HOMEWORK 8: REINFORCEMENT LEARNING

10-601 Introduction to Machine Learning (Spring 2018)
Carnegie Mellon University
https://piazza.com/cmu/spring2018/10601
OUT: April 17, 2018
DUE: April 27, 2018 11:59 PM
TAs: Jennifer, Sienna, Yu, Han

Summary In this assignment, you will implement reinforcement learning algorithms for solving a maze. As a warmup, Section 1 will lead you through an on-paper example of how value iteration and q-learning work. Then, in Section 2, you will implement these two algorithms to solve mazes.

START HERE: Instructions

- **Collaboration Policy**: Collaboration on solving the homework is allowed, after you have thought about the problems on your own. It is also OK to get clarification (but not solutions) from books or online resources, again after you have thought about the problems on your own. There are two requirements: first, cite your collaborators fully and completely (e.g., "Jane explained to me what is asked in Question 3.4"). Second, write your solution *independently*: close the book and all of your notes, and send collaborators out of the room, so that the solution comes from you only. See the collaboration policy on the website for more information: http://www.cs.cmu.edu/~mgormley/courses/10601-s18/about.html
- **Late Submission Policy:** See the late submission policy here: http://www.cs.cmu.edu/~mgormley/courses/10601-s18/about.html
- **Submitting your work:** You will use Gradescope to submit answers to all questions, and Autolab to submit your code. Please follow instructions at the end of this PDF to correctly submit all your code to Autolab.
  - **Gradescope:** For written problems such as derivations, proofs, or plots we will be using Gradescope (https://gradescope.com/). Please use ther provided template. Submissions can be handwritten onto the template, but should be labeled and clearly legible. If your writing is not legible, you will not be awarded marks. Alternatively, submissions can be written in LaTeX. Regrade requests can be made, however this gives the TA the opportunity to regrade your entire paper, meaning if additional mistakes are found then points will be deducted.
  - **Autolab:** You will submit your code for programming questions on the homework to Autolab (https://autolab.andrew.cmu.edu/). After uploading your code, our grading scripts will autograde your assignment by running your program on a virtual machine (VM). The software installed on the VM is identical to that on `linux.andrew.cmu.edu`, so you should check that your code runs correctly there. If developing locally, check that the version number of the programming language environment (e.g. Python 2.7, Octave 3.8.2, Open-

JDK 1.8.0, g++ 4.8.5) and versions of permitted libraries (e.g. `numpy` 1.7.1) match those on `linux.andrew.cmu.edu`. (Octave users: Please make sure you do not use any Matlab-specific libraries in your code that might make it fail against our tests. Python3 users: Please pay special attention to the instructions at the end of this PDF) You have a **total of 10 Autolab submissions**. Use them wisely. In order to not waste Autolab submissions, we recommend debugging your implementation on your local machine (or the linux servers) and making sure your code is running correctly first before any Autolab submission.

- **Materials:** Download from Autolab the tar file ("Download handout"). The tar file will contain all the data that you will need in order to complete this assignment.

1 Written [30 points]

Answer the following questions in the HW8 solutions template provided. Then upload your solutions to Gradescope. You may use LaTeX or print the template and hand-write your answers then scan it in. Failure to use the template may result in a penalty.

1.1 Warmup Questions[14 points]

1.1.1 Value Iteration[6 points]

In this question you will carry out value iteration by hand to solve a maze, similar to what you will do in the programming part. A map of the maze is shown in the table below, where 'G' represents the goal of the agent (it's the terminal state); 'H' represents an obstacle; the zeros are the state values V(s) that are initialized to zero.

0	0	G
H	0	H
0	0	0

Table 1.1: Map of the maze

The agent can choose to move up, left, right, or down at each of the 6 states (the goal and obstacles are not valid initial states, "not moving" is not a valid action). The transitions are deterministic, so if the agent chooses to move left, the next state will be the grid to the left of the previous one. However, if it hits the wall (edge) or obstacle (H), it stays in the previous state. The agent receives a reward of -1 whenever it takes an action. The discount factor γ is 1.

