

# AI Final Exam

Rong Zhang rz214

## Question 1

In this question, two python file are attached which are Q1\_2.py and Q1\_4.py.

a) There are 695 empty cells in the map, so without priority knowledge, the probability I am at G is  $1/695=0.00143884$ .

b) I didn't get a result in both this question and question (c) (my code is not working). Here I still attach my code (Q1\_2.py) and I will only talk about something I've done in this question.

Every time we move to a direction, the probability matrix will change. This is implemented in function move().

I implemented an A\* algorithm to find the sequence of actions. I choose number of previous actions as  $g(t)$ , and define a heuristic function

$$h(t) = \sum p(Cell_i) \times distance(Cell_i, goal)$$

in which distance is an Euclidean distance:

$$distance(a, b) = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$$

Then this algorithm will search the next probability matrix with minimum  $f(t)=g(t)+h(t)$  to move

I created a fringe to store the probability matrices which will be chosen next with minimum  $f(t)$  and a close-list to store the probability matrices I've been in. Every time I access a new probability matrix, I will calculate  $h(t)$  of four possible moves, and add these four matrices to the fringe (at the same time, compare these matrices with close list, if find the same one which means I have accessed the state before, the matrix will not be added into the fringe). Next, I will choose the probability matrix with minimum  $f(t)$  in the fringe and take the move. After move, add this probability matrix to the close list. However, with algorithm running, the size of close-list is getting bigger and bigger, which will stop my program.

d) In this question, my attached code is Q1\_4.py.

There are two steps in this question.

Firstly, when I observe the number of blocks surrounded by me, I will re-distribute the probability to the cells surrounded by this number of blocks. Probability of the cells surrounded by others numbers of blocks will be set to 0.

$$P(Cell_i) = \begin{cases} \frac{P(Cell_i)}{P_{total}}, & \text{if } Cell_i \text{ is surrounded by numbers of observation} \\ 0, & \text{if } Cell_i \text{ is not surrounded by numbers of observation} \end{cases}$$

where

$$P_{total} = \sum_{Cell \text{ surrounded by numbers of observation}} P(Cell_i)$$

In my code, function reDisP() is used to complete this work.

Secondly, I will be told to move one step to a direction. By doing this, the probability of each cell will be moved to the neighborhood in that direction, except the ones near the walls. The probability of the cells near the wall will be the sum of itself and the cell move to it. In my code, function move() is used to complete this.

1) My program will print the result in format “x, y: probability”. The cell whose probability is 0 will not be printed. The result is a list of 128 cells, and the probability of all of them is 0.0078125. Here is a screenshot for part of the result. Full result is too long to show in my report, please run my code to check them.

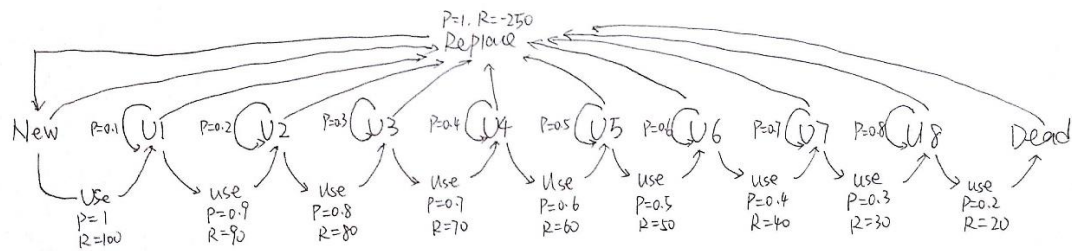
```
1 , 5 : 0.0078125
1 , 17 : 0.0078125
1 , 29 : 0.0078125
2 , 1 : 0.0078125
2 , 11 : 0.0078125
2 , 13 : 0.0078125
2 , 23 : 0.0078125
2 , 25 : 0.0078125
2 , 35 : 0.0078125
3 , 4 : 0.0078125
3 , 5 : 0.0078125
3 , 6 : 0.0078125
3 , 16 : 0.0078125
3 , 17 : 0.0078125
3 , 18 : 0.0078125
3 , 28 : 0.0078125
3 , 29 : 0.0078125
3 , 30 : 0.0078125
4 , 3 : 0.0078125
4 , 9 : 0.0078125
4 , 15 : 0.0078125
4 , 21 : 0.0078125
4 , 27 : 0.0078125
4 , 33 : 0.0078125
5 , 5 : 0.0078125
5 , 17 : 0.0078125
5 , 29 : 0.0078125
8 , 3 : 0.0078125
8 , 9 : 0.0078125
8 , 15 : 0.0078125
8 , 21 : 0.0078125
8 , 27 : 0.0078125
8 , 33 : 0.0078125
9 , 4 : 0.0078125
9 , 5 : 0.0078125
9 , 6 : 0.0078125
9 , 16 : 0.0078125
9 , 17 : 0.0078125
9 , 18 : 0.0078125
9 , 28 : 0.0078125
9 , 29 : 0.0078125
9 , 30 : 0.0078125
10 , 1 : 0.0078125
10 , 35 : 0.0078125
13 , 5 : 0.0078125
```

