1. Unsupervised Learning ¶

In [10]:
```
%matplotlib inline
import scipy
import numpy as np
import itertools
import matplotlib.pyplot as plt
```

1. Generating the data

First, we will generate some data for this problem. Set the number of points $N = 400$, their dimension $D = 2$, and the number of clusters $K = 2$, and generate data from the distribution $p(x|z = k) = \mathcal{N}(\mu_k, \Sigma_k)$. Sample 200 data points for $k = 1$ and 200 for $k = 2$, with

$$\mu_1 = \begin{bmatrix} 0.1 \\ 0.1 \end{bmatrix}, \mu_2 = \begin{bmatrix} 6.0 \\ 0.1 \end{bmatrix} \text{ and } \Sigma_1 = \Sigma_2 = \begin{bmatrix} 10 & 7 \\ 7 & 10 \end{bmatrix}$$

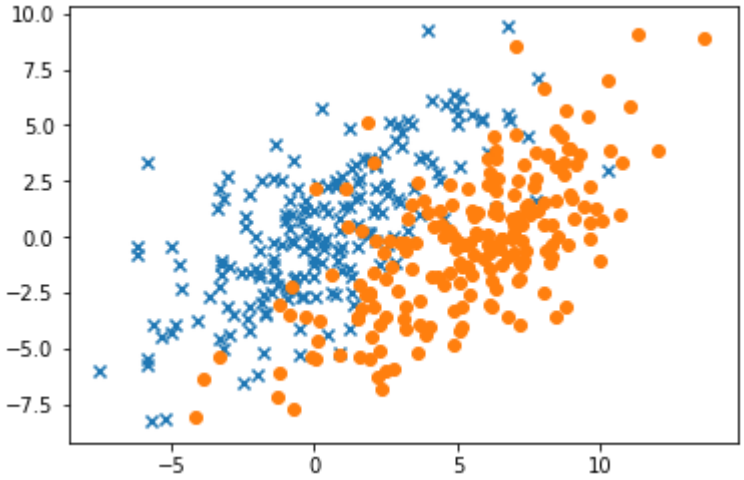
Here, $N = 400$. Since you generated the data, you already know which sample comes from which class. Run the cell in the IPython notebook to generate the data.

In [3]:
```
# TODO: Run this cell to generate the data
num_samples = 400
cov = np.array([[1., .7], [.7, 1.]]) * 10
mean_1 = [.1, .1]
mean_2 = [6., .1]

x_class1 = np.random.multivariate_normal(mean_1, cov, num_samples // 2)
x_class2 = np.random.multivariate_normal(mean_2, cov, num_samples // 2)
xy_class1 = np.column_stack((x_class1, np.zeros(num_samples // 2)))
xy_class2 = np.column_stack((x_class2, np.ones(num_samples // 2)))
data_full = np.row_stack([xy_class1, xy_class2])
np.random.shuffle(data_full)
data = data_full[:, :2]
labels = data_full[:, 2]
```

Make a scatter plot of the data points showing the true cluster assignment of each point using different color codes and shape (x for first class and circles for second class):

In [5]:
```
# TODO: Make a scatterplot for the data points showing the true cluster assign
ments of each point
plt.scatter(xy_class1[:,0], xy_class1[:,1], marker='x') # first class, x shape
plt.scatter(xy_class2[:,0], xy_class2[:,1], marker="o") # second class, circle
shape
```

Out[5]:
```
<matplotlib.collections.PathCollection at 0x1bf3d2a8d30>
```



2. Implement and Run K-Means algorithm

Now, we assume that the true class labels are not known. Implement the k-means algorithm for this problem. Write two functions: `km_assignment_step` , and `km_refitting_step` as given in the lecture (Here, `km_` means k-means). Identify the correct arguments, and the order to run them. Initialize the algorithm with

$$\hat{\mu}_1 = \begin{bmatrix} 0.0 \\ 0.0 \end{bmatrix}, \hat{\mu}_2 = \begin{bmatrix} 1.0 \\ 1.0 \end{bmatrix}$$

and run it until convergence. Show the resulting cluster assignments on a scatter plot either using different color codes or shape or both. Also plot the cost vs. the number of iterations. Report your misclassification error.

In [6]:
```
def cost(data, R, Mu):
    N, D = data.shape
    K = Mu.shape[1]
    J = 0
    for k in range(K):
        J += np.sum(np.dot(np.linalg.norm(data - np.array([Mu[:, k], ] * N), a
xis=1)**2, R))
    return J
```

In [7]:
```
# TODO: K-Means Assignment Step
def km_assignment_step(data, Mu):
    """ Compute K-Means assignment step

    Args:
        data: a NxD matrix for the data points
        Mu: a DxK matrix for the cluster means locations

    Returns:
        R_new: a NxK matrix of responsibilities
    """

    # Fill this in:
    N, D = data.shape[0], data.shape[1]
    K = Mu.shape[1]
    r = np.zeros((N,K))
    for k in range(K):
         r[:, k] = np.linalg.norm(Mu[:,k] - data, axis=1)
    arg_min = np.argmin(r, axis=1)
    # argmax/argmin along dimension 1
    R_new = np.zeros((N,K))
    R_new[range(N), arg_min]= 1
    # R_new[..., ...] = 1 # Assign to 1
    return R_new
```

In [8]:
```
# TODO: K-means Refitting Step
def km_refitting_step(data, R, Mu):
    """ Compute K-Means refitting step.

    Args:
        data: a NxD matrix for the data points
        R: a NxK matrix of responsibilities
        Mu: a DxK matrix for the cluster means locations

    Returns:
        Mu_new: a DxK matrix for the new cluster means locations
    """
    N, D = data.shape[0], data.shape[1]
    K = R.shape[1]

    Mu_new = np.zeros((D,K))
    for k in range(K):
        Mu_new[:, k] = np.mean(data[R==k], 0)
    print(Mu_new.shape)
    return Mu_new
```

In [9]:
```
# TODO: Run this cell to call the K-means algorithm
N, D = data.shape
K = 2
max_iter = 100
class_init = np.random.binomial(1., .5, size=N)
R = np.vstack([class_init, 1 - class_init]).T

Mu = np.zeros([D, K])
Mu[:, 1] = 1.
R.T.dot(data), np.sum(R, axis=0)

cost_list = []
iteration_list = []
for it in range(max_iter):
    R = km_assignment_step(data, Mu)
    Mu = km_refitting_step(data, R, Mu)
    print(it, cost(data, R, Mu))
    cost_list.append( cost(data, R, Mu))
    iteration_list.append(it)

class_1 = np.where(R[:, 0])
class_2 = np.where(R[:, 1])
```

```
(2, 2)
0 25344.22632709909
(2, 2)
1 24967.0662901935
(2, 2)
2 24967.0662901935
(2, 2)
3 24967.0662901935
(2, 2)
4 24967.0662901935
(2, 2)
5 24967.0662901935
(2, 2)
6 24967.0662901935
(2, 2)
7 24967.0662901935
(2, 2)
8 24967.0662901935
(2, 2)
9 24967.0662901935
(2, 2)
10 24967.0662901935
(2, 2)
11 24967.0662901935
(2, 2)
12 24967.0662901935
(2, 2)
13 24967.0662901935
(2, 2)
14 24967.0662901935
(2, 2)
15 24967.0662901935
(2, 2)
16 24967.0662901935
(2, 2)
17 24967.0662901935
(2, 2)
18 24967.0662901935
(2, 2)
19 24967.0662901935
(2, 2)
20 24967.0662901935
(2, 2)
21 24967.0662901935
(2, 2)
22 24967.0662901935
(2, 2)
23 24967.0662901935
(2, 2)
24 24967.0662901935
(2, 2)
25 24967.0662901935
(2, 2)
26 24967.0662901935
(2, 2)
27 24967.0662901935
(2, 2)
```