1. (1 point) How many possible deterministic policies are there in this environment, including both optimal and non-optimal policies?
2. (3 points) Compute the state values after each round of synchronous value iteration updates on the map of the maze before convergence. For example, after the first round, the values should look like this:

-1	-1	G
H	-1	H
-1	-1	-1

Table 1.2: Value function after round 1

3. (2 points. Select all that apply) Which of the following changes will result in the same optimal policy as the settings above?
   A. The agent receives a reward of 10 when it takes an action that reaches G and receives a reward of -1 whenever it takes an action that doesn't reach G. Discount factor is 1.
   B. The agent receives a reward of 10 when it takes an action that reaches G and doesn't receive any reward whenever it takes an action that doesn't reach G. Discount factor is 1.
   C. The agent receives a reward of 10 when it takes an action that reaches G and doesn't receive any reward whenever it takes an action that doesn't reach G. Discount factor is 0.9.

D. The agent receives a reward of -10 when it takes an action that reaches G and doesn't receive any reward whenever it takes an action that doesn't reach G. Discount factor is 0.9.

1.1.2 Q-learning [8 points]

In this question, we will practice using the Q-learning algorithm to play tic-tac-toe. Tic-tac-toe is a simple two-player game. Each player, either X (cross) or O (circle), takes turns marking a location in a 3x3 grid. The player who first succeeds in placing three of their marks in a column, a row, or a diagonal wins the game.

1	2	3
4	5	6
7	8	9

Table 1.3: tic-tac-toe board positions

We will model the game as follows: each board location corresponds to an integer between 1 and 9, illustrated in the graph above. Actions are also represented by an integer between 1 and 9. Playing action a results in marking the location a and an action a is only valid if the location a has not been marked by any of the players. We train the model by playing against an expert. The agent only receives a possibly nonzero reward when the game ends. Note a game ends when a player wins or when every location in the grid has been occupied. The reward is +1 if it wins, -1 if it loses and 0 if the game draws.

O	X	
O	O	X
		X

Table 1.4: State 1 (circle's turn)

To further simplify the question, let's say we are the circle player and it's our turn. Our goal is to try to learn the best end-game strategy given the current state of the game illustrated in table1.4. The possible actions we can take are the positions that are unmarked: $\{3, 7, 8\}$. If we select action 7, the game ends and we receive a reward of +1; if we select action 8, the expert will select action 3 to end the game and we'll receive a reward of -1; if we select action 3, the expert will respond by selecting action 7, which results in the state of the game in table1.5. In this scenario, our only possible action is 8, which ends the game and we receive a reward of 0.

O	X	O
O	O	X
X		X

Table 1.5: State 2 (circle's turn)

Suppose we apply a learning rate $\alpha = 0.01$ and discount factor $\gamma = 1$. The Q-values are initialized as:

$$Q(1, 3) = 0.6$$
$$Q(1, 7) = -0.3$$
$$Q(1, 8) = -0.5$$

$$Q(2, 8) = 0.8$$

1. (1 point) In the first episode, the agent takes action 7, receives +1 reward, and the episode terminates. Derive the updated Q-value after this episode. Remember that given the sampled experience (s, a, r, s') of (state, action, reward, next state), the update of the Q value is:

   $$Q(s, a) = Q(s, a) + \alpha(r + \gamma max_{a' \in A} Q(s', a') - Q(s, a))$$

   Note if s' is the terminal state, $Q(s', a') = 0$ for all a'.

2. (1 point) In the second episode, the agent takes action 8, receives a reward of -1, and the episode terminates. Derive the updated Q-value based on this episode.

3. (2 points) In the third episode, the agent takes action 3, receives a reward of 0, and arrives at State 2 (1.5). It then takes action 8, receives a reward of 0, and the episode terminates. Derive the updated Q-values after each of the two experiences in this episode. Suppose we update the corresponding Q-value right after every single step.

4. (2 points) If we run the three episodes in cycle forever, what will be the final values of the four Q-values.

