CS 4710: Artificial Intelligence

P4: Inference under Uncertainty, Naive Bayes and Classifiers

- Fall 2021
- Due: 10:00 PM Friday, November 05, 2021

In this assignment, you will use probabilistic inference and particle filtering to guide Pacman through a tracking problem. Note: We will use the Pacman framework developed at Berkeley. This framework is used worldwide to teach AI, therefore it is very important that you DO NOT publish your solutions online.

- Note: this is one of the hardest of the Pacman assignments, so we recommend you start early. As always, feel free to post your questions on Piazza.

Written Assignment (20 points)

As usual please go to the PDF in the written folder for the written instructions.

HMMs (5 points)

Typing Simulation (7 points)

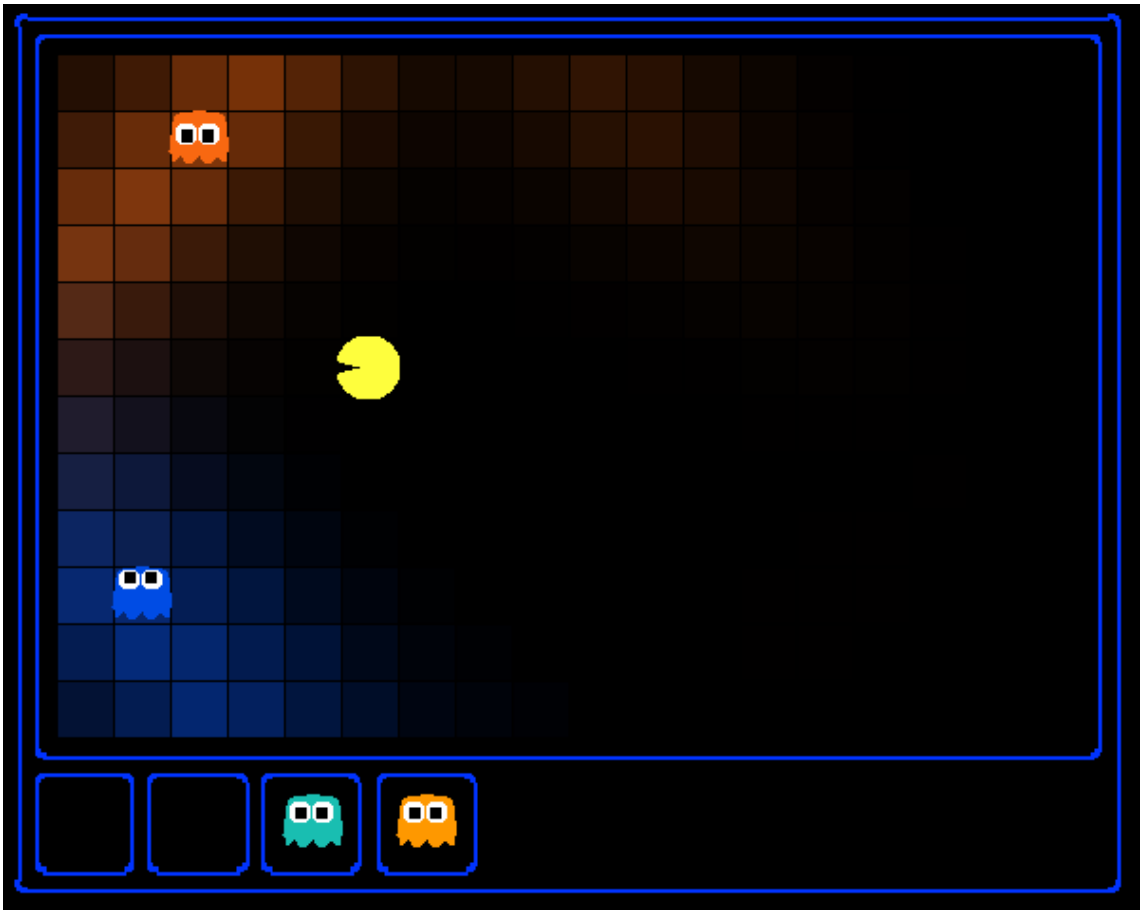
Part A (2 point)

Part B (2 points)

Part C (3 points)

Bayes Nets (8 points)

Pacman (14 points)



Files you'll edit (and thus submit to Gradescope for grading):

`bustersAgents.py` Agents for playing the Ghostbusters variant of Pacman.

`inference.py` Code for tracking ghosts over time using their sounds.

Pacman spends his life running from ghosts, but things were not always so. Legend has it that many years ago, Pacman's great grandfather Grandpac learned to hunt ghosts for sport. However, he was blinded by his power and could only track ghosts by their banging and clanging.