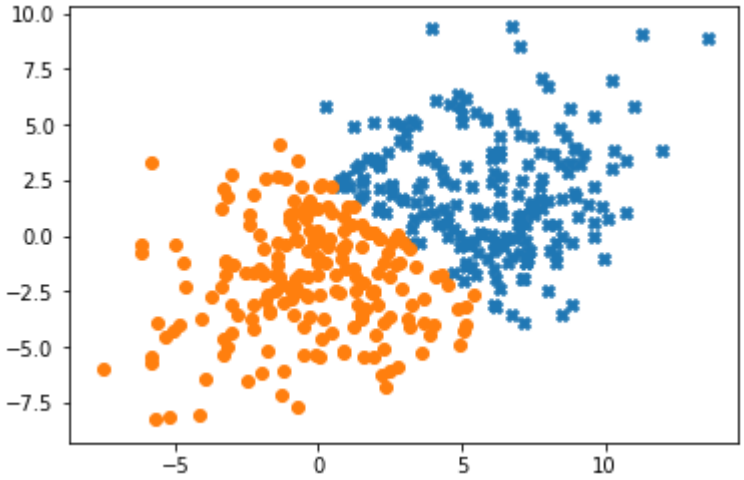
```
28 24967.0662901935
(2, 2)
29 24967.0662901935
(2, 2)
30 24967.0662901935
(2, 2)
31 24967.0662901935
(2, 2)
32 24967.0662901935
(2, 2)
33 24967.0662901935
(2, 2)
34 24967.0662901935
(2, 2)
35 24967.0662901935
(2, 2)
36 24967.0662901935
(2, 2)
37 24967.0662901935
(2, 2)
38 24967.0662901935
(2, 2)
39 24967.0662901935
(2, 2)
40 24967.0662901935
(2, 2)
41 24967.0662901935
(2, 2)
42 24967.0662901935
(2, 2)
43 24967.0662901935
(2, 2)
44 24967.0662901935
(2, 2)
45 24967.0662901935
(2, 2)
46 24967.0662901935
(2, 2)
47 24967.0662901935
(2, 2)
48 24967.0662901935
(2, 2)
49 24967.0662901935
(2, 2)
50 24967.0662901935
(2, 2)
51 24967.0662901935
(2, 2)
52 24967.0662901935
(2, 2)
53 24967.0662901935
(2, 2)
54 24967.0662901935
(2, 2)
55 24967.0662901935
(2, 2)
56 24967.0662901935
```

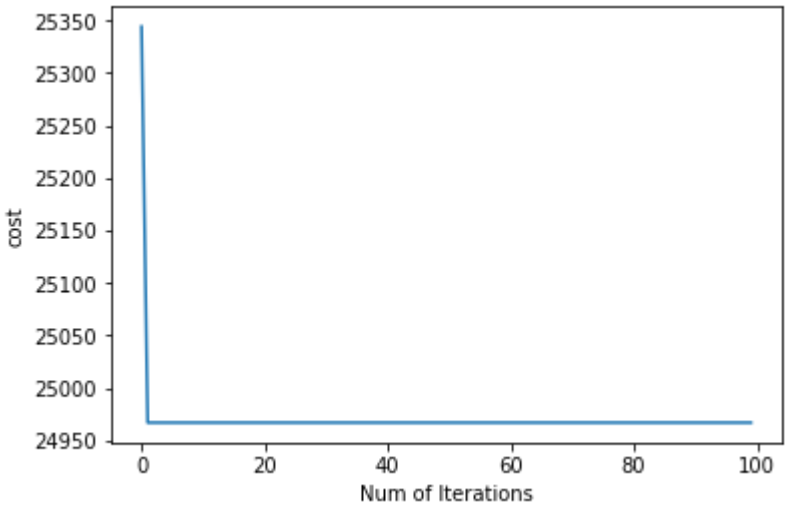
```
(2, 2)
57 24967.0662901935
(2, 2)
58 24967.0662901935
(2, 2)
59 24967.0662901935
(2, 2)
60 24967.0662901935
(2, 2)
61 24967.0662901935
(2, 2)
62 24967.0662901935
(2, 2)
63 24967.0662901935
(2, 2)
64 24967.0662901935
(2, 2)
65 24967.0662901935
(2, 2)
66 24967.0662901935
(2, 2)
67 24967.0662901935
(2, 2)
68 24967.0662901935
(2, 2)
69 24967.0662901935
(2, 2)
70 24967.0662901935
(2, 2)
71 24967.0662901935
(2, 2)
72 24967.0662901935
(2, 2)
73 24967.0662901935
(2, 2)
74 24967.0662901935
(2, 2)
75 24967.0662901935
(2, 2)
76 24967.0662901935
(2, 2)
77 24967.0662901935
(2, 2)
78 24967.0662901935
(2, 2)
79 24967.0662901935
(2, 2)
80 24967.0662901935
(2, 2)
81 24967.0662901935
(2, 2)
82 24967.0662901935
(2, 2)
83 24967.0662901935
(2, 2)
84 24967.0662901935
(2, 2)
```

```
85 24967.0662901935
(2, 2)
86 24967.0662901935
(2, 2)
87 24967.0662901935
(2, 2)
88 24967.0662901935
(2, 2)
89 24967.0662901935
(2, 2)
90 24967.0662901935
(2, 2)
91 24967.0662901935
(2, 2)
92 24967.0662901935
(2, 2)
93 24967.0662901935
(2, 2)
94 24967.0662901935
(2, 2)
95 24967.0662901935
(2, 2)
96 24967.0662901935
(2, 2)
97 24967.0662901935
(2, 2)
98 24967.0662901935
(2, 2)
99 24967.0662901935
```

In [11]:
```
# TODO: Make a scatterplot for the data points showing the K-Means cluster ass
ignments of each point
plt.figure(0)
plt.scatter(data[class_1, 0], data[class_1,1], marker="X") # first class, x sh
ape
plt.scatter(data[class_2, 0], data[class_2,1], marker="o") # second class, cir
cle
plt.figure(1)
plt.plot(iteration_list, cost_list)
plt.xlabel("Num of Iterations")
plt.ylabel("cost")


# f2.show()
```

Out[11]:
```
Text(0, 0.5, 'cost')
```





3. Implement EM algorithm for Gaussian mixtures

Next, implement the EM algorithm for Gaussian mixtures. Write three functions: `log_likelihood`, `gm_e_step`, and `gm_m_step` as given in the lecture. Identify the correct arguments, and the order to run them. Initialize the algorithm with means as in Qs 2.1 k-means initialization, covariances with $\hat{\Sigma}_1 = \hat{\Sigma}_2 = I$, and $\hat{\pi}_1 = \hat{\pi}_2$.