5. (2 points) What will happen if the agent adopts the greedy policy (always pick the action that has the highest current Q-value) during training? Calculate the final four Q-values in this case.

1.2 Empirical Questions[16 points]

The following questions should be completed after you work through the programming portion of this assignment (Section 2).

We will ask you to solve the following two mazes using value iteration and q-learning.

Maze 1 (3x5):

```
**.*G
...*.
S*...
```

Maze 2 (8x8):

```
...*..*G
......*.
...*..*.
...*....
.*..*...
.....*.*
***...*.
S.......
```

1. (4 points) Solve the two mazes using value iteration. Report the time taken for execution, the number of iterations it took to converge, and the value function of the states marked in 1.6 and 1.7. **Please round to two decimal places**. Use discount factor $\gamma = 0.9$. We say that the algorithm converges when the change of $V(s)$ for all s is small (less than 0.001).

		(0, 2)		
(2, 0)				(2, 4)

Table 1.6: Maze 1 Map

(0, 0)							
							(6, 7)
(7, 0)							

Table 1.7: Maze 2 Map

2. (4 points) Solve the two mazes using Q-learning agent. Specifically, run the agent for 2000 episodes with epsilon-greedy policy. Report the time taken for execution, the average number of steps in each

episode, and the value function of the states marked in 1.6 and 1.7. **Please round to two decimal places**. Use discount factor $\gamma = 0.9$, learning rate of 0.1 and epsilon $\epsilon = 0.2$.

3. (2 points) Examine and comment on the results for both methods. Which method runs faster? Do you see any difference between the value functions obtained? Are the optimal policies different? What do you think have caused the differences observed, if any?

4. (4 points) Now try using ϵ of 0.01 and 0.8 and compare the results with those you obtained using ϵ of 0.2. Examine and comment on the results **for Maze 2 only**: do you observe any difference? Especially examine the time taken for execution and the value functions obtained.

5. (2 points) Which method (value iteration or Q-learning) do you think is more suitable for this task of maze solving? Explain your reasoning. In what cases do you think the "worse" method here would become a better one?

2 Programming [70 points]

Your goal in this assignment is to implement the value iteration and Q-learning algorithms to solve mazes. You will implement all of the functions needed to initialize, train, evaluate, and obtain the optimal policies and values with these reinforcement learning algorithms.

The programs you write will be automatically graded using the Autolab system. You may write your programs in **Octave, Python, Java, or C++**. However, you should use the same language for all parts below.

We will break the coding part down into two sections: value iteration and Q-learning.

2.1 Specification of the maze

In this assignment, you will be given a `.txt` file as input to your program. This file describes structure of the maze. Any maze is rectangular with a start state in the bottom left corner and a goal state in the upper right corner. We use "S" to indicate the **initial state** (start state), "G" to indicate the **terminal state** (goal state), "*" to indicate an obstacle, and "." to indicate a state an agent can go through. The agent cannot go through states with obstacles. You can assume that there are no obstacles at "S" and "G". There are walls around the border of the maze. For example, an input file could look like the following

```
.*..*..G
S......*
```

The state is described by a tuple (x, y) where x is the row and y is the column (zero-indexed). In the above example, the initial state S is $(1, 0)$ and the terminal state G is $(0, 7)$. In addition, there are obstacles at states $(0, 1)$, $(0, 4)$, and $(1, 7)$.

There are four available actions, $0, 1, 2, 3$, at each state and they correspond to the directions "West", "North", "East", "South" respectively. The state the agent ends up in after taking any action at any state is **deterministic**: if an agent is at the initial state S and takes action 2 (East), then the next state of the agent is $(1, 1)$. If the agent takes an action that would result in going into a state with an obstacle, it will instead stay in the same state. For example, if the agent takes action 2 (East) at state $(0, 0)$ in the above example, the next state of the agent will still be $(0, 0)$. If the agent takes an action that would result in hitting the wall, it will also stay in the same state. For example, if the agent takes action 0 (West) at state $(0, 0)$ in the above example, the next state of the agent will still be $(0, 0)$.

