Introduction

Silly Premise:
As expected, Mark and Elon have gotten into another Twitter fight over AI. This time, Mark has annoyed Elon so much that he threw his phone against the wall! Elon still has to type up his twitter response using his swipe keyboard, but his touchscreen has started to return data with errors. You will be helping Elon filter his touchscreen data using Hidden Markov Models (HMMs) so he can write his tweet.

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 \( i \)th 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:
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, ..., \( N - 1 \), 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, ....

1.3 Assumptions

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

1. The \( t \)th 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.

\[
Pr(X_{t+1}|X_{t}, X_{t-1}, ..., X_1) = Pr(X_{t+1}|X_t)
\]

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

\[
Pr(e_t|X_t, e_{t-1}, e_{t-2}, ..., e_1) = Pr(e_{t+1}|X_{t+1})
\]

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:

\[
Pr(X_{t+1}|e_{1,2,...,t+1}) = \alpha \cdot Pr(e_{t+1}|X_{t+1}) \sum_{X_t} Pr(X_{t+1}|X_t) \cdot Pr(X_t|e_1, e_2, ..., e_t)
\]

(1)

where \( \alpha \) is a normalizing constant.

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 Your Code

You should write your own 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 a sensor (sometimes called “observation”) model. For this reason, we supply some sample models in `supplied_models.py`. This file contains functions with sample data within them:

1. \texttt{observation\_model(\text{observation, state})}:
   Returns the probability of \texttt{observation} being emitted when we are in \texttt{state}.

2. \texttt{transition\_model(\text{old\_state, new\_state})}:
   Returns the probability of transitioning from \texttt{old\_state} to \texttt{new\_state}.

To run your HMM, you can run the file by executing \texttt{python supplied\_models.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: \texttt{suppliedModel} and \texttt{mismatchedObservations}. Each contain a relevant sensor and transition model. By default, the \texttt{suppliedModel} is used in \texttt{python supplied\_models.py} when you enter observations, but it can be easily changed to use \texttt{mismatchedObservations}. \texttt{mismatchedObservations} only accepts the three observations 'A', 'B' and 'C' and as a result is much easier to hand simulate. \texttt{mismatchedObservations} is notable since the number of states and observations is explicit, and the number of states and observations are different.

So that you can quickly verify whether your solution is correct, we also included a compiled TA solution in the \texttt{ta\_solution} folder. You can treat \texttt{part\_1\_solution.so} as if it were a python file, and import the HMM using \texttt{from ta\_solution.\text{part\_1\_solution} import HMM} to make sure that your HMM and the solution return the same values.

2 Part 2: Finger Tracking

2.1 The Problem

Now you get to apply the skills you learned implementing the generic HMM to help Elon filter his noisy touchscreen data. You will do this by filling in the stencil code for the \texttt{touchscreenHMM} class in \texttt{touchscreen.py}.

Your goal is to write \texttt{filter\_noisy\_data}, which takes in the touchscreen’s noisy reading of the location of Elon’s finger and outputs a probability distribution over the true location of Elon’s finger, considering the current noisy readings as well as all past noisy readings. To compare this to Part 1, \texttt{filter\_noisy\_data} should behave exactly as if you were to call \texttt{tell}, passing in the current noisy reading, and then \texttt{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 \texttt{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 \texttt{sensor\_model} and \texttt{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 \texttt{filter\_noisy\_data()} from an instance of your \texttt{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:

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

Where 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 momentum and some other interesting behaviours). 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.

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 Elon’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 representing numpy arrays:

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

  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.

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.

- simulator.so This file contains finger simulator algorithm as a compiled module. 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:

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

Where `width` and `height` are the dimensions of the touchscreen, and `frames` are the number of frames in the simulation. The default is a 20x20 touchscreen over 100 frames, which is what all grading scripts will be run on.

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 the `get_frame()` function 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

```python
sim.visualize_simulation(noisy=True, actual=True, frame_length=.1)
```

By changing the values of `noisy` and `actual`, you can plot the noisy data, actual position, or both. If you plot both, yellow represents the actual position, red represents the noisy position, and white 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()`.

