy readings. To compare this to Part 1, `filter_noisy_data` should behave exactly as if you were to call `tell()`, passing in the current noisy reading, and then `ask()`,

passing in the current timestep. This is the function that will be directly evaluated by the grader to determine your score in Part 2.

Note that you cannot use the compiled HMM TA solution in your handin, but you can use your implementation from part 1 if you would like.

You should write `touchscreenHMM` by formulating this situation as a HMM. This means that you should come up with both a sensor model and a transition model that can be used in conjunction with one another to filter the noisy finger data. We have provided you with stencils for `sensor_model` and `transition_model` to get you started. Since we are not grading them, you may change them in any way you like.

We will be calling the function `filter_noisy_data()` from an instance of your `touchscreenHMM`. When implemented, this function will take in a noisy frame (a NumPy array) and return the distribution of where the actual finger position is as another NumPy array. This is very similar to the `tell()` function you implemented in part 1, but in two dimensions. For example, `filter_noisy_data(frame)` could return:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & .1 & .2 \\ 0 & 0 & .1 & .5 \\ 0 & 0 & 0 & .1 \end{bmatrix}$$

When `frame` is the 2D NumPy array:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

You need to get the noisy frame by calling `get_frame()` on an instance of `touchscreenSimulator`. Every time `get_frame()` is called, the subsequent frame is returned. By calling `get_frame(actual_position=True)`, you are able to also get the actual position of the finger for testing. It will return a tuple of (`noisy_frame`, `actual_frame`).

This problem differs from a generic HMM because the finger moves in a way that depends on more than just the previous observation (the finger has directional velocity and some other interesting behaviors). Remember to take this into account in your solution! However, it is suggested to implement a regular HMM first before trying to take additional time points into account. Check out the "Hints and Observations" in section 2.4 for more on this.

It can be a bit strange to think of a distribution as a 2D array, and can be difficult transitioning between them. NumPy luckily has a library that makes it easy to convert things into 2D arrays:
https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.reshape.html

2.2 Simulator

- **Simulator**
  The simulator simulates both the movement of George's finger and the noisy readings of the touchscreen. The simulator returns a sequence of frames it has created, often producing the true position of the finger, but erring occasionally. A frame is represented as a NumPy array. Note that the actual position can only move to an adjacent square, and is more likely to continue in the same direction than change direction.

  For example, a simulation for a 4x4 board over 3 frames could look like this with the matrices repre-

senting NumPy arrays:

$$
\left[
\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},\
\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix},\
\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}
\right]
$$

For testing, the actual path of the finger is also provided. See the relevant source code section for more information on how to use the simulator.

- **Simulator Visualizer**
  A visualizer for the simulator for debugging and testing. Note that depending on the parameters the visualizer function is given, the visualizer will either display the actual position of the finger, the noisy data, or both simultaneously. See the relevant source code section for more information on how to use the visualizer.

- **Simulation Testing Manager**
  The simulation testing manager contains a few handy features to debug and test your solution. You'll be able to save simulations to test against the same data multiple times. The testing manager also builds upon the simulation visualizer to show your solution's probability distribution side-by-side with the simulation. See the next section (2.3) for more information on how to use the testing manager.

2.3   Relevant Source Code

Please note that in order to keep the scenario "realistic" we have compiled the algorithm that adds noise and the simulator so you can't see the source code. Unfortunately this means that Part 2 will only run on department machines at the moment.

- `simulator.so` This file contains the finger simulator algorithm as a compiled module. It uses a *top secret* algorithm to add noise to the actual position of the finger, so you will have to use the visualizer and/or analyze simulations to understand it better. It contains class `touchscreenSimulator`. You can make function calls to this file as if it were a `.py` file.

  To create an instance of a touchscreen simulator, you can use:

  `touchscreenSimulator(width=20, height=20, frames=100)`

  Where `width` and `height` are the dimensions of the touchscreen, and `frames` is the number of frames in the simulation. The default is a 20×20 touchscreen over 100 frames, which is what all grading scripts will be run on.