In this project, you will design Pacman agents that use sensors to locate and eat invisible ghosts. You'll advance from locating single, stationary ghosts to hunting packs of multiple moving ghosts with ruthless efficiency.

As always, your code will be autograded for technical correctness. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder.

Notes:

This is one of the hardest of the Pacman assignments, but also one of the most interesting. Be sure you understand the theoretical aspects of tracking and hidden Markov models, as well as particle filters, before you get started.

Intro

In our version of Ghostbusters, the goal is to hunt down scared but invisible ghosts. Pacman, ever resourceful, is equipped with sonar (ears) that provides noisy readings of the Manhattan distance to each ghost. The game ends when Pacman has eaten all the ghosts. To start, try playing a game yourself using the keyboard.

```
python busters.py
```

The blocks of color indicate where the each ghost could possibly be, given the noisy distance readings provided to Pacman. The noisy distances at the bottom of the display are always non-negative, and always within 7 of the true distance. The probability of a distance reading decreases exponentially with its difference from the true distance.

Your primary task in this project is to implement inference to track the ghosts. For the keyboard based game above, a crude form of inference was implemented for you by default: all squares in which a ghost could possibly be are shaded by the color of the ghost. Naturally, we want a better estimate of the ghost's position. Fortunately, Bayes' Nets provide us with powerful tools for making the most of the information we have. Throughout the rest of this project, you will implement algorithms for performing both exact and approximate inference using Bayes' Nets. The lab is challenging, so we do encouarge you to start early and seek help when necessary.

While watching and debugging your code with the autograder, it will be helpful to have some understanding of what the autograder is doing. There are 2 types of tests in this project, as differentiated by their `*.test` files found in the subdirectories of the test_cases folder. For tests of class `DoubleInferenceAgentTest`, your will see visualizations of the inference distributions generated by your code, but all Pacman actions will be preselected according to the actions of the staff implementation. This is necessary in order to allow comparision of your distributions with the staff's distributions. The second type of test is `GameScoreTest`, in which your BustersAgent will actually select actions for Pacman and you will watch your Pacman play and win games.

As you implement and debug your code, you may find it useful to run a single test at a time. In order to do this you will need to use the `-t` flag with the autograder. For example if you only want to run the first test of question 1, use:

```
python autograder.py -t test_cases/q1/1-ExactElapse
```

In general, all test cases can be found inside `test_cases/q*`.

Question 1 (4 points): Exact Inference with Time Elapse

We have provided you with the implementation for the observe method in the `ExactInference` class of `inference.py` to correctly update the agent's belief distribution over ghost positions given an observation from Pacman's sensors. Further details can be found in the comments of the observe function.

With the exact inference from observations implementation in mind, Pacman's observations are not his only source of knowledge about where a ghost may be. Pacman also has knowledge about the ways that a ghost may move; namely that the ghost can not move through a wall or more than one space in one timestep.

To understand why this is useful to Pacman, consider the following scenario in which there is Pacman and one Ghost. Pacman receives many observations which indicate the ghost is very near, but then one which indicates the ghost is very far. The reading indicating the ghost is very far is likely to be the result of a buggy sensor.

Pacman's prior knowledge of how the ghost may move will decrease the impact of this reading since Pacman knows the ghost could not move so far in only one move.

In this question, you will implement the `elapseTime` method in `ExactInference`. Your agent has access to the action distribution for any `GhostAgent`.

Since Pacman is not utilizing any observations about the ghost, this means that Pacman will start with a uniform distribution over all spaces, and then update his beliefs according to how he knows the Ghost is able to move. Since Pacman is not observing the ghost, this means the ghost's actions will not impact Pacman's beliefs. Over time, Pacman's beliefs will come to reflect places on the board where he believes ghosts are most likely to be given the geometry of the board and what Pacman already knows about their valid movements.

For the tests in this question we will sometimes use a ghost with random movements and other times we will use the `GoSouthGhost`. This ghost tends to move south so over time, and without any observations, Pacman's belief distribution should begin to focus around the bottom of the board. To see which ghost is used for each test case you can look in the .test files.

To run the autograder for this question and visualize the output:

`python autograder.py -q q1`

As an example of the `GoSouthGhostAgent`, you can run

`python autograder.py -t test_cases/q1/2-ExactElapse`

and observe that the distribution becomes concentrated at the bottom of the board.

As you watch the autograder output, remember that lighter squares indicate that pacman believes a ghost is more likely to occupy that location, and darker squares indicate a ghost is less likely to occupy that location. For which of the test cases do you notice differences emerging in the shading of the squares? Can you explain why some squares get lighter and some squares get darker?