- **run_visual_simulation.py**: Contains a script that runs all the visualization code on a new simulation. This can be useful if you want to run a ton of simulations just to observe their behaviour. You can call this function by running the following command in shell:

  ```shell
dir python run_visual_simulation.py
```

- **noisy_algorithm.so**: This file contains the compiled algorithm that is used to add noise to the actual position of the finger. It distributes the observed location over a secret algorithm, which you will have to use the visualizer to understand better. You will never have to use this file manually.

- **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 and return it.

### 2.3 Testing Your Code

For part 2, we are providing the testing functions that we will be using to grade the accuracy of the touchscreen HMM. These functions are in `simulation_tester.py` in the `touchscreenEvaluator` class:

- **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. This works by awarding more points for higher distributions near the actual position of the finger by using a normal distribution. We will always be using a 20x20 touchscreen.

- **evaluate_touchscreen_hmm(touchscreenHMM, simulation)**: This function takes in an instance of your `touchscreenHMM` and runs `calc_score()` over all the frames in a given `simulation`. It calculates an accuracy measure by comparing the score of a perfect estimation and the one returned by the touchscreenHMM, as well as the accuracy of a touchscreenHMM that just returns the noisy simulation as a probability distribution. The output is a tuple with the first value representing your touchscreenHMM’s accuracy, and the second value the accuracy if you just returned the noisy frame.
To individually test your simulation, you can use the following code snippet. This uses the same evaluation function that we will be using to grade your handin. We will be using the return values of `evaluate_touchscreen_hmm` in comparison to the actual HMM:

```python
[1]: student_solution = touchscreenHMM()
[2]: simulation_instance = touchscreenSimulator(width=20, height=20, frames=100)
[3]: evaluator = touchscreenEvaluator()
[4]: score = evaluator.evaluate_touchscreen_hmm(
    student_solution, simulation_instance)
[5]: score
```

>>> (0.82, 0.67)

We will be using these scores in comparison to an ideal HMM to determine your actual point score for this section. Don’t worry too much about how your score maps to your grade; just try to get the best score that you can. But do not spend too long on this! If you’ve spent longer than 18 hours of genuine work on section 2, just move on to section 3 and write about what you’ve tried.

2.4 Timeouts

There are a few timeout restrictions for this assignment. Initializing an instance of your `touchscreenHMM` should take less than 5 seconds. Actually running `filter_noisy_data` on a 20 by 20 touchscreen should take at most 1 minute per frame.

2.5 Hints

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

- Elon’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 Elon’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. You will benefit from doing this both because it will probably lead to a better solution and because writing about your other approaches may qualify you for extra credit (see Section 3).

3 Part 3: Writeup

3.1 What to Write

Now that you have implemented your HMM, and Elon has written his response tweet, you will write about your approach to completing Part 2. Your writeup should answer the following questions:

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. (For extra credit, if you have answered all other questions) What other approaches did you consider? Why did they seem promising? Why didn’t you end up using them?

You should compose your writeup in LaTeX. You may answer these questions in numbered or paragraph format, as long as it is clear where you answer each of them. Your writeup should not contain any Python code; you should describe your approach in words, mathematical notation, and maybe a bit of very readable pseudocode.

### 3.2 Grading

You will notice from the rubric that the grading of this writeup departs from the grading policy in one important way: we will not be grading you on persuasiveness. This means that you do not have to report the accuracy of your method or do anything else to defend it in your writeup. We are more interested in what choices you made and how you made them than in why the choices are good. Other than that, we are grading with the same criteria—accuracy, clarity, and completeness—as always.

### 4 Package Requirements

There are some additional packages that are needed for this project, namely scipy, matplotlib, and numpy. 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/](https://pip.pypa.io/en/stable/installing/)

To install the dependencies that you need with pip, you can run the following command in terminal:

```bash
terminal:~$ pip install -r requirements.txt --user
```

If this doesn’t work you can individually install the packages with pip.

### 5 Install and Handin Instructions

To install the stencil, run `cs1410_install HMM` in your cs1410 course directory.

Turn in `hmm.py` and `touchscreen.py` via the handin script.

Turn in a physical copy of the writeup to the handin bin on the second floor of the CIT.

Because this is a large project, we are giving you more time to complete it. Your code and writeup will be due on Thursday, more than two weeks from now. Because the assignment is long, having only a few days to resubmit would probably not be too helpful if you did something very wrong in your original submission. To compensate, we have provided support code that evaluates your code in the same way that our grader will (see the Testing Your Code sections). For this reason, we will not allow resubmissions for this assignment.