1. Unsupervised Learning

In [1]:

```
%matplotlib inline
import scipy
import numpy as np
import itertools
import matplotlib.pyplot as plt
import time
```

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 [179]:

```
# 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]
```

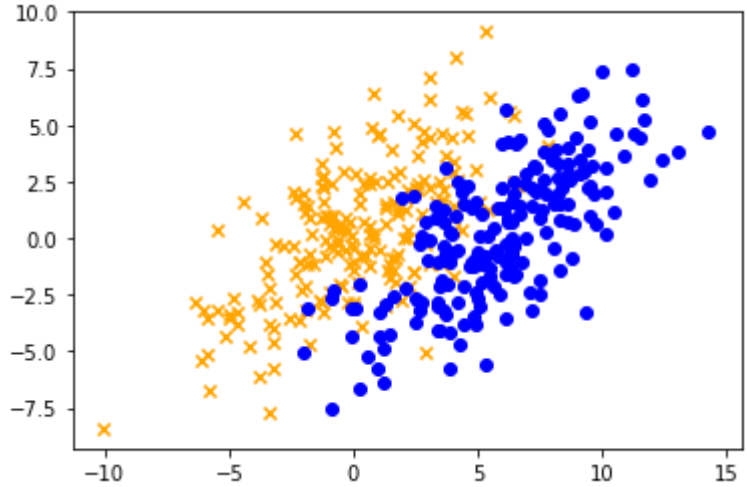
Make a scatter plot of the data points showing the true cluster assignment of each point using different color codes and shape (x for first class and circles for second class):

In [180]:

```
# TODO: Make a scatterplot for the data points showing the true cluster assignments of each
plt.scatter(xy_class1[:, 0], xy_class1[:, 1], marker='x', c='orange')
plt.scatter(xy_class2[:, 0], xy_class2[:, 1], marker='o', c='blue')
```

Out[180]:

```
<matplotlib.collections.PathCollection at 0x1ddc7e845f8>
```



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 [181]:

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

In [182]:

```
# 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 # Number of datapoints and dimension of datapoint
    K = Mu.shape[1] # number of clusters
    r = np.zeros((N, K))

    for k in range(K):
        r[:, k] = np.linalg.norm(data - np.array([Mu[:, k], ] * N), axis=1)**2

    arg_min = np.argmin(r, axis=1) # argmax/argmin along dimension 1
    R_new = np.zeros((N, K)) # Set to zeros/ones with shape (N, K)
    R_new[range(N), arg_min] = 1 # Assign to 1
    return R_new
```

In [183]:

```
# 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 # Number of datapoints and dimension of datapoint
    K = Mu.shape[1]  # number of clusters
    Mu_new = np.zeros((D, K))
    for k in range(K):
        Rk = R[:, k]
        Mu_new[:, k] = np.dot(Rk, data) / np.sum(Rk)
    return Mu_new
```

In [215]:

```
# 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)

costs = []
assign_time = 0;
refit_time = 0;

for it in range(max_iter):
    assign_start = time.time()
    R = km_assignment_step(data, Mu)
    assign_time += time.time() - assign_start

    refit_start = time.time()
    Mu = km_refitting_step(data, R, Mu)
    refit_time += time.time() - refit_start

    c = cost(data, R, Mu)
    costs.append(c)
    print(it, c)

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

```
0 5613.351836728001
1 5318.994008627165
2 5246.14970346552
3 5233.715352943092
4 5229.082541507936
5 5227.82423246978
6 5226.8118522851
7 5226.420112289794
8 5225.719679794525
9 5225.719679794525
10 5225.719679794525
11 5225.719679794525
12 5225.719679794525
13 5225.719679794525
14 5225.719679794525
15 5225.719679794525
16 5225.719679794525
17 5225.719679794525
18 5225.719679794525
19 5225.719679794525
20 5225.719679794525
21 5225.719679794525
22 5225.719679794525
23 5225.719679794525
24 5225.719679794525
25 5225.719679794525
26 5225.719679794525
```