  After creating an instance of the simulator, make sure to call `sim.run_simulation()` before anything else. This will generate the data (the noisy and actual frames) within the simulation instance. Since we are trying to simulate an online algorithm where you only get data frame by frame, you can get the next frame by calling `get_frame()` on an instance of `touchscreenSimulator`. This will return a NumPy array representing a screen for the current timepoint. This function also auto increments time, so every time you call the function, the next timepoint will be returned.

  You can use the visualizer on a simulation `sim` by calling

  `sim.visualize_simulation(noisy=True, actual=True, frame_length=0.1)`

  By changing the values of `noisy` and `actual`, you can plot the noisy data, actual position, or both. If you plot both, orange represents the actual position, yellow represents the noisy position, and purple represents when the two are in the same place. The actual screens are represented as 2D NumPy arrays filled with 0s besides the touch location which is represented as a 1. Tip: check out `numpy.nonzero()`.

- `touchscreen_runner.py`: This file contains the driver script that you will use for most of your testing and debugging. Although the script offers several flags for configuration, you can leave most fields as their default values, since these are the parameters you will be graded on. To get a quick summary of the flags that the touchscreen runner supports, run

<center>python touchscreen_runner.py --help</center>

The important flags to know are:

<dl>
<dt>--width:</dt> <dd>The width of the touchscreen (optional, defaults to 20)</dd>
<dt>--height:</dt> <dd>The height of the touchscreen (optional, defaults to 20)</dd>
<dt>--frames:</dt> <dd>The number of frames to simulate (optional, defaults to 100)</dd>
<dt>--visualize:</dt> <dd>Shows a visualization of touchscreen simulation. This can be useful if you want to run a ton of simulations just to observe their behavior. When used with --evaluate, also visualizes the probability distributions returned by your touchscreenHMM at each timestep.</dd>
<dt>--evaluate:</dt> <dd>Scores the student's HMM on the given simulation. See Section 4.2 for more info about these scores and how they will translate into your grade.</dd>
</dl>

The `touchscreen_runner.py` file also supports flags --save_file, for saving simulations to a text file format, and --load_file, for loading those saved simulations. The reproducibility offered by the ability to load saved simulations can be useful for testing and debugging.

The text files created by the touchscreen runner will be saved in this format (see `simple_sim.txt`):

```
20 20 100
8 12 8 12
18 6 9 11
18 6 10 10
...
```

The first line contains three integers, representing the `width`, `height`, and number of `frames` of the simulation. In the example above, the simulation is size 20×20 and is 100 frames in length. Each subsequent line (representing one frame) contains four integers: `noisy_x`, `noisy_y`, `actual_x`, `actual_y`. In the second line: 8 12 8 12 denotes a noisy location of $(8, 12)$ and an actual location of $(8, 12)$ for that timepoint.

- `visualizer.py` Contains the visualizer for this project. It uses `matplotlib` to plot the `numpy` arrays and automatically progresses through the frames. You won't ever have to call this function directly.

- `constants.py` This file contains the constants for the noisy simulation. We will be testing your touchscreen with the given constants, but feel free to experiment to see what changing them does!

- `generate_data.py` You may find it useful to generate statistics or do some calculations comparing the noisy data to the actual finger location (hint hint). To make things easier, the function `create_simulations(size, frames)` will create a list of all frames in a random simulation for you. The `create_simulations` function is imported for you at the bottom of `touchscreen.py`, which you can run with the command `python touchscreen.py`.

2.4   Observations and Hints

- To simplify the problem for you, we have guaranteed that the actual and noisy positions are the same for the first frame of a simulation. Use this knowledge to your advantage.

- In a single timestep, George's finger will always either stay in the same place or move to one of the 8 adjacent locations (including diagonally adjacent locations).

- George's finger is more likely to continue moving in the same direction as it moved in the last timestep than it is to move in any other direction. Otherwise, his finger moves in a uniformly random direction.

- If George's finger is about to move off the edge of the touchscreen, he will pause for one timestep, and then be more likely to move in the opposite direction at the next timestep, just as if he had been moving in the opposite direction before.