- 2) By running my code, you can input an observation sequence and an action sequence, and my program will output the result in format “x, y: probability”. There may be more than one cell for me to ‘most likely to be in’, and all of them will be printed.

## Question 2

In this question, three python files are attached, which are Q2\_1.py, Q2\_3.py, Q2\_4.py.

Here is a Markov Decision Process framework of this question.



a) Use Bellman equation to calculate the optimal utility of each state.

For state 'New':

$$U^*(New) = \text{MAX}[100 + \beta U^*(Used_1), -250 + \beta U^*(New)]$$

For  $0 < i < 8$ :

$$U^*(Used_i) = \text{MAX}[(100 - 10 \times i) + \beta[(1 - 0.1 \times i) \times U^*(Used_i) + 0.1 \times i \times U^*(Used_{i+1})], -250 + \beta U^*(New)]$$

Especially when  $i=8$ :

$$U^*(Used_8) = \text{MAX}[(100 - 10 \times i) + \beta[0.2 \times U^*(Used_8) + 0.8 \times U^*(Dead)], -250 + \beta U^*(New)]$$

For state 'Dead'

$$U^*(Dead) = -250 + \beta U^*(New)$$

I implemented code using value iteration (error set is 0.001), and calculate the result of optimal utility. Here is the result table. You can also run Q2\_1.py for this result.

STATE	UTILITY
New	800.5306559
Used1	778.3675014
Used2	643.2213408
Used3	556.1226157
Used4	502.8350489
Used5	475.8450497
Used6	470.4774949
Used7	470.4774949

Used8	470.4774949
Dead	470.4774949

- b) By calculating the optimal utility, I also get the results table of optimal policy. You can also run Q2\_1.py for this result.

STATE	POLICY
New	Use
Used1	Use
Used2	Use
Used3	Use
Used4	Use
Used5	Use
Used6	Replace
Used7	Replace
Used8	Replace
Dead	Replace

- c) In this question, new policy utility for replace an used machine is

$$-price + \beta[0.5 \times U^*(Used_1) + 0.5 \times U^*(Used_2)]$$

If this policy is better than replace a new one, customers will buy the used machine. So we need to find the maximum value of price that satisfy:

$$-price + \beta[0.5 \times U^*(Used_1) + 0.5 \times U^*(Used_2)] > -250 + \beta U^*(New)$$

Initialize the price as 250. To find the highest price, I decrease the value of price by 1 every loop until termination.

The result is 169. You can also run Q2\_3.py for this result.

- d) I test several beta values in this question using algorithms indicated in question a) (beta=0.1, 0.3, 0.5, 0.7, 0.9, 0.99, 0.992, 0.994, 0.996, 0.998, 0.999). From the result, when beta>=0.99,

the optimal policies will be stable. In case I can say, for sufficient large  $\beta(\geq 0.99)$ , the optimal policies is always optimal for all  $\beta$  values.

Here is a comparison of optimal policies in different  $\beta$  values:

<i>STATE</i>	<b><i>Beta&gt;0.99</i></b>	<i>Beta=0.9</i>	<i>Beta=0.7</i>	<i>Beta=0.5</i>	<i>Beta=0.3</i>	<i>Beta=0.1</i>
<i>New</i>	<b>Use</b>	Use	Use	Use	Use	Use
<i>Used1</i>	<b>Use</b>	Use	Use	Use	Use	Use
<i>Used2</i>	<b>Use</b>	Use	Use	Use	Use	Use
<i>Used3</i>	<b>Use</b>	Use	Use	Use	Use	Use
<i>Used4</i>	<b>Replace</b>	Use	Use	Use	Use	Use
<i>Used5</i>	<b>Replace</b>	Use	Use	Use	Use	Use
<i>Used6</i>	<b>Replace</b>	Replace	Use	Use	Use	Use
<i>Used7</i>	<b>Replace</b>	Replace	Use	Use	Use	Use
<i>Used8</i>	<b>Replace</b>	Replace	Use	Use	Use	Use
<i>Dead</i>	<b>Replace</b>	Replace	Replace	Replace	Replace	Replace

The larger  $\beta$  is, the more we consider future rewards. When  $\beta \geq 0.99$ , we can find it replacing a new machine will be a better policy in state Used4 to Used8 and Dead.