```
27 5225.719679794525
28 5225.719679794525
29 5225.719679794525
30 5225.719679794525
31 5225.719679794525
32 5225.719679794525
33 5225.719679794525
34 5225.719679794525
35 5225.719679794525
36 5225.719679794525
37 5225.719679794525
38 5225.719679794525
39 5225.719679794525
40 5225.719679794525
41 5225.719679794525
42 5225.719679794525
43 5225.719679794525
44 5225.719679794525
45 5225.719679794525
46 5225.719679794525
47 5225.719679794525
48 5225.719679794525
49 5225.719679794525
50 5225.719679794525
51 5225.719679794525
52 5225.719679794525
53 5225.719679794525
54 5225.719679794525
55 5225.719679794525
56 5225.719679794525
57 5225.719679794525
58 5225.719679794525
59 5225.719679794525
60 5225.719679794525
61 5225.719679794525
62 5225.719679794525
63 5225.719679794525
64 5225.719679794525
65 5225.719679794525
66 5225.719679794525
67 5225.719679794525
68 5225.719679794525
69 5225.719679794525
70 5225.719679794525
71 5225.719679794525
72 5225.719679794525
73 5225.719679794525
74 5225.719679794525
75 5225.719679794525
76 5225.719679794525
77 5225.719679794525
78 5225.719679794525
79 5225.719679794525
80 5225.719679794525
81 5225.719679794525
82 5225.719679794525
83 5225.719679794525
84 5225.719679794525
85 5225.719679794525
86 5225.719679794525
87 5225.719679794525
```

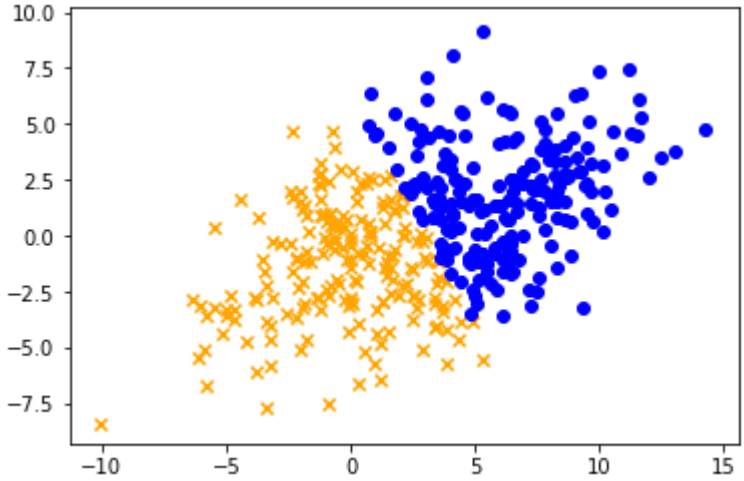
```
88 5225.719679794525
89 5225.719679794525
90 5225.719679794525
91 5225.719679794525
92 5225.719679794525
93 5225.719679794525
94 5225.719679794525
95 5225.719679794525
96 5225.719679794525
97 5225.719679794525
98 5225.719679794525
99 5225.719679794525
```

In [216]:

```
# TODO: Make a scatterplot for the data points showing the K-Means cluster assignments of e
plt.scatter(data[class_1, 0], data[class_1, 1], marker='x', c='orange')
plt.scatter(data[class_2, 0], data[class_2, 1], marker='o', c='blue')
```

Out[216]:

```
<matplotlib.collections.PathCollection at 0x1ddc932dda0>
```

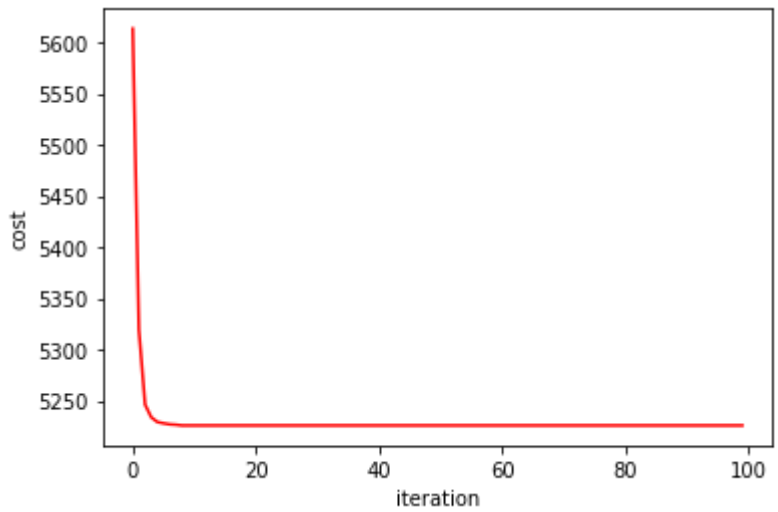


In [217]:

```
plt.xlabel("iteration")
plt.ylabel("cost")

plt.plot(range(max_iter), costs, color='red')
```

Out[217]:

```
[<matplotlib.lines.Line2D at 0x1ddc9392358>]
```



In [218]:

```
expected = np.argmax(R, axis=1)
total_mis = np.sum(np.abs(labels - expected))

print("The misclassification error is {}.".format(total_mis / data.shape[0]))
```

```
The misclassification error is 0.2475.
```

In [220]:

```
print("The total assignment time is {}.".format(assign_time))
print("The total refitting time is {}.".format(refit_time))
```

```
The total assignment time is 0.10485982894897461.
The total refitting time is 0.006563663482666016.
```

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:

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

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 [221]:

```
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 [222]:

```
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 # Number of datapoints and dimension of datapoint
    K = Mu.shape[1] # number of mixtures
    L, T = 0., 0.

    for n in range(N):
        for k in range(K):
            # Compute the likelihood from the k-th Gaussian weighted by the mixing coeffici
            T += Pi[k] * normal_density(data[n], Mu[:, k], Sigma[k])

        L += np.log(T)

    return L
```

In [223]:

```
# 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 # Number of datapoints and dimension of datapoint
    K = Mu.shape[1] # number of mixtures
    Gamma = np.zeros((N, K)) # zeros of shape (N,K), matrix of responsibilities

    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, :]) # Normalize by sum across second dimension (mixt

    return Gamma
```

In [224]:

```
# 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 # Number of datapoints and dimension of datapoint
    K = Gamma.shape[1] # number of mixtures
    Nk = np.sum(Gamma, axis=0) # Sum along first axis
    Mu = np.zeros((D, K))
    Sigma = [None] * K

    for k in range(K):
        Mu[:, k] = (1 / Nk[k]) * np.dot(Gamma[:, k], data)
        func = data - np.array([Mu[:, k]] * N)
        res = np.column_stack([Gamma[:, k]] * D)
        Sigma[k] = (1 / Nk[k]) * np.dot((res * func).T, func)

    Pi = Nk / np.sum(Nk)

    return Mu, Sigma, Pi
```

In [225]:

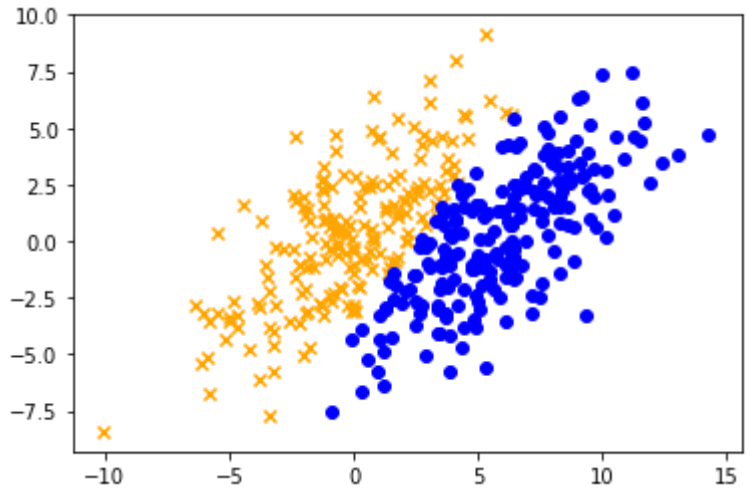
```
# TODO: Run this cell to call the Gaussian Mixture EM algorithm
N, D = data.shape
K = 2
Mu = np.zeros([D, K])
Mu[:, 1] = 1.
Sigma = [np.eye(2), np.eye(2)]
Pi = np.ones(K) / K
Gamma = np.zeros([N, K]) # Gamma is the matrix of responsibilities

max_iter  = 200
logs = []
e_time = 0;
m_time = 0;

for it in range(max_iter):
    e_start = time.time()
    Gamma = gm_e_step(data, Mu, Sigma, Pi)
    e_time += time.time() - e_start

    m_start = time.time()
    Mu, Sigma, Pi = gm_m_step(data, Gamma)
    m_time += time.time() - m_start

    logs.append(log_likelihood(data, Mu, Sigma, Pi))
    # This function makes the computation longer, but good for debugging
    # print(it, log_likelihood(data, Mu, Sigma, Pi))

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

In [226]:

```
# TODO: Make a scatterplot for the data points showing the Gaussian Mixture cluster assignm
plt.scatter(data[class_1, 0], data[class_1, 1], marker='x', c='orange')
plt.scatter(data[class_2, 0], data[class_2, 1], marker='o', c='blue')
```

Out[226]:

```
<matplotlib.collections.PathCollection at 0x1ddc93f9da0>
```

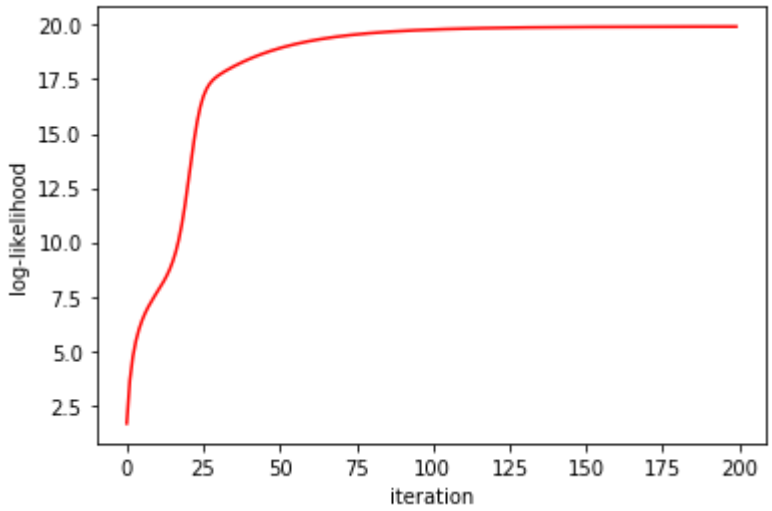


In [227]:

```
plt.xlabel("iteration")
plt.ylabel("log-likelihood")

plt.plot(range(max_iter), logs, color='red')
```

Out[227]:

```
[<matplotlib.lines.Line2D at 0x1ddc94623c8>]
```



In [228]:

```
expected = np.argmax(Gamma, axis=1)
total_mis = np.sum(np.abs(labels - expected))

print("The misclassification error is {}.".format(total_mis / data.shape[0]))
```

```
The misclassification error is 0.1.
```

In [230]:

```
print("The total E-step time is {}.".format(e_time))
print("The total M-step time is {}.".format(m_time))
```

```
The total E-step time is 11.905616760253906.
The total M-step time is 0.1569352149963379.
```

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?

Answer

- By comparing the resulting cluster assignments, we can see that EM performs much better than k-Means. Also the k-Means' misclassification error is often 2-3 times of the EM's misclassification error, which shows that EM is more accurate.
- By comparing the convergence rates, we can see that EM takes a much longer time to converge. For k-Means, the bottleneck is the assignment step. For EM, the bottleneck is the E-step.
- The algorithm performance varies for different realizations of data. But EM always performs better than k-Means.

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 [2]:

```
from qlearning import qlearn
from maze import MazeEnv, ProbabilisticMazeEnv
from plotting_utils import plot_steps_vs_iters, plot_several_steps_vs_iters, plot_policy_fr
```

In [ ]:

```
# %load qlearning.py
import numpy as np
import math
import copy


def qlearn(env, num_iters, alpha, gamma, epsilon, max_steps, \
           use_softmax_policy, init_beta=None, k_exp_sched=None):
    """ Runs tabular Q learning algorithm for stochastic environment.

    Args:
        env: instance of environment object
        num_iters (int): Number of episodes to run Q-learning algorithm
        alpha (float): The learning rate between [0,1]
        gamma (float): Discount factor, between [0,1)
        epsilon (float): Probability in [0,1] that the agent selects a random move instead
                selecting greedily from Q value
        max_steps (int): Maximum number of steps in the environment per episode
        use_softmax_policy (bool): Whether to use softmax policy (True) or Epsilon-Greedy (
        init_beta (float): If using stochastic policy, sets the initial beta as the paramet
        k_exp_sched (float): If using stochastic policy, sets hyperparameter for exponentia
            on beta

    Returns:
        q_hat: A Q-value table shaped [num_states, num_actions] for environment with num_st
            number of states (e.g. num rows * num columns for grid) and num_actions number
            actions (e.g. 4 actions up/down/left/right)
        steps_vs_iters: An array of size num_iters. Each element denotes the number
            of steps in the environment that the agent took to get to the goal
            (capped to max_steps)
    """
    action_space_size = env.num_actions
    state_space_size = env.num_states
    q_hat = np.zeros(shape=(state_space_size, action_space_size))
    steps_vs_iters = np.zeros(num_iters)

    for i in range(num_iters):
        # TODO: Initialize current state by resetting the environment
        curr_state = env.reset()
        num_steps = 0
        done = False