- Run the visualizer on a few simulations. Look for patterns in the errors that the touchscreen makes. How would you describe these errors? How would you model this in your HMM?

- A state does not necessarily have to be a cell. For example, you can trade off runtime for accuracy by grouping together adjacent cells as one state. Concretely, one state could be that the finger is in one of the four cells in the top-left corner. There are conceivably other useful state representations as well.

- It will be difficult for you to get a sense for how well you are doing based on your score if there is not anything to compare it to. We recommend trying a few different solutions and comparing their scores (see Section 2.5 for scoring details). You will benefit from doing this both because it will probably lead to a better solution and because it will be useful when writing about your other approaches (see Section 3).

- You may want to take a look at question 5 in the writeup. This may give you some ideas on how to implement your touchscreenHMM.

2.5   Testing Part 2

You can run unit_tests.py for your solution in part 2, as well as write your own additional tests. In addition, we provide the below testing functionality:

We are providing you with the testing functions that we will be using to grade your filter_noisy_data implementation. These functions are contained in simulation_evaluator.py, which has code for a class called touchscreenEvaluator. A touchscreenEvaluator has two methods:

- calc_score(actual_frame, estimated_frame): This function calculates the individual score for a estimated_frame, which is a distribution of possible locations for the true position of a finger (returned by your touchscreenHMM's filter_noisy_data function). This works by awarding more points for higher distributions near the actual position of the finger by using a normal distribution.

- evaluate_touchscreen_hmm(touchscreenHMM, simulation): This function takes in an instance of your touchscreenHMM and outputs a map with 6 key-value pairs. The first value is an accuracy score for your touchscreenHMM, a weighted result of running calc_score() over all the frames in a given simulation. The second value is an accuracy score for if you just returned the noisy frame in filter_noisy_data. The fourth value is a number that we will be using to evaluate the consistency of your solution, and the fifth value is the same metric for if you just returned the noisy frame in filter_noisy_data. The rubric_ scores are given as estimates of what your actual autograded score would be for that single run (we will be averaging over multiple runs).

3   Part 3: Writeup

3.1   HMM Questions

Now that you have implemented your HMM, and George has successfully ordered his shipment, you will write about your approach to completing part 2. In your writeup, you should answer the following questions in separately numbered sections:

1. How did you model this situation as a HMM? What did you choose as the hidden states, observations, transition models, and sensor models? Be sure to describe these clearly and completely. This may easily take more than a page.

2. How did you select the transition and sensor models that you ended up using? How did observations from the simulator influence your choices? Account for each choice that you made.

3. What assumptions do you make in your approach? In particular, does your HMM relax either assumption 2 or 3 (or both) in section 1.3? For each of your assumptions, indicate whether they hold in reality.

4. What other approaches did you consider? Why did they seem promising? Why didn't you end up using them?

5. Smoothing and Prediction Inference Tasks

   (a) Explain (with examples, equations, etc.) what is smoothing in the context of HMMs. How would you modify your existing filtering code to implement smoothing over the touchscreen data? What should we expect as reasonable arguments and outputs for a function called `touchscreen_smoothing`?

   (b) Explain (with examples, equations, etc.) what is prediction in the context of HMMs. How would you modify your existing filtering code to implement making predictions over the touchscreen data? What should we expect as reasonable arguments and outputs for a function called `touchscreen_prediction`?

3.2 SRC Questions

Hidden Markov Models are used in a wide range of applications such as finance, bioinformatics, and speech recognition. As these applications see increasing use in industry, many people whose jobs revolved around the same applications may lose their jobs. Read the following article
`https://time.com/5876604/machines-jobs-coronavirus/` on the impact of AI replacing human workers during the pandemic. Please answer the following question:

- Imagine if there was an artificial intelligence algorithm created to make laws. Would you be comfortable replacing human law makers with that algorithm? What about the algorithms replacing truck drivers or McDonald's cashiers. Where do we draw the line for which jobs should we allow to be replaced or should we uphold that all jobs should be protected from being automated away?

Then pick **ONE** of the following three questions

1. Pick one of the job fields (ex: finance) discussed in the article which have been impacted by the increased application of AI. Did the displacement of jobs in that field surprise you? Do you think it was inevitable for these jobs to be "automated away"?