The goal of the agent is to take as few steps as possible to get to the goal state from the start state. Hence, the reward is -1 whenever an agent takes an action, regardless of the state at which it takes an action.

2.2 Value iteration

Value iteration is a planning method used when the model of the world is known. We assume that we have access to the following information about the world a priori.

- A set of states $s \in S$
- A set of actions a an agent could take at each state $s \in S$
- Transition probability $T(s, a, s')$: the probability of being at state s' after taking action a at state s
- Reward function $R(s, a, s')$: the reward of being at state s' after taking action a at state s

Write a program `value_iteration.{py|java|cpp|m}` that implements the value iteration algorithm. Initialize $V(s) = 0$ for each state s that is not occupied by an obstacle and we don't model $V(s)$ when s is occupied by an obstacle. Please implement **synchronous** value iteration.

The autograder runs and evaluates the output from the files generated, using the following command:

2.2.1 Command Line Arguments

For Python:	`$ python value_iteration.py [args...]`
For Java:	`$ javac -cp "./lib/ejml-v0.33-libs/*:./"value_iteration.java` `$ java -cp "./lib/ejml-v0.33-libs/*:./" value_iteration [args...]`
For C++:	`$ g++ -g -std=c++11 -I./lib value_iteration.cpp; ./a.out [args...]`
For Octave:	`$ octave -qH value_iteration.m [args...]`

Where above `[args...]` is a placeholder for 6 command-line arguments: `<maze_input>` `<value_file>` `<q_value_file>`, `<policy_file>`, `<num_epoch>`, `<discount_factor>`. These arguments are described in detail below:

1. `<maze_input>`: path to the environment input `.txt` described previously
2. `<value_file>`: path to output the values $V(s)$
3. `<q_value_file>`: path to output the q-values $Q(s, a)$
4. `<policy_file>`: path to output the optimal actions $\pi(s)$
5. `<num_epoch>`: the number of epochs your program should train the agent for. In one epoch, your program should update the value $V(s)$ once for each $s \in S$.
6. `<discount_factor>`: the discount factor γ

As an example, if you implemented your program in Python, the following command line would run your program with maze input file `tiny_maze.txt` for 5 epochs and a discount factor of 0.9.

```
$ python value_iteration.py tiny_maze.txt value_output.txt \
q_value_output.txt policy_output.txt 5 0.9
```

2.2.2 Output

Your program should write three output `.txt` files containing the value and the optimal action of each state s and the q-value for each state action pair (s, a) after running the value iteration algorithm for the number of epochs specified by the command line argument.

The `<value_file>` should be formatted as "x y value" for each state $s = (x, y)$ and its corresponding value $V(s)$. You should output one line for each state that is not occupied by an obstacle. The order of the states in the output does not matter. Use `\n` to create a new line.

The `<q_value_file>` should be formatted as "x y action q_value" for each state $s = (x, y)$ and action a pair and the corresponding Q value $Q(s, a)$. You should output one line for each state-action pair where the

state is not occupied by an obstacle. The order of the state-action pair of the output does not matter. Use `\n` to create a new line. **Please compute $Q(s, a)$ using the values $V(s)$ after the specified number of epochs.**

The `<policy_file>` should be formatted as "x y optimal_action" for each state $s = (x, y)$ and the corresponding optimal policy $\pi(s)$. You should output one line for each state that is not occupied by an obstacle. The order of the states in the output does not matter. If there is a draw in the Q values, pick the action represented by the smallest number. For example, for state s, if $Q(s, 0) = Q(s, 2)$, then output action 0. Use `\n` to create a new line. **Please compute $\pi(s)$ using the values $V(s)$ after the specified number of epochs.**

Your output should match those of a reference implementation – this will be checked by the autograder by running your program and evaluating your output file against the reference solution.