In addition to the update equations in the lecture, for the M (Maximization) step, you also need to use this following equation to update the covariance Σ_k:

$$\hat{\mathbf{\Sigma}_k} = \frac{1}{N_k} \sum_{n=1}^{N} r_k^{(n)} (\mathbf{x}^{(n)} - \hat{\mu_k})(\mathbf{x}^{(n)} - \hat{\mu_k})^\top$$

Run the algorithm until convergence and show the resulting cluster assignments on a scatter plot either using different color codes or shape or both. Also plot the log-likelihood vs. the number of iterations. Report your misclassification error.

In [12]:
```
def normal_density(x, mu, Sigma):
    return np.exp(-.5 * np.dot(x - mu, np.linalg.solve(Sigma, x - mu))) \
        / np.sqrt(np.linalg.det(2 * np.pi * Sigma))
```

In [13]:
```
def log_likelihood(data, Mu, Sigma, Pi):
    """ Compute log likelihood on the data given the Gaussian Mixture Parameters.

    Args:
        data: a NxD matrix for the data points
        Mu: a DxK matrix for the means of the K Gaussian Mixtures
        Sigma: a list of size K with each element being DxD covariance matrix
        Pi: a vector of size K for the mixing coefficients

    Returns:
        L: a scalar denoting the log likelihood of the data given the Gaussian Mixture
    """
    # Fill this in:
    N, D = data.shape[0], data.shape[1]
    K = Mu.shape[1]
    L, T = 0., 0.
    for n in range(N):
        for k in range(K):
            T += Pi[k]* normal_density(data[n], Mu[:,k], Sigma[k])
            # Compute the likelihood from the k-th Gaussian weighted by the mixing coefficients
        L += np.log(T)
    return L
```

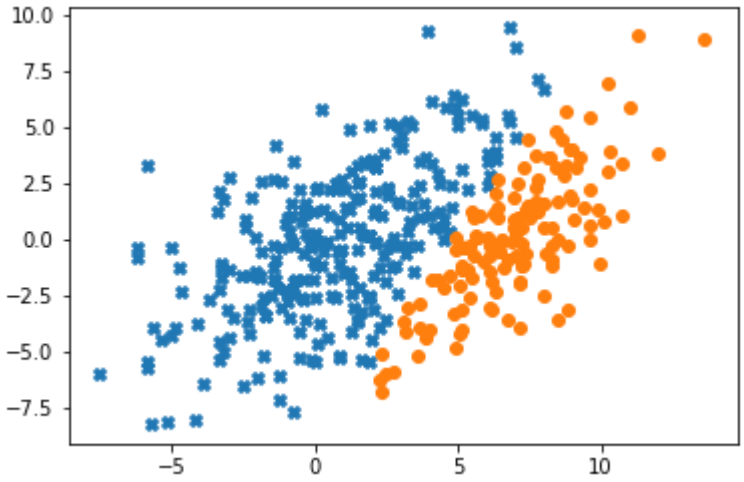
In [14]:
```
# TODO: Gaussian Mixture Expectation Step
def gm_e_step(data, Mu, Sigma, Pi):
    """ Gaussian Mixture Expectation Step.

    Args:
        data: a NxD matrix for the data points
        Mu: a DxK matrix for the means of the K Gaussian Mixtures
        Sigma: a list of size K with each element being DxD covariance matrix
        Pi: a vector of size K for the mixing coefficients

    Returns:
        Gamma: a NxK matrix of responsibilities
    """
    # Fill this in:
    N, D = data.shape[0], data.shape[1]
    K = Mu.shape[1]
    Gamma = np.zeros((N,K))
    for n in range(N):
        for k in range(K):
            Gamma[n, k] = Pi[k] * normal_density(data[n,:], Mu[:,k], Sigma[k])
        Gamma[n, :] /= np.sum(Gamma[n, :])
    return Gamma
```

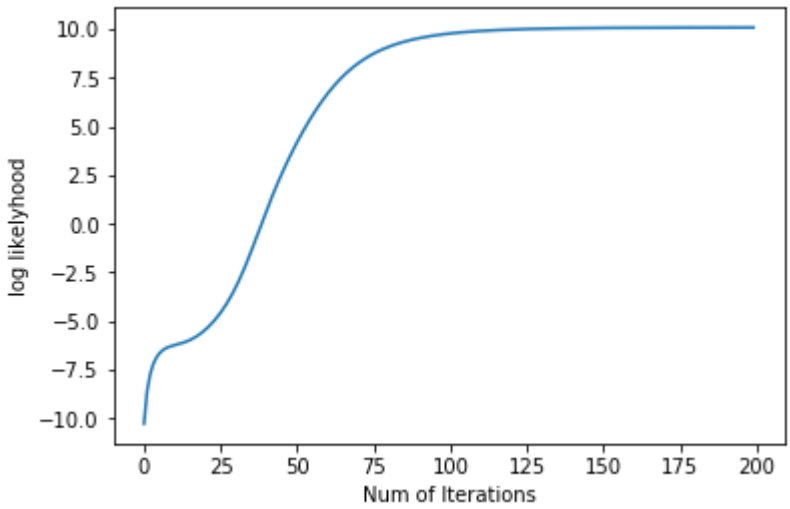
In [16]:
```
# TODO: Gaussian Mixture Maximization Step
def gm_m_step(data, Gamma):
    """ Gaussian Mixture Maximization Step.

    Args:
        data: a NxD matrix for the data points
        Gamma: a NxK matrix of responsibilities

    Returns:
        Mu: a DxK matrix for the means of the K Gaussian Mixtures
        Sigma: a list of size K with each element being DxD covariance matrix
        Pi: a vector of size K for the mixing coefficients
    """
    # Fill this in:
    N, D = data.shape[0], data.shape[1]
    K = Gamma.shape[1]
    Nk = np.sum(Gamma, axis=0)# Sum along first axis
    Mu = 1./Nk * (np.dot(data.T, Gamma))
    Sigma = [np.eye(D) for i in range(K)]
    for k in range(K):
        last = data-Mu[:,k]
        middle = np.eye(N)*Gamma[:, k]
        first = last.T
        third = first @ middle @ last
        Sigma[k] = (1./Nk[k])*third
    Pi =  Nk/N
    return Mu, Sigma, Pi
```