        # TODO: Keep looping while environment isn't done and less than maximum steps
        while not done and num_steps < max_steps:
            num_steps += 1

            # Choose an action using policy derived from either softmax Q-value
            # or epsilon greedy
            if use_softmax_policy:
                assert(init_beta is not None)
                assert(k_exp_sched is not None)
                # TODO: Boltzmann stochastic policy (softmax policy)
                # Call beta_exp_schedule to get the current beta value
                beta = beta_exp_schedule(init_beta, i, k_exp_sched)
                action = softmax_policy(q_hat, beta, curr_state)
            else:
                # TODO: Epsilon-greedy
                action = epsilon_greedy(q_hat, epsilon, curr_state, \
                                        action_space_size)
```

```
            # TODO: Execute action in the environment and observe the next state, reward, a
            next_state, reward, done = env.step(action)

            # TODO: Update Q_value
            if next_state != curr_state:
                new_value = reward + gamma * max(q_hat[next_state]) - \
                q_hat[curr_state, action]
                # TODO: Use Q-learning rule to update q_hat for the curr_state and action:
                # i.e., Q(s,a) <- Q(s,a) + alpha*[reward + gamma * max_a'(Q(s',a')) - Q(s,a
                q_hat[curr_state, action] += alpha * new_value

                # TODO: Update the current state to be the next state
                curr_state = next_state

        steps_vs_iters[i] = num_steps

    return q_hat, steps_vs_iters


def epsilon_greedy(q_hat, epsilon, state, action_space_size):
    """ Chooses a random action with p_rand_move probability,
    otherwise choose the action with highest Q value for
    current observation

    Args:
        q_hat: A Q-value table shaped [num_rows, num_col, num_actions] for
            grid environment with num_rows rows and num_col columns and num_actions
            number of possible actions
        epsilon (float): Probability in [0,1] that the agent selects a random
            move instead of selecting greedily from Q value
        state: A 2-element array with integer element denoting the row and column
            that the agent is in
        action_space_size (int): number of possible actions

    Returns:
        action (int): A number in the range [0, action_space_size-1]
            denoting the action the agent will take
    """
    # TODO: Implement your code here
    # Hint: Sample from a uniform distribution and check if the sample is below
    # a certain threshold
    rand_prob = np.random.uniform()

    if rand_prob < epsilon:
        action = np.random.randint(action_space_size)
    else:
        values = q_hat[state]
        max_value = max(values)
        actions = [i for i in range(action_space_size) if values[i] == max_value]
        action = np.random.choice(actions)

    return action


def softmax_policy(q_hat, beta, state):
    """ Choose action using policy derived from Q, using
    softmax of the Q values divided by the temperature.

    Args:
        q_hat: A Q-value table shaped [num_rows, num_col, num_actions] for
```

```
            grid environment with num_rows rows and num_col columns
        beta (float): Parameter for controlling the stochasticity of the action
        state: A 2-element array with integer element denoting the row and column
            that the agent is in

    Returns:
        action (int): A number in the range [0, action_space_size-1]
            denoting the action the agent will take
    """
    # TODO: Implement your code here
    # Hint: use the stable_softmax function defined below
    x = beta * q_hat[state]
    softmax = stable_softmax(x, 0)
    action = np.random.choice(q_hat.shape[1], p=softmax)

    return action


def beta_exp_schedule(init_beta, iteration, k=0.1):
    beta = init_beta * np.exp(k * iteration)
    return beta


def stable_softmax(x, axis=2):
    """ Numerically stable softmax:
    softmax(x) = e^x /(sum(e^x))
               = e^x / (e^max(x) * sum(e^x/e^max(x)))

    Args:
        x: An N-dimensional array of floats
        axis: The axis for normalizing over.