You can also run Q2\_4.py for this result.

### Question 3

In this question, two python file are attached, which are Q3\_1.py and Q3\_3.py.

- a) I implemented a logistic regression model to classify the given images.

Use given image (25 data) as input, and create weights in same size (25 data) randomly. Initiate  $w_0$  (intercept) randomly. The output will be a single value. Expected output ( $y$ ) for ClassA is 0, for ClassB is 1.

Firstly pre-process the input data. Because the sample size is limited, I generate a set of Gaussian noises (mean=0, standard deviation=0.1), and add them to the input to avoid over-

fitting.

$$x_i = x_{i0} + noise_i$$

So we have:

$$Output = h_w \left( w_0 + \sum_i w_i x_i \right)$$

Where:

$$h_w(x) = \frac{1}{1 + e^{-x}}$$

Define the loss function as:

$$Loss(output_t, y) = -y \ln[output_t] - (1 - y) \ln[1 - output_t]$$

Then gradient will be:

$$\nabla Loss(output_t, y) = (output_t - y) \times x$$

I choose learning rate  $\alpha=0.2$ , then train the model with Gradient Descent:

$$w_{i,t+1} = w_{i,t} - \alpha(output - y) \times x_i$$

Train this model until termination: 1000 times training of all the images or the sum of loss < 0.0005.

Use Mystery.txt as input and get the result:

```
0.9292380570605002
0.001171249325707813
0.9999888049216782
0.11850430465957558
0.8402464162120867
```

The classification of this model for Mystery set is B A B A B.

You can run Q3\_1.py for the result, and the result may differ from this in numbers but they give the same classification.

I think this model works well in classifying these two types of images. For input set, we can know ClassA are images with black cells mainly on top-left of the matrix, and ClassB are black cells mainly on down-right of the matrix. So we can easily find that Mystery is a set of B A B A, and the last is unknown. By running my algorithm, it gives correct answer on this classification, and predict the last image in ClassB (this is not confused because the input data is not strict enough).

- b) To avoid over-fitting, we can enlarge the data size, add noise to the system, or find a most

suitable training times for the model. Here I add some noise signals to input data.

I generate different sets of Gaussian noise distribution with mean=0, standard deviation=0.1 and add them to the input data before I input the data to the model. So that input data now is not as clear as 0/1 values. By adding the noise, the noise will pass to the output by the weights, and affect the loss. When the loss is reduced, the penalty of noise will be reduced too. In this way, noises can affect the update of weights. This will reduce the over-fitting to the model.

$$x_i = x_{i0} + noise_i$$

Another method of avoid over-fitting is find a suitable training times. After some test, I think 1000 times is a suitable value for the model to classify A and B accurately.

- c) In this question, I implemented a linear Perceptron using step function as an activator.

Like the first question, I add some noise to the input data in same way. Using given image as input, and generate weight randomly. Generate  $w_0$  (intercept) randomly. The output of this model will be 1/0.

For Perceptron, we have:

$$Output = Step\left(w_0 + \sum_i w_i x_i\right)$$

Where

$$Step(x) = \begin{cases} 1, & x > 0 \\ 0, & x < 0 \end{cases}$$

Then loss will be

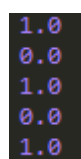
$$Loss(output_t, y) = \sum_{i=1}^N 1 \{output_t \neq y\}$$

I choose learning rate  $\alpha=0.4$ , then train the model with Gradient Descent:

$$w_{i,t+1} = w_{i,t} - \alpha(output - y) \times x_i$$

Train this model until Loss=0.

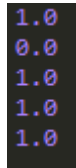
Here is a result using Perceptron:



It gives a good classification of Mystery which is B A B A B. However, because this model is a



linear model and input data cannot be divided linearly, sometimes it will give the wrong answer like this:



```
1.0
0.0
1.0
1.0
1.0
```

It made a mistake on the fourth image of Mystery.