In [17]:
```
# TODO: Run this cell to call the Gaussian Mixture EM algorithm
N, D = data.shape
K = 2
Mu = np.zeros([D, K])
Mu[:, 1] = 1.
Sigma = [np.eye(2), np.eye(2)]
Pi = np.ones(K) / K
Gamma = np.zeros([N, K]) # Gamma is the matrix of responsibilities

max_iter  = 200

log_list = []
it_list = []
for it in range(max_iter):
    Gamma = gm_e_step(data, Mu, Sigma, Pi)
    Mu, Sigma, Pi = gm_m_step(data, Gamma)
    it_list.append(it)
    log_list.append(log_likelihood(data, Mu, Sigma, Pi))
    # print(it, log_likelihood(data, Mu, Sigma, Pi)) # This function makes the
computation longer, but good for debugging

class_1 = np.where(Gamma[:, 0] >= .5)
class_2 = np.where(Gamma[:, 1] >= .5)
```

In [18]:
```
# TODO: Make a scatterplot for the data points showing the Gaussian Mixture cl
uster assignments of each point
plt.figure(0)
plt.scatter(data[class_1, 0], data[class_1,1], marker="X") # first class, x sh
ape
plt.scatter(data[class_2, 0], data[class_2,1], marker="o")
plt.figure(1)
# print(log_list)
plt.plot(it_list, log_list)
plt.xlabel("Num of Iterations")
plt.ylabel("log likelyhood")
```

Out[18]:
```
Text(0, 0.5, 'log likelyhood')
```





4. Comment on findings + additional experiments

Comment on the results:

- Compare the performance of k-Means and EM based on the resulting cluster assignments.
- Compare the performance of k-Means and EM based on their convergence rate. What is the bottleneck for which method?
- Experiment with 5 different data realizations (generate new data), run your algorithms, and summarize your findings. Does the algorithm performance depend on different realizations of data?

d) i) The k-means clustering algorithm sets the cluster to be divided through a diagonal that goes from the top left to botton right, while the EM divides it through from top right to bottom left. This can be a result of K-means algorithm converging to a local minimum, which made it choose the center mean that it did.

ii) Since k-means only runs one iteration because it converges after one, it get stuck at a local minimum and so it sets the center means at the incorrect classification. EM runs through several iterations before converging, hence why its able to classify the data better than k-means. See the figures above for further details.

iii) Because the data is randomize, sometimes the k-means algorithm classifies properly, whereas other time it gets stuck at a local min. Depending on the data, the EM algorithm will take a while to converge; ...

2. Reinforcement Learning

There are 3 files:

1. `maze.py` : defines the `MazeEnv` class, the simulation environment which the Q-learning agent will interact in.
2. `qlearning.py` : defines the `qlearn` function which you will implement, along with several helper functions. Follow the instructions in the file.
3. `plotting_utils.py` : defines several plotting and visualization utilities. In particular, you will use `plot_steps_vs_iters` , `plot_several_steps_vs_iters` , `plot_policy_from_q`

In [11]:
```
from qlearning import qlearn
from maze import MazeEnv, ProbabilisticMazeEnv
from plotting_utils import plot_steps_vs_iters, plot_several_steps_vs_iters, p
lot_policy_from_q
```

1. Basic Q Learning experiments

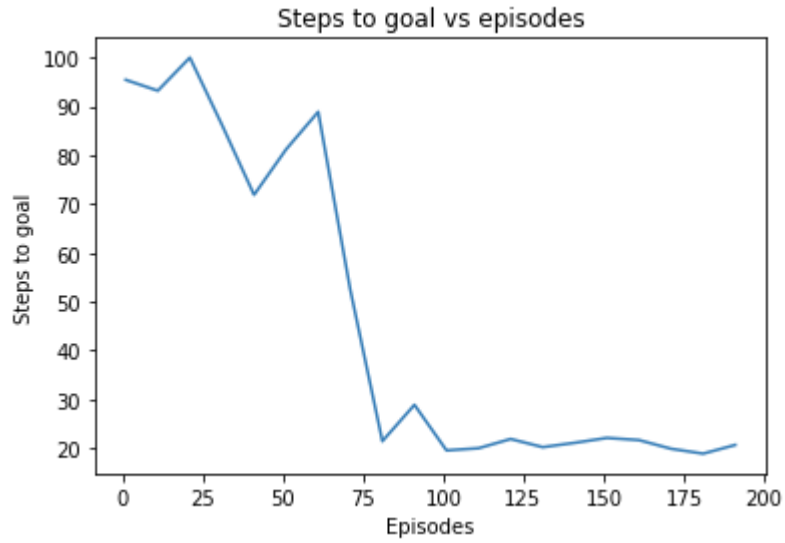
(a) Run your algorithm several times on the given environment. Use the following hyperparameters:

1. Number of episodes = 200
2. Alpha (α) learning rate = 1.0
3. Maximum number of steps per episode = 100. An episode ends when the agent reaches a goal state, or uses the maximum number of steps per episode
4. Gamma (γ) discount factor = 0.9
5. Epsilon (ϵ) for ϵ-greedy = 0.1 (10% of the time). Note that we should "break-ties" when the Q-values are zero for all the actions (happens initially) by essentially choosing uniformly from the action. So now you have two conditions to act randomly: for epsilon amount of the time, or if the Q values are all zero.

In [20]:
```
# TODO: Fill this in
num_iters = 200
alpha =1.0
gamma = 0.9
epsilon = 0.1
max_steps = 100
use_softmax_policy = False

# TODO: Instantiate the MazeEnv environment with default arguments
env = MazeEnv()

# TODO: Run Q-learning:
#(env, num_iters, alpha, gamma, epsilon, max_steps, use_softmax_policy, init_b
eta=None, k_exp_sched=None):
q_hat, steps_vs_iters = qlearn(env, num_iters, alpha, gamma, epsilon, max_step
s, use_softmax_policy)
```

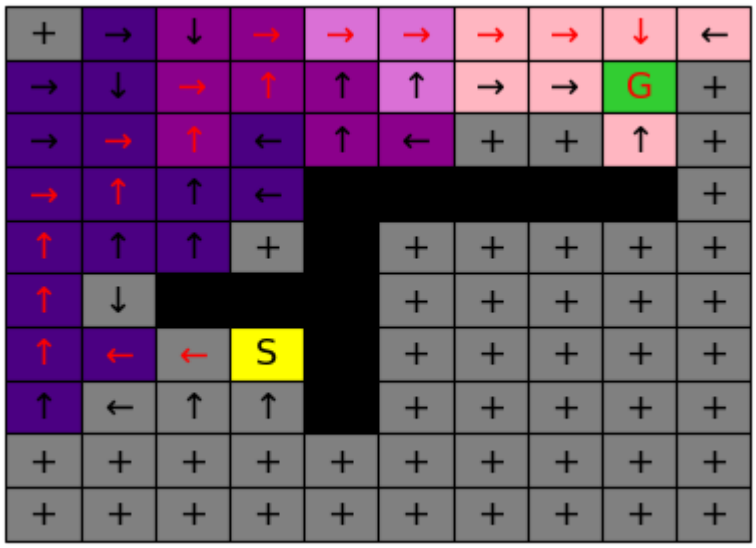
Plot the steps to goal vs training iterations (episodes):

In [21]:
```
# TODO: Plot the steps vs iterations
plot_steps_vs_iters(steps_vs_iters)
```



Visualize the learned greedy policy from the Q values:

In [22]:
```
# TODO: plot the policy from the Q value
plot_policy_from_q(q_hat,env)
```




```
<Figure size 720x720 with 0 Axes>
```

(b) Run your algorithm by passing in a list of 2 goal locations: (1,8) and (5,6). Note: we are using 0-indexing, where (0,0) is top left corner. Report on the results.