2. The latter half of the article focuses on retraining workers as a solution to the unemployment issue, do you think this is a viable approach?

3. Which party do you think is responsible for accounting for the displacement of jobs from AI? The developers of the AI? The government? The employer? Or another party?

4 Grading

We will give you your score based on the rubric in `rubric.txt`. You can take a look at this file to get a sense of the point allocations.

4.1   Part 1

For the basic HMM, we will be testing the functionality by adding test data to the HMM using `tell()` and comparing the results of `ask()` to the actual distribution. This portion will be auto graded with a certain window of error forgiveness (0.001). The run time for `tell()` and `ask()` should be less than 5 seconds. See the corresponding **Section 1.4: Testing Part 1** section for more info.

4.2   Part 2

For the touchscreen problem, your score will not be an all or nothing field like past auto-graded sections, but rather a sliding scale based on the performance of the HMM. Initializing an instance of your `touchscreenHMM` should take less than 1 minute, and each call to `filter_noisy_data` on a 20 by 20 touchscreen should take at most 10 seconds per frame. Our tests will consist of multiple simulations (size: 20×20) of 100 frames. We will be grading your implementation on two main factors:

- *accuracy*: we look at each frame returned from your `filter_noisy_data` call, and score you based on how tightly your distribution captures the actual finger location over all frames.

- *consistency*: we reward distributions that don't deviate as often from the actual finger location. For example, a solution that simply returns the noisy frame would not receive a good consistency score because its distribution often misses the actual location completely.

The evaluator we provide you returns a map with six key-value pairs: `accuracy_score`, `noisy_score`, `missed_frames`, `noisy_frames`, and two rubric estimates `rubric_1` and `rubric_2`. The `accuracy_score` will be a float between 0 and 100, where a higher score corresponds to better accuracy for this portion. You will be receiving some points as long as you are scoring higher than the `noisy_score`. For the consistency score, a **lower** number of `missed_frames` will result in a **higher** score. You should definitely obtain a lower number than `noisy_frames`.

Note: Because of the nature of the assignment and how our scoring system works, we expect perfect scores to be *exceptionally* rare. While we as TAs can give you ideas or guide you in the right direction, it may be worth noting that even our own solutions are not optimal! We know this assignment is difficult and want to see what you can do, which is why we are giving you almost 3 weeks to complete it. Don't worry too much about how high your scores should be, and just try to get the best scores that you can. If you've spent longer than 18 hours of genuine work on part 2, move on to the writeup and write about what you've tried.

5   Virtual Environment and Requirements

As usual, you should be using our virtual environment to run your code:

- To activate: `$ source /course/cs1410/venv/bin/activate`

- Then simply run your files as: `$ python <executable.py>`

- To deactivate: `$ deactivate`

Alternatively, if you choose to work on your own machine, there are some additional packages that are needed for this project, namely `scipy`, `matplotlib`, and `numpy`. These are already installed in our virtual environment, but you will need to install them on your own machine.

One quick way to install the dependencies you don't have is to use pip. If you don't have pip installed on your machine, you can find instructions here: `https://pip.pypa.io/en/stable/installing/`

6   Install and Handin Instructions

To install, accept the GitHub Classroom assignment at this link. This will create a private GitHub repository with the stencil code for you to work on the assignment.

To handin,

1. Make sure to push any changes you want to test to your private repository. You can do this by running

   ```
   git add .
   git commit -m "<a commit message describing what you changed>"
   git push
   ```

2. On Gradescope, click on the assignment you are submitting for.

3. Under "Submission Method", please select GitHub.

4. Under "Repository", you can search for your repository by typing "csci-1410" and selecting the repository for this assignment.

5. Under "Branch", you can select any branch that you want to be graded. So if you're testing something on a branch, you can see how its functionality performs here, before merging it to master. Feel free to upload your assignment as many times as you like before the deadline.

In accordance with the course grading policy, your written homework should not have any identifiable information on it, including your banner ID, name, or CS login.