So in fact, this model is not as good as logistic regression model. It is unstable to give a good or bad answer when predicting a new input. As a more stable choice, logistic regression is in a better performance in classifying these two class images.

You can run Q3\_3.py for this result.

#### Question 4

A ChefBot need to complete all the work of chefs. It need to be able to:

- a) Classifying different kinds of food, move them to different storage places. Get the food from the storage when needed.
- b) Get accurate amount of different kinds of food and seasoning based on recipes.
- c) Clean and wash food materials for cooking. Wash dishes.
- d) Cut food materials and prepare other things for cooking.
- e) Cook.
- f) Serve dishes for the customers.
- g) Communicate with customers, record their orders and suggest food for them.

...

A good ChefBot need to show good performance in different events. It needs to be quick and accurate when classifying the food and prepare cooking, and can deal with different conditions in cooking. It also need to communicate with the customers effectively to record their orders and serves them. To build such a bot, there will be some problems. For each of the problem, I give an analysis and an algorithm to deal with this problem:

1. Classifying different kinds of food is an important ability for a ChefBot. However, this is not an easy work for a bot. For different food, they are in different shapes and colors. We can let

the bot learn the pictures of different food to recognize them. Deep learning can be used here. We can construct a neural network (CNN), using images of food as input, then use their name as output to train this network. Because we want the bot to recognize the food as accurate as it can, we need to provide an enough large quantity of training images. We can pre-process the input image using clustering to reduce the input data size. In training, it is important to find a suitable number of hidden nodes. We can set a big number firstly, then train the network. If we find a node is useless, we can delete this node. We will find a suitable network to classify the food.

However, when our bot want to classify seasonings, it faces some problems. Salt and sugar looks similarly, ketchups are the same as Sriracha sauce in color. In this condition, we can try another way: recognize these seasonings by their labels. We need to construct another model to recognize the text on the label of the seasonings (OCR). In this part, we can construct a SVM and use different photos of text as input, then use the text as output. Train this model, then the bot can recognize different kinds of seasonings by taking a photo from the texts on their labels.

2. A good knife is needed in cooking. Our bot has a good knife for cutting the vegetables. However, with time going by, this knife is getting worse. A knife in bad condition will increase the time in cooking. When will the bot ask the owner to replace a new knife? We need to use a MDP process to deal with this problem. Define several states of a knife's life time. In state 'new', the knife can be used to state 'used', and because time consuming on preparing food is short when the knife is new, it will get a good reward by using a new knife. There will be several states for 'used' in different levels. In state 'used', time consuming on preparing food is longer than before, so it will get a smaller reward than before with higher levels. At each 'used' state, it has a probability to the next 'used' level, otherwise it stayed in the same state. Also, it can take the action 'replace' to replace a new knife. At last, the knife will be broken. In state 'broken', we need to replace a new knife. Action 'replace' will result in a minus reward which is the price of the knife.

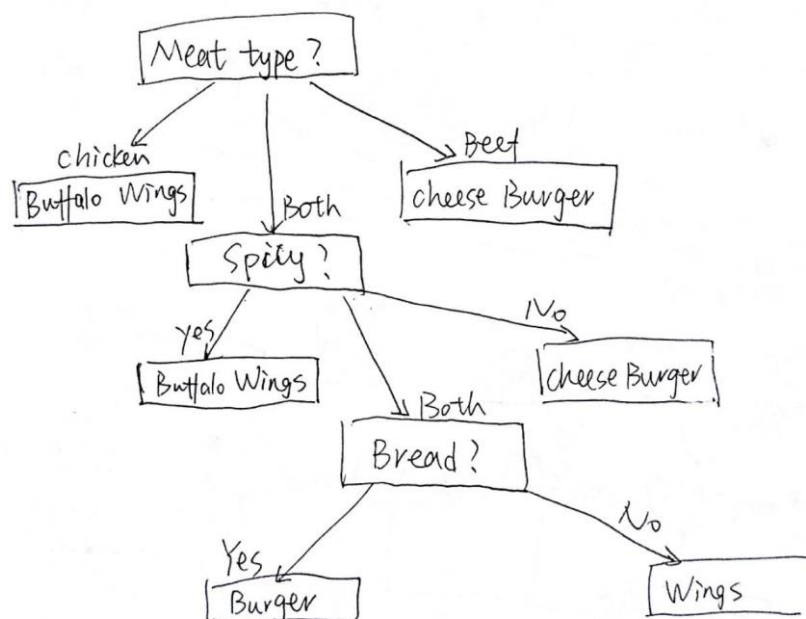
Build a MDP framework for this process, and use Bellman equations to represent them. Use value-iteration to calculate the optimal utility and optimal policy. User can choose different discount value to let the bot find the optimal policy at that time. Same questions may happen

on other cookware in the kitchen, and build a MDP is always a good choice in finding an optimal policy to deal with these things.