In [23]:
```
# TODO: Fill this in (same as before)
num_iters = 200
alpha = 1.0
gamma = 0.9
epsilon = 0.1
max_steps = 100
use_softmax_policy = False

# TODO: Set the goal
goal_locs = [[1,8], [5,6]]
env = MazeEnv(goals=goal_locs)

# TODO: Run Q-learning:
q_hat, steps_vs_iters = qlearn(env, num_iters, alpha, gamma, epsilon, max_step
s, use_softmax_policy)
```

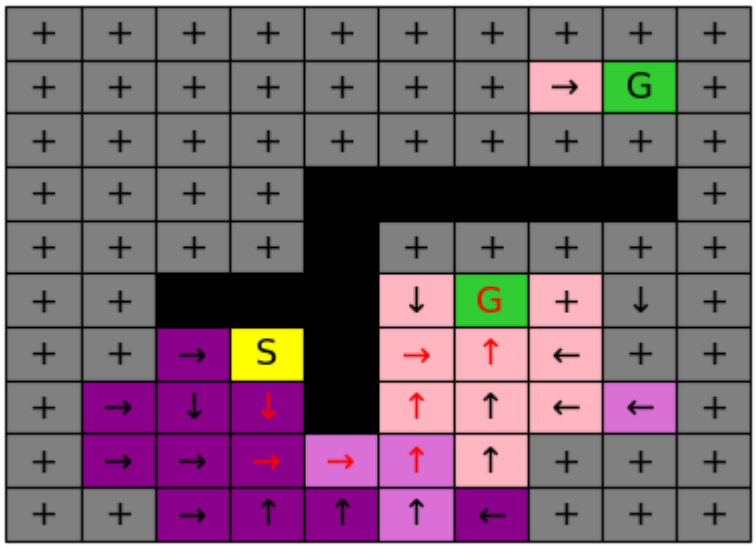
Plot the steps to goal vs training iterations (episodes):

In [24]:
```
# TODO: Plot the steps vs iterations
# plot_steps_vs_iters(...)
```

Plot the steps to goal vs training iterations (episodes): 2.2 Results : The results are as followed: In this experiment, it was demonstrated that the algorithm converged quicker than in our first experiment when we had only the one goal. This may be due to the fact that the two goals were relatively closer to the starting position so it was easier to find the optimal Q each iteration.

plot_steps_vs_iters(steps_vs_iters)

In [25]:
```
# TODO: plot the policy from the Q values
# plot_policy_from_q(...)
plot_policy_from_q(q_hat,env)
```



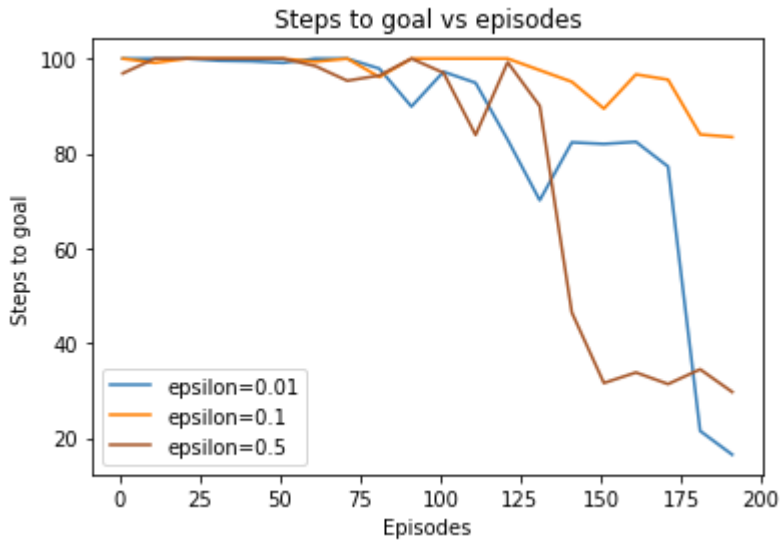

```
<Figure size 720x720 with 0 Axes>
```

2. Experiment with the exploration strategy, in the original environment

(a) Try different ϵ values in ϵ-greedy exploration: We asked you to use a rate of ϵ=10%, but try also 50% and 1%. Graph the results (for 3 epsilon values) and discuss the costs and benefits of higher and lower exploration rates.

In [8]:
```
# TODO: Fill this in (same as before)
#
num_iters = 200
alpha = 1.0
gamma = 0.9
epsilon_list = [0.01, 0.1, 0.5]
max_steps = 100
use_softmax_policy = False


# TODO: set the epsilon lists in increasing order:
# epsilon_list = ...
env = MazeEnv()

steps_vs_iters_list = []
for epsilon in epsilon_list:
    q_hat, steps_vs_iters = qlearn(env, num_iters,
                                   alpha, gamma, epsilon, max_steps,use_softma
x_policy)

    steps_vs_iters_list.append(steps_vs_iters)
```

In [9]:
```
# TODO: Plot the results
label_list = ["epsilon={}".format(eps) for eps in epsilon_list]
plot_several_steps_vs_iters(steps_vs_iters_list, label_list)
```

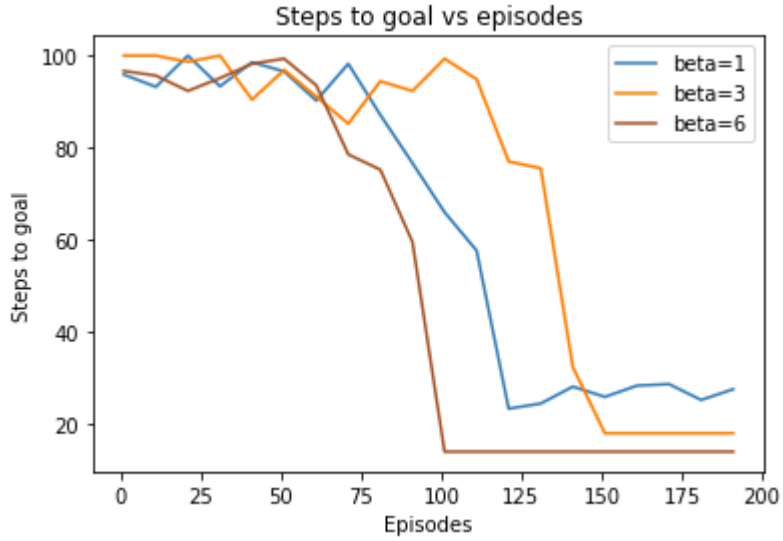


2.3 Results: The exploration rates that are smaller take longer to find the the quickest path, however will more likely find the optimal path. Whereas the exploration rates that are higher will be determine faster path after less episodes, however they may never find the optimal path to the goal (or goals)

(b) Try exploring with policy derived from **softmax of Q-values** described in the Q learning lecture. Use the values of $\beta \in \{1, 3, 6\}$ for your experiment, keeping β fixed throughout the training.