For example, if maze.txt contains the following maze

```
.G
S*
```

and `<num_epoch>` is 5, the outputs should be

`<value_file>`

```
0 0 -1.0
0 1 0.0
1 0 -1.9
```

`<q_value_file>`

```
0 0 0 -1.9
0 0 1 -1.9
0 0 2 -1.0
0 0 3 -2.71
0 1 0 0.0
0 1 1 0.0
0 1 2 0.0
0 1 3 0.0
1 0 0 -2.71
1 0 1 -1.9
1 0 2 -2.71
1 0 3 -2.71
```

`<policy_file>`

```
0 0 2.0
0 1 0.0
1 0 1.0
```

2.3 Environment class

When an agent does not yet have a model of the world, it acts in the environment and receives feedback from the environment when it takes an action. The agent does not have explicit access to the transition probability

$T(s, a, s')$ and the reward $R(s, a, s')$ for each tuple (s, a, s') like we had for value iteration. To simulate this for q-learning, we ask you to implement an environment class such that it supports three methods. The environment also keeps track of the current state of the agent. Suppose the current state is s and the agent takes an action a. The environment returns the following three values when the agent takes an action a at state s:

- `next_state`: the state of the agent after taking action a at state s
- `reward`: the reward received from taking action a at state s
- `is_terminal`: an integer value indicating whether the agent reaches a terminal state after taking action a at state s. The value is 1 if terminal state is reached and 0 otherwise.

In this assignment, we will ask you to implement an environment class that supports the following abstraction interface:

- constructor:
  The constructor takes in a string containing the file name that specifies the maze environment described in 2.1. Based on the content of this file, you should maintain a representation of the maze. You should also initialize the state of the agent here.
  Input arguments: $filename$
- method **step**
  This function takes in an action a, simulates a step, sets the current state to the next state, and returns $next\_state$, $reward$, $is\_terminal$.
  Input arguments: a
  Return value: $next\_state$, $reward$, $is\_terminal$
- method **reset**
  This function resets the agent state to the initial state and returns the initial state.
  Input arguments: (Does not take any arguments)
  Return value: the initial state

To give you some feedback on whether you implemented the environment correctly, we'll have an autograder set up to test for the implementation of the environment.

Write a program `environment.{py|java|cpp|m}` that implements an environment class and the corresponding code to test the environment class.

The autograder runs and evaluates the output from the files generated, using the following command:

2.3.1 Command Line Arguments

For Python:
```
$ python environment.py [args...]
```

For Java:
```
$ javac -cp "./lib/ejml-v0.33-libs/*:./" environment.java
$ java -cp "./lib/ejml-v0.33-libs/*:./" environment [args...]
```

For C++:
```
$ g++ -g -std=c++11 -I./lib environment.cpp; ./a.out [args...]
```

For Octave:
```
$ octave -qH environment.m [args...]
```

Where above `[args...]` is a placeholder for 3 command-line arguments: `<maze_input>` `<output_file>` `<action_seq>`. These arguments are described in detail below:

1. `<maze_input>`: path to the environment input `.txt` described in section 2.1
2. `<output_file>`: path to output feedback from the environment after the agent takes the sequence of actions specified with the next argument.
3. `<action_seq_file>`: path to the file containing a sequence of 0, 1, 2, 3s that indicates the actions to take in order. This file has exactly one line. A white space separates any two adjacent numbers. For example, if the file contains the line 0 3 2, the agent should first go left, then go down, and finally go right.

As an example, if you implemented your program in Python, the following command line would run your program.

```
$ python environment.py medium_maze.txt output.feedback \
action_seq.txt
```

2.3.2 Output: Environment Feedback Files

Your program should write one output file whose filename ends with `.feedback`. It should contain the return values of the **step** method in order. Each line should be formatted as "`next_state_x next_state_y reward is_terminal`", where "`next_state_x`" and "`next_state_y`" are the x and y coordinates of the next state returned by the **method**. Exactly one space separates any adjacent values on the same line. Use `\n` to create a new line.

Your output should exactly match those of a reference implementation – this will be checked by the autograder by running your program and evaluating your output file against the reference solution.