Hints:

- Instructions for obtaining a distribution over where a ghost will go next, given its current position and the `gameState`, appears in the comments of `ExactInference.elapseTime` in `inference.py`.
- We assume that ghosts move independently of one another, so while you can develop all of your code for one ghost at a time, adding multiple ghosts should still work correctly.
- Before typing any code, write down the equation of the inference problem you are trying to solve.
- Try printing `noisyDistance`, `emissionModel`, and `PacmanPosition` (in the observe function) to get started.
- In the Pacman display, high posterior beliefs are represented by bright colors, while low beliefs are represented by dim colors. You should start with a large cloud of belief that shrinks over time as more evidence accumulates.
- Beliefs are stored as `util.Counter` objects (like dictionaries) in a field called `self.beliefs`.
- The only thing stored in `ExactInference` is `self.beliefs`.

Question 2 (3 points): Exact Inference Full Test

Now that Pacman knows how to use both his prior knowledge and his observations when figuring out where a ghost is, he is ready to hunt down ghosts on his own. This question will use the `observe` and `elapseTime` implementations together, along with a simple greedy hunting strategy which you will implement for this question. In the simple greedy strategy, Pacman assumes that each ghost is in its most likely position according to its beliefs, then moves toward the closest ghost. Up to this point, Pacman has moved by randomly selecting a valid action.

Implement the `chooseAction` method in `GreedyBustersAgent` in `bustersAgents.py`. Your agent should first find the most likely position of each remaining (uncaptured) ghost, then choose an action that minimizes the distance to the closest ghost. If correctly implemented, your agent should win the game in `q2/3-gameScoreTest` with a score greater than 700 at least 8 out of 10 times. Note: the autograder will also check the correctness of your inference directly, but the outcome of games is a reasonable sanity check.

To run the autograder for this question and visualize the output:

```
python autograder.py -q q2
```

Note: If you want to run this test (or any of the other tests) without graphics you can add the following flag:

```
python autograder.py -q q2 --no-graphics
```

Hints: +When correctly implemented, your agent will thrash around a bit in order to capture a ghost. +The comments of `chooseAction` provide you with useful method calls for computing maze distance and successor positions. +Make sure to only consider the living ghosts, as described in the comments.

Question 3 (3 points): Approximate Inference Observation

Approximate inference is very trendy among ghost hunters this season. Next, you will implement a particle filtering algorithm for tracking a single ghost.

Implement the functions `initializeUniformly`, `getBeliefDistribution`, and observe for the `ParticleFilter` class in `inference.py`. A correct implementation should also handle two special cases. (1) When all your particles receive zero weight based on the evidence, you should resample all particles from the prior to recover. (2) When a ghost is eaten, you should update all particles to place that ghost in its prison cell, as described in the comments of observe. When complete, you should be able to track ghosts nearly as effectively as with exact inference.

To run the autograder for this question and visualize the output:

```
python autograder.py -q q3
```

Hints: +A particle (sample) is a ghost position in this inference problem. +The belief cloud generated by a particle filter will look noisy compared to the one for exact inference. +`util.sample` or `util.nSample` will help you obtain samples from a distribution. If you use `util.sample` and your implementation is timing out, try using `util.nSample`.

Question 4 (4 points): Approximate Inference with Time Elapse

Implement the `elapseTime` function for the `ParticleFilter` class in `inference.py`. When complete, you should be able to track ghosts nearly as effectively as with exact

inference.

Note that in this question, we will test both the `elapseTime` function in isolation, as well as the full implementation of the particle filter combining `elapseTime` and `observe`.

To run the autograder for this question and visualize the output:

```
python autograder.py -q q4
```

It is recommended to run the autograder with the flag `--no-graphics` to prevent unnecessary timeout.

For the tests in this question we will sometimes use a ghost with random movements and other times we will use the `GoSouthGhost`. This ghost tends to move south so over time, and without any observations, Pacman's belief distribution should begin to focus around the bottom of the board. To see which ghost is used for each test case you can look in the .test files. As an example, you can run

```
python autograder.py -t test_cases/q4/2-ParticleElapse
```

and observe that the distribution becomes concentrated at the bottom of the board.

K-Means (6 points)

In this assignment, you will implement a K-Means clustering algorithm. Detailed instructions for each function can be found in the code/kmeans/kmeans.py file. To visualize your implementation, simply run:

```
python kmeans.py
```

Note: this requires package numpy and matplotlib. You can simply run `pip install numpy matplotlib` to install them.

Additionally, we provide a student version of the autograder we will use called `autograder.py`. To be clear, the autograder used to generate the grades will be essentially the same with the exception that the point values are changed. As a result, we provide `generator.py` to generate a new set of points to test against. This will generate a new file called `new.npy` which can be run through your code with:

```
python kmeans.py new.npy
```

To run the autograder, simply run:

```
python autograder.py
```

Note that if you change `easy.npy` or `hard.npy` the autograder will give you failing output.