In [12]:
```
# TODO: Fill this in for Static Beta with softmax of Q-values
num_iters = 200
alpha = 1.
gamma = 0.9
epsilon = 0.1
max_steps = 100

# TODO: Set the beta
beta_list = [1, 3, 6]
use_softmax_policy = True
k_exp_schedule = 0 # (float) choose k such that we have a constant beta during
training

env = MazeEnv()
steps_vs_iters_list = []
for beta in beta_list:
    q_hat, steps_vs_iters = qlearn(env, num_iters,
                                   alpha, gamma, epsilon, max_steps,use_softma
x_policy, init_beta=beta, k_exp_sched=k_exp_schedule)
    steps_vs_iters_list.append(steps_vs_iters)
```

In [13]:
```
label_list = ["beta={}".format(beta) for beta in beta_list]
# TODO:
plot_several_steps_vs_iters(steps_vs_iters_list, label_list)
```



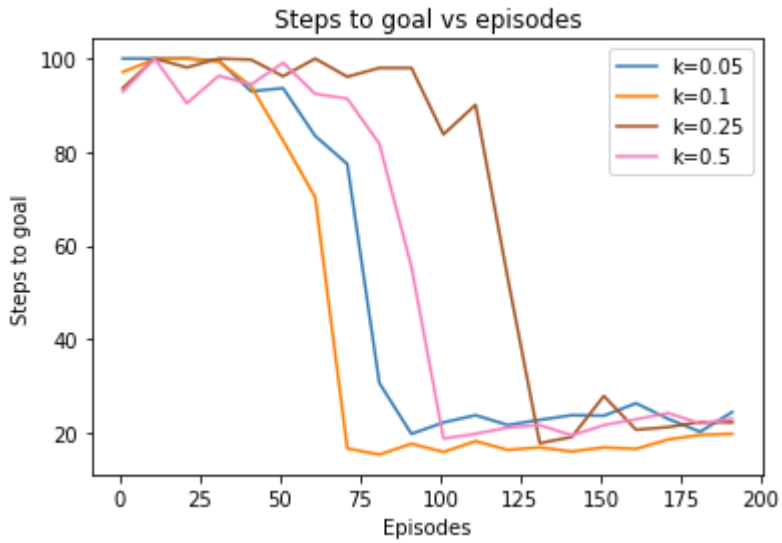
(c) Instead of fixing the $\beta = \beta_0$ to the initial value, we will increase the value of β as the number of episodes t increase:

$$\beta(t) = \beta_0 e^{kt}$$

That is, the β value is fixed for a particular episode. Run the training again for different values of $k \in \{0.05, 0.1, 0.25, 0.5\}$, keeping $\beta_0 = 1.0$. Compare the results obtained with this approach to those obtained with a static β value.

In [14]:
```
# TODO: Fill this in for Dynamic Beta
num_iters = 200
alpha = 1.0
gamma = 0.9
epsilon = 0.1
max_steps = 100
use_softmax_policy = True

# TODO: Set the beta
beta = 1.0
# use_softmax_policy = ...
k_exp_schedule_list = [0.05, 0.1, 0.25, 0.5]
env = MazeEnv()

steps_vs_iters_list = []
for k_exp_schedule in k_exp_schedule_list:
    q_hat, steps_vs_iters = qlearn(env, num_iters,
                                   alpha, gamma, epsilon, max_steps,use_softma
x_policy, init_beta=beta, k_exp_sched=k_exp_schedule)
    steps_vs_iters_list.append(steps_vs_iters)
```

In [15]:
```
# TODO: Plot the steps vs iterations
label_list = ["k={}".format(k_exp_schedule) for k_exp_schedule in k_exp_schedu
le_list]
plot_several_steps_vs_iters(steps_vs_iters_list, label_list)#%% md
## 3. Stochastic Environments
```



Results between static and dynamic beta: When beta is static, and is a smaller value, it will explore more paths because the penality of not finding the goal is not as severe . As we increase the value of beta and it remains static, it is focused more on finding the path to the goal, it starts to act like the greedy algorithm using episilon. Compared to when beta is dynamic, where we are able to gain some exploration steps and then switch over to exploitation where our main focus in reaching the goal. Because there is always a trade off, as k -> 0, beta becomes constant, and hence depending on the initial beta, it will begin to act like the greedy alorgithm, whereas a high k forces large paths to be explore, with less of a focus on the goal.

(a) Make the environment stochastic (uncertain), such that the agent only has a 95% chance of moving in the chosen direction, and has a 5% chance of moving in some random direction.

In [33]:
```
# TODO: Implement ProbabilisticMazeEnv in maze.py
```

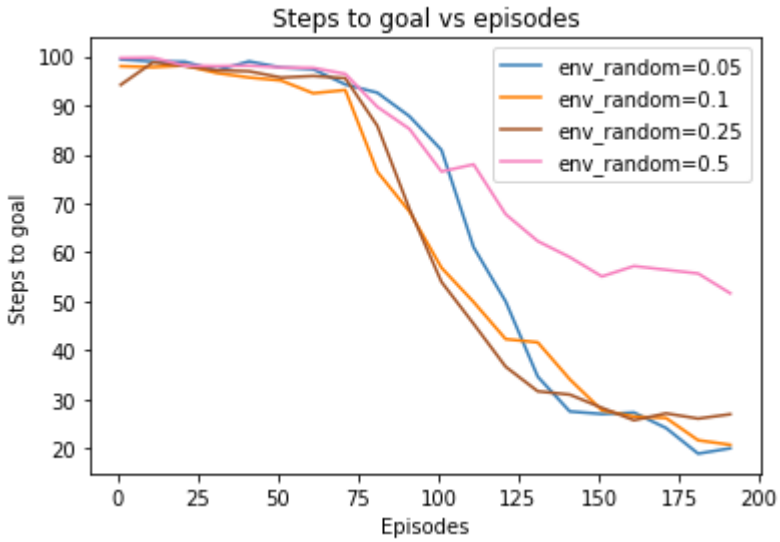
(b) Change the learning rule to handle the non-determinism, and experiment with different probability of environment performing random action $p_{rand} \in \{0.05, 0.1, 0.25, 0.5\}$ in this new rule. How does performance vary as the environment becomes more stochastic?