For example, if `medium_maze.txt` contains the following maze

```
.*..*..G
S......*
```

and action_seq.txt contains the following,

```
0 1 3
```

the output should be

```
1 0 -1 0
0 0 -1 0
1 0 -1 0
```

2.4 Q learning

The Q learning algorithm is a model-free reinforcement learning algorithm and we assume we don't have access to the model of the environment we're interacting with. We also don't build a complete model of the environment during the learning process. A learning agent interacts with the environment solely based on calls to the **init**, **step** and **reset** methods of the Environment class we asked you to implement in 2.3. Then the Q learning algorithm updates the q-values based on the values returned by these methods. Write a

program `q_learning.{py|java|cpp|m}` that implements the Q learning algorithm. Initialize all the q values to 0 and use the epsilon-greedy strategy for action selection.

Let the learning rate be α and discount factor be γ. Recall that we have the information after one interaction with the environment, (s, a, r, s'). The update rule based on this information is:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha\left(r + \gamma \underset{a'}{max} Q(s',a')\right)$$

The epsilon-greedy action selection method selects the optimal action with probability $1-\epsilon$ and selects uniformly at random from one of the 4 actions (0, 1, 2, 3) with probability ϵ. The reason that we use an epsilon-greedy action selection is we would like the agent to do explorations as well. For the purpose of testing, we will test the case where $\epsilon = 0$ and $1 > \epsilon > 0$. When $\epsilon = 0$, the program becomes deterministic and your output have to match our reference output accurately. In this case, if there is a draw in the greedy action selection process, pick the action represented by the smallest number. For example, if we're at state s and $Q(s,0) = Q(s,2)$, then take action 0. And when $\epsilon > 0$, your reference output will need to fall in a certain range that we determine by running exhaustive experiments based on the input parameters.

The auto-grader runs and evaluates the output from the files generated, using the following command:

2.4.1 Command Line Arguments

For Python:
```
$ python q_learning.py [args...]
```

For Java:
```
$ javac -cp "./lib/ejml-v0.33-libs/*:./" q_learning.java
$ java -cp "./lib/ejml-v0.33-libs/*:./" q_learning [args...]
```

For C++:
```
$ g++ -g -std=c++11 -I./lib q_learning.cpp; ./a.out [args...]
```

For Octave:
```
$ octave -qH q_learning.m [args...]
```

Where above `[args...]` is a placeholder for 9 command-line arguments: `<maze_input>` `<value_file>` `<q_value_file>` `<policy_file>` `<num_episodes>` `<max_episode_length>` `<learning_rate>` `<discount_factor>` `<epsilon>`. These arguments are described in detail below:

1. `<maze_input>`: path to the environment input `.txt` described previously
2. `<value_file>`: path to output the values $V(s)$
3. `<q_value_file>`: path to output the q-values $Q(s,a)$
4. `<policy_file>`: path to output the optimal actions $\pi(s)$
5. `<num_episodes>`: the number of episodes your program should train the agent for. One episode is a sequence of states, actions and rewards, which ends with terminal state or ends when the maximum episode length has been reached.
6. `<max_episode_length>`: the maximum of the length of an episode. When this is reached, we terminate the current episode.
7. `<learning_rate>`: the learning rate α of the q learning algorithm
8. `<discount_factor>`: the discount factor γ.

9. <epsilon>: the value ϵ for the epsilon-greedy strategy

As an example, if you implemented your program in Python, the following command line would run your program with maze input file tiny_maze.txt for 1000 episodes with the maximum episode length of 20, learning rate of 0.8, discount factor of 0.9, and epsilon of 0.05:

```
$ python q_learning.py tiny_maze.txt value_output.txt \
q_value_output.txt policy_output.txt 1000 20 0.8 0.9 0.05
```

2.4.2 Output

Your program should write one output .txt file containing the optimal action of each state s and the q-values for each state action pair (s, a) after running the value iteration algorithm for the number of epochs specified by the command line.

Note for the purpose of testing for correctness of your implementation, you can directly parse the given maze and directly get access to where the locations of the obstacles are at. In general, when using model-free methods, these details should be hidden from the agent.