3. Here is a big warehouse to store different food materials. After getting an order from customer, the bot will go to pick up some food in the warehouse. It remembers the structure of the warehouse and the locations of different food, and it wants to find an optimal path to get the food and save time. This is a search problem. We can apply A\* algorithm to this bot to ensure finding an optimal path consuming shorter time comparing with BFS. Use distance to the goal as heuristic function to find an optimal path. If the location of food is unknown, DFS or BFS will work for this problem.

Similar algorithm can be applied to another scene: if the bot need to serve a dish to a customer, it needs to find an optimal path from kitchen to the table of the customer.

4. A ChefBot need to communicate with customers. Sometimes the ChefBot need to give suggestions for the customer when customer has dilemma on two kinds of food. Now the ChefBot needs a decision tree to deal with this problem. For example, when a customer has trouble on choosing cheese burger or buffalo chicken wings, we can construct a decision tree like this:



By using this decision tree, the bot can ask the customer some questions to choose a suitable food for them. However, this is a simple example. In fact, the bot may face a more complex problem. For each decision problem, we need to collect enough samples for input vectors and their output choice. Using these samples to find the attributes affecting this decision, and train the decision tree for the problem to let the bot use tree to make decisions for the problem.

Sometimes the customers don't give a specific choice of food, but they have some requests and ask the bot to suggest a dish that they may like. The bot has a database to store different recipes, and different recipes have different features. They are in same probability to be chosen firstly. What the bot needs to do is localizing the dish that the customer wants the most. Once the customer makes a request, the bot can update the probability distribution of dishes using the request (like "I want spicy food" then the bot will update the probability of spicy food, and the probability of non-spicy food will be set to zero). By repeating this step, the bot can recommend some dishes the customer may like with highest probability.

5. The most important thing for a ChefBot is cooking. The bot may face a planning problem in cooking. Here is an example: our bot has several plates to put the food material, and has some pots to cook the food. It is not accepted to put raw material and cooked material in one plate, also not accepted to put raw meat and raw vegetables in one plate. One pot can only cook one type of material at the same time. Now the bot has a steak, a rib, and some mushrooms. It has two plates (A,B) and one empty pot (a). Plate A is of steak and rib, plate B is of mushrooms. A customer orders a dish which is steak with mushrooms, now the bot needs to cook this dish. This bot needs to make a plan to choose the order for cook them to get the dish, at the same time it needs to obey the rules.

We can define some states in this problem:  $\text{InPlate}(\text{food}, \text{plate})$ ,  $\text{InPot}(\text{food}, \text{pot})$ ,  $\text{Raw}(\text{food})$ ,  $\text{Cooked}(\text{food})$ ,  $\text{Meat}(\text{food})$ ,  $\text{Vegetables}(\text{food})$  (raw and cooked are opposite, meat and vegetables are opposite when they are raw). Also we can define some actions:  $\text{AddToPot}(\text{food}, \text{pot}, \text{plate})$ ,  $\text{GetFromPot}(\text{food}, \text{pot}, \text{plate})$ .  $\text{AddToPot}$  will make the material from raw to cooked. Now we can describe the problem using these states and find a solution using the actions.

We can easily find a plan to solve this problem is  $\text{AddToPot}(\text{mushroom}, \text{a}, \text{B})$ ,

GetFromPot(mushroom,a,B), AddToPot(steak,a,A), GetFromPot(steak,a,B). But for the bot, it needs to find the solution by some search algorithms. Using backward state search is a good idea, the bot can reduce the search space of the actions with much lower branching factor than forward search.

For different conditions, there will be different kinds of planning problems and most of them are more complex than this example. By defining different states, actions and describe the problem in pre-condition and final condition, the bot can use backward state search to find a working plan to deal with the problems in cooking.