Results : As it has been shown in the previous experiments, if there is too much randomness and the algorithm is able to choose its path freely, without much penality, it is shown that it cannot converge to an optimal solutions. Furthermore, in the average case when the randomness is not such a high value, it is shown that the behaviour is pretty similar; The graphs show this similarity as they seem to have similar shapes.

Use the same parameters as in first part, except change the alpha (α) value to be **less than 1**, e.g. 0.5.

In [4]:
```
# TODO: Use the same parameters as in the first part, except change alpha
num_iters = 200
alpha = 0.5
gamma = 0.9
epsilon = 0.1
max_steps = 100
use_softmax_policy = False

# Set the environment probability of random
env_p_rand_list = [0.05, 0.1, 0.25, 0.5]

steps_vs_iters_list = []
for env_p_rand in env_p_rand_list:
    # Instantiate with ProbabilisticMazeEnv
    env = ProbabilisticMazeEnv()
    env.p_rand = env_p_rand

    # Note: We will repeat for several runs of the algorithm to make the resul
t less noisy
    avg_steps_vs_iters = np.zeros(num_iters)
    for i in range(10):
        q_hat, steps_vs_iters = qlearn(env, num_iters,
                                       alpha, gamma, epsilon, max_steps,use_softma
x_policy)
        avg_steps_vs_iters += steps_vs_iters
    avg_steps_vs_iters /= 10
    steps_vs_iters_list.append(avg_steps_vs_iters)
```

In [41]:
```
label_list = ["env_random={}".format(env_p_rand) for env_p_rand in env_p_rand_
list]
plot_several_steps_vs_iters(steps_vs_iters_list, label_list)#%% md
# 3. Did you complete the course evaluation?
```