The <value_file> should be formatted as "x y value" for each state $s = (x, y)$ and its corresponding value $V(s)$. You should output one line for each state that is not occupied by an obstacle. The order of the states in the output does not matter. Use \n to create a new line.

The <q_value_file> should be formatted as "x y action q_value" for each state $s = (x, y)$ and action a pair and the corresponding Q value $Q(s, a)$. You should output one line for each state-action pair where the state is not occupied by an obstacle. The order of the state-action pair of the output does not matter. Use \n to create a new line.

The <policy_file> should be formatted as "x y optimal_action" for each state $s = (x, y)$ and the corresponding optimal policy $\pi(s)$. You should output one line for each state that is not occupied by an obstacle. The order of the states in the output does not matter. If there is a draw in the Q values, pick the action represented by the smallest number. For example, for state s, if $Q(s, 0) = Q(s, 2)$, then output action 0. Use \n to create a new line. Your output should match those of a reference implementation – this will be checked by the autograder by running your program and evaluating your output file against the reference solution.

For example, if tiny_maze.txt contains the following maze

```
.G
S*
```

and we run the program with number of episodes of 1000, the maximum episode length of 20, learning rate of 0.8, discount factor of 0.9, and epsilon of 0.05, the outputs should be

<value_file>

```
0 0 -1.0
0 1 0.0
1 0 -1.9
```

<q_value_file>

```
0 0 0 -1.9
0 0 1 -1.9
0 0 2 -1.0
0 0 3 -2.7099998634
0 1 0 0.0
0 1 1 0.0
0 1 2 0.0
0 1 3 0.0
1 0 0 -2.71
1 0 1 -1.9
1 0 2 -2.71
1 0 3 -2.70999998982
```

<policy_file>

```
0 0 2.0
0 1 0.0
1 0 1.0
```

2.5 Autolab Submission

You must submit a .tar file named `rl.tar` containing `environment.{py|m|java|cpp}`, `value_iteration.{py|m|java|cpp}`, `q_learning.{py|m|java|cpp}`.

You can create that file by running:

```
tar -cvf rl.tar environment.{py|m|java|cpp} \
value_iteration.{py|m|java|cpp} q_learning.{py|m|java|cpp}
```

from the directory containing your code. You may make up to **10 submissions** to Autolab before the deadline, but only your last submission will be graded.

Some additional tips: **DO NOT** compress your files; you are just creating a tarball. Do not use tar `-czvf`. **DO NOT** put the above files in a folder and then tar the folder. Autolab is case sensitive, so observe that all your files should be named in **lowercase**. You must submit this file to the corresponding homework link on Autolab. The autograder for Autolab prints out some additional information about the tests that it ran. You can view this output by selecting "Handin History" from the menu and then clicking one of the scores you received for a submission. For example on this assignment, among other things, the autograder will print out which language it detects (e.g. Python, Octave, C++, Java).

Python3 Users: Please include a blank file called python3.txt (case-sensitive) in your tar submission and we will execute your submitted program using Python 3 instead of Python 2.7.

Linear Algebra Libraries It is often more convenient to have a linear algebra library at your disposal. In this assignment, Java users may use EJML[a] and C++ users Eigen[b]. Details below. (As usual, Python users have numpy; Octave users have built-in matrix support.)

Java EJML is a pure Java linear algebra package with three interfaces. We strongly recommend using the SimpleMatrix interface. Autolab will use EJML version 3.3. The command line arguments above demonstrate how we will call you code. The classpath inclusion `-cp "./lib/ejml-v0.33-libs/*:./"` will ensure that all the EJML jars are on the classpath as well as your code.

C++ Eigen is a header-only library, so there is no linking to worry about—just `#include` whatever components you need. Autolab will use Eigen version 3.3.4. The command line arguments above demonstrate how we will call you code. The argument `-I./lib` will include the `lib/Eigen` subdirectory, which contains all the headers.

We have included the correct versions of EJML/Eigen in the handout.tar for your convenience. Do **not** include EJML or Eigen in your Autolab submission tar; the autograder will ensure that they are in place.

[a] https://ejml.org
[b] http://eigen.tuxfamily.org/