In [3]:
```
#Answer: yes
```

In [ ]:
```

```

```
 1 import numpy as np
 2 import copy
 3 import math
 4
 5 ACTION_MEANING = {
 6     0: "UP",
 7     1: "RIGHT",
 8     2: "LEFT",
 9     3: "DOWN",
10 }
11
12 SPACE_MEANING = {
13     1: "ROAD",
14     0: "BARRIER",
15     -1: "GOAL",
16 }
17
18
19 class MazeEnv:
20
21     def __init__(self, start=[6,3], goals=
   [[1, 8]]):
22         """Deterministic Maze Environment
   """
23
24         self.m_size = 10
25         self.reward = 10
26         self.num_actions = 4
27         self.num_states = self.m_size *
   self.m_size
28
29         self.map = np.ones((self.m_size,
   self.m_size))
```

```
30          self.map[3, 4:9] = 0
31          self.map[4:8, 4] = 0
32          self.map[5, 2:4] = 0
33
34          for goal in goals:
35              self.map[goal[0], goal[1]] = -
    1
36
37          self.start = start
38          self.goals = goals
39          self.obs = self.start
40
41      def step(self, a):
42          """ Perform a action on the
    environment
43
44              Args:
45                  a (int): action integer
46
47              Returns:
48                  obs (list): observation
    list
49                  reward (int): reward for
    such action
50                  done (int): whether the
    goal is reached
51          """
52          done, reward = False, 0.0
53          next_obs = copy.copy(self.obs)
54
55          if a == 0:
56              next_obs[0] = next_obs[0] - 1
57          elif a == 1:
```

```
58              next_obs[1] = next_obs[1] + 1
59          elif a == 2:
60              next_obs[1] = next_obs[1] - 1
61          elif a == 3:
62              next_obs[0] = next_obs[0] + 1
63          else:
64              raise Exception("Action is Not
    Valid")
65
66          if self.is_valid_obs(next_obs):
67              self.obs = next_obs
68
69          if self.map[self.obs[0], self.obs[
   1]] == -1:
70              reward = self.reward
71              done = True
72
73          state = self.get_state_from_coords
   (self.obs[0], self.obs[1])
74
75          return state, reward, done
76
77      def is_valid_obs(self, obs):
78          """ Check whether the observation
   is valid
79
80              Args:
81                  obs (list): observation [x
   , y]
82
83              Returns:
84                  is_valid (bool)
85          """
```

```
 86
 87         if obs[0] >= self.m_size or obs[0
    ] < 0:
 88             return False
 89
 90         if obs[1] >= self.m_size or obs[1
    ] < 0:
 91             return False
 92
 93         if self.map[obs[0], obs[1]] == 0:
 94             return False
 95
 96         return True
 97
 98     @property
 99     def _get_obs(self):
100         """ Get current observation
101         """
102         return self.obs
103
104     @property
105     def _get_state(self):
106         """ Get current observation
107         """
108         return self.get_state_from_coords
    (self.obs[0], self.obs[1])
109
110     @property
111     def _get_start_state(self):
112         """ Get the start state
113         """
114         return self.get_state_from_coords
    (self.start[0], self.start[1])
```

```
115
116     @property
117     def _get_goal_state(self):
118         """ Get the start state
119         """
120         goals = []
121         for goal in self.goals:
122             goals.append(self.
    get_state_from_coords(goal[0], goal[1]))
123         return goals
124
125     def reset(self):
126         """ Reset the observation into
    starting point
127         """
128         self.obs = self.start
129         state = self.
    get_state_from_coords(self.obs[0], self.
    obs[1])
130         return state
131
132     def get_state_from_coords(self, row,
    col):
133         state = row * self.m_size + col
134         return state
135
136     def get_coords_from_state(self, state
    ):
137         row = math.floor(state/self.
    m_size)
138         col = state % self.m_size
139         return row, col
140
```

```
141
142 class ProbabilisticMazeEnv(MazeEnv):
143     """ (Q2.3) Hints: you can refer the
    implementation in MazeEnv
144     """
145
146     def __init__(self, goals=[[2, 8]],
    p_random=0.05):
147         """ Probabilistic Maze
    Environment
148
149             Args:
150                 goals (list): list of
    goals coordinates
151                 p_random (float): random
    action rate
152         """
153
154         super(ProbabilisticMazeEnv, self)
    .__init__()
155         self.goals = goals
156         self.p_rand = p_random
157
158     def step(self, a):
159         done, reward = False, 0.0
160         next_obs = copy.copy(self.obs)
161
162         p = self.p_rand
163         action = np.random.randint(self.
    num_actions) # random action choice
164         a = np.random.choice([a, action],
     1, p=[1-p, p])
165
```

```
166         if a == 0:
167             next_obs[0] = next_obs[0] - 1
168         elif a == 1:
169             next_obs[1] = next_obs[1] + 1
170         elif a == 2:
171             next_obs[1] = next_obs[1] - 1
172         elif a == 3:
173             next_obs[0] = next_obs[0] + 1
174         else:
175             raise Exception("Action is
    Not Valid")
176 
177         if self.is_valid_obs(next_obs):
178             self.obs = next_obs
179 
180         if self.map[self.obs[0], self.obs
    [1]] == -1:
181             reward = self.reward
182             done = True
183 
184         state = self.
    get_state_from_coords(self.obs[0], self.
    obs[1])
185 
186         return state, reward, done
187 
```

```
 1 import numpy as np
 2 import math
 3 import copy
 4
 5
 6 def qlearn(env, num_iters, alpha, gamma,
   epsilon, max_steps, use_softmax_policy,
   init_beta=None, k_exp_sched=None):
 7     """ Runs tabular Q learning algorithm
   for stochastic environment.
 8
 9     Args:
10         env: instance of environment
   object
11         num_iters (int): Number of
   episodes to run Q-learning algorithm
12         alpha (float): The learning rate
   between [0,1]
13         gamma (float): Discount factor,
   between [0,1)
14         epsilon (float): Probability in [0
   ,1] that the agent selects a random move
   instead of
15                 selecting greedily from Q
   value
16         max_steps (int): Maximum number of
    steps in the environment per episode
17         use_softmax_policy (bool): Whether
    to use softmax policy (True) or Epsilon-
   Greedy (False)
18         init_beta (float): If using
   stochastic policy, sets the initial beta
   as the parameter for the softmax
```

```
19          k_exp_sched (float): If using
    stochastic policy, sets hyperparameter for
     exponential schedule
20              on beta
21
22      Returns:
23          q_hat: A Q-value table shaped [
    num_states, num_actions] for environment
    with with num_states
24              number of states (e.g. num
    rows * num columns for grid) and
    num_actions number of possible
25              actions (e.g. 4 actions up/
    down/left/right)
26          steps_vs_iters: An array of size
    num_iters. Each element denotes the number
27              of steps in the environment
    that the agent took to get to the goal
28              (capped to max_steps)
29      """
30      action_space_size = env.num_actions
31      state_space_size = env.num_states
32      q_hat = np.zeros(shape=(
    state_space_size, action_space_size))
33      steps_vs_iters = np.zeros(num_iters)
34
35      for i in range(num_iters):
36          # TODO: Initialize current state
    by resetting the environment
37          curr_state = env.reset()
38          num_steps = 0
39          done = False
40          # TODO: Keep looping while
```

```
40 environment isn't done and less than
   maximum steps
41         while (done == False and num_steps
    < max_steps):
42             num_steps += 1
43             # Choose an action using
   policy derived from either softmax Q-value
44             # or epsilon greedy
45             if use_softmax_policy:
46                 assert (init_beta is not
   None)
47                 assert (k_exp_sched is not
    None)
48                 # assert(k_exp_sched is
   not None)
49                 # TODO: Boltzmann
   stochastic policy
50                 # print(q_hat.shape)
51                 beta = beta_exp_schedule(
   init_beta, num_steps,
52                                          k
   =k_exp_sched)  # Call beta_exp_schedule to
    get the current beta value
53                 action = softmax_policy(
   q_hat, beta, curr_state)
54             else:
55                 # TODO: Epsilon-greedy
56                 # choose the maximum Q-
   value
57                 action = epsilon_greedy(
   q_hat, epsilon, curr_state,
   action_space_size)
58
```

```
59              # TODO: Execute action in the
   environment and observe the next state,
   reward, and done flag
60              next_state, reward, done = env
   .step(action)
61              # print(env._get_goal_state==
   next_state)
62              # TODO: Update Q_value
63              if next_state != curr_state:
64                  # new_value = ...
65                  # TODO: Use Q-learning
   rule to update q_hat for the curr_state
   and action:
66                  # i.e., Q(s,a) <- Q(s,a
   ) + alpha*[reward + gamma * max_a'(Q(s',a
   )) - Q(s,a)]
67                  # print(q_hat[curr_state,
   action] + alpha*(reward + (gamma * (np.max
   (q_hat[next_state,:])-q_hat[curr_state,
   action])))))
68                  q_hat[curr_state, action]
   = q_hat[curr_state, action] + alpha * (
69                                  reward + gamma
    * np.max(q_hat[next_state, :]) - q_hat[
   curr_state, action])
70                  # TODO: Update the current
    staet to be the next state
71                  curr_state = next_state
72          steps_vs_iters[i] = num_steps
73      return q_hat, steps_vs_iters
74
75 def epsilon_greedy(q_hat, epsilon, state,
   action_space_size):
```

```
76      """ Chooses a random action with p_rand_move probability,
77      otherwise choose the action with highest Q value for
78      current observation
79
80      Args:
81          q_hat_3D: A Q-value table shaped [num_rows, num_col, num_actions] for
82                grid environment with num_rows rows and num_col columns and num_actions
83                number of possible actions
84          epsilon (float): Probability in [0,1] that the agent selects a random
85                move instead of selecting greedily from Q value
86          obs: A 2-element array with integer element denoting the row and column
87                that the agent is in
88          action_space_size (int): number of possible actions
89
90      Returns:
91          action (int): A number in the range [0, action_space_size-1]
92                denoting the action the agent will take
93      """
94      # Hint: Sample from a uniform distribution and check if the sample is below
```

```
 95      # a certain threshold
 96      # ...
 97      if np.random.uniform(0,1)<epsilon :
 98          return np.random.randint(
    action_space_size)
 99      elif list(q_hat[state,:])==[0]*
    action_space_size:
100              return np.random.randint(
    action_space_size)
101      return np.argmax(q_hat[state,:])
102
103
104
105 def softmax_policy(q_hat, beta, state):
106      """ Choose action using policy
    derived from Q, using
107      softmax of the Q values divided by
    the temperature.
108
109      Args:
110          q_hat: A Q-value table shaped [
    num_rows, num_col, num_actions] for
111              grid environment with
    num_rows rows and num_col columns
112          beta (float): Parameter for
    controlling the stochasticity of the
    action
113          obs: A 2-element array with
    integer element denoting the row and
    column
114              that the agent is in
115
116      Returns:
```

```
117         action (int): A number in the
    range [0, action_space_size-1]
118             denoting the action the agent
     will take
119         """
120         # TODO: Implement your code here
121         # Hint: use the stable_softmax
    function defined below
122         # ...
123         # print(q_hat[:,3])
124         pr = stable_softmax(beta * q_hat, 1)
125         actions = np.arange(q_hat.shape[1])
126         return np.random.choice(actions, 1, p
    =pr[state, :])
127
128
129 def beta_exp_schedule(init_beta,
    iteration, k=0.1):
130         beta = init_beta * np.exp(k *
    iteration)
131         return beta
132
133
134 def stable_softmax(x, axis=1):
135         """ Numerically stable softmax:
136         softmax(x) = e^x /(sum(e^x))
137                    = e^x / (e^max(x) * sum(e^
    x/e^max(x)))
138
139         Args:
140             x: An N-dimensional array of
    floats
141             axis: The axis for normalizing
```

```
141 over.
142 
143     Returns:
144         output: softmax(x) along the
    specified dimension
145     """
146     max_x = np.max(x, axis, keepdims=True
    )
147     z = np.exp(x - max_x)
148     out = z / np.sum(z, axis, keepdims=
    True)
149     return out
150 
```