    Returns:
        output: softmax(x) along the specified dimension
    """
    max_x = np.max(x, axis, keepdims=True)
    z = np.exp(x - max_x)
    output = z / np.sum(z, axis, keepdims=True)

    return output
```

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.

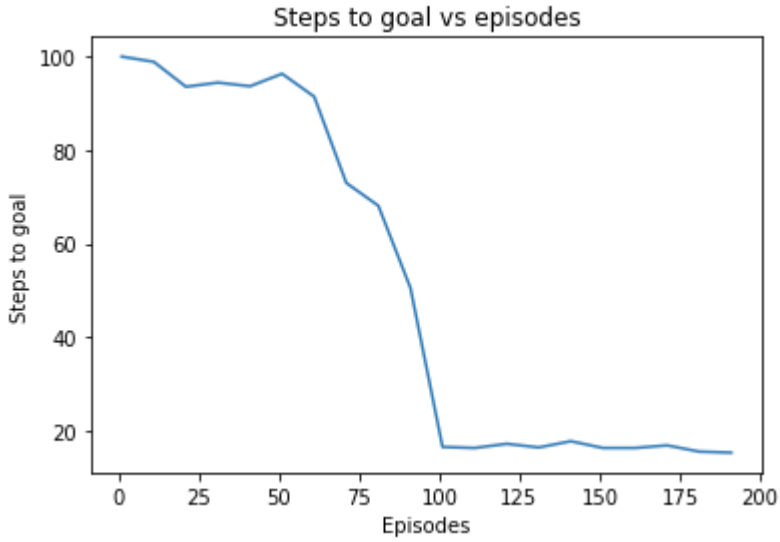
In [6]:

```
# 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:
q_hat, steps_vs_iters = qlearn(env, num_iters, alpha, gamma, epsilon, \
                               max_steps, use_softmax_policy)
```

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

In [7]:

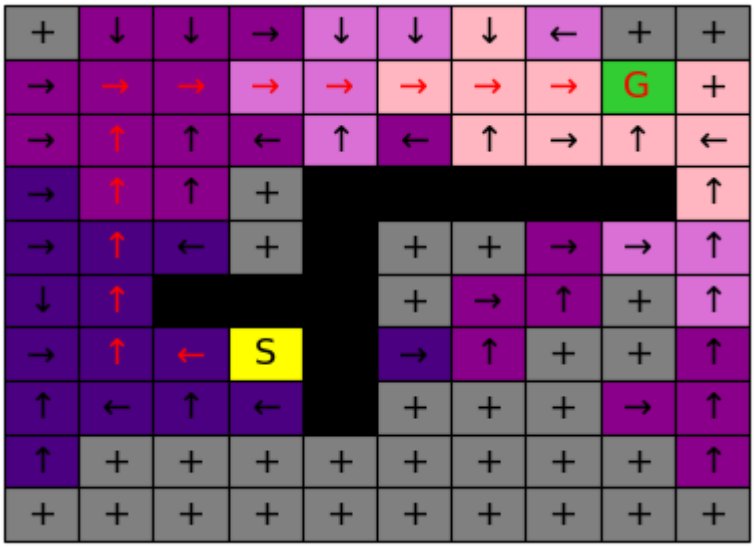
```
# TODO: Plot the steps vs iterations
plot_steps_vs_iters(steps_vs_iters)
```



Visualize the learned greedy policy from the Q values:

In [8]:

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




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

Summary

- Given a goal location, the algorithm returns the optimal path to the goal location. But it cannot always give the shortest path, more iterations or a higher epsilon may be needed.

(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.

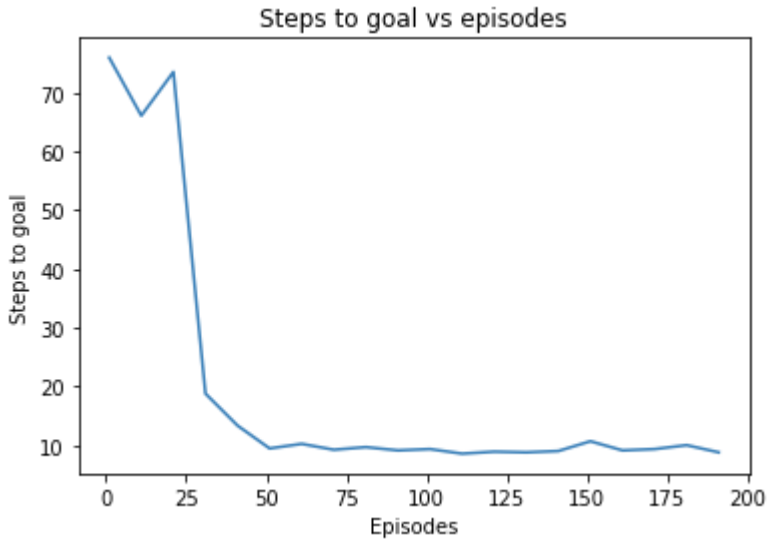
In [9]:

```
# 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_steps, use_softmax_policy)
```

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

In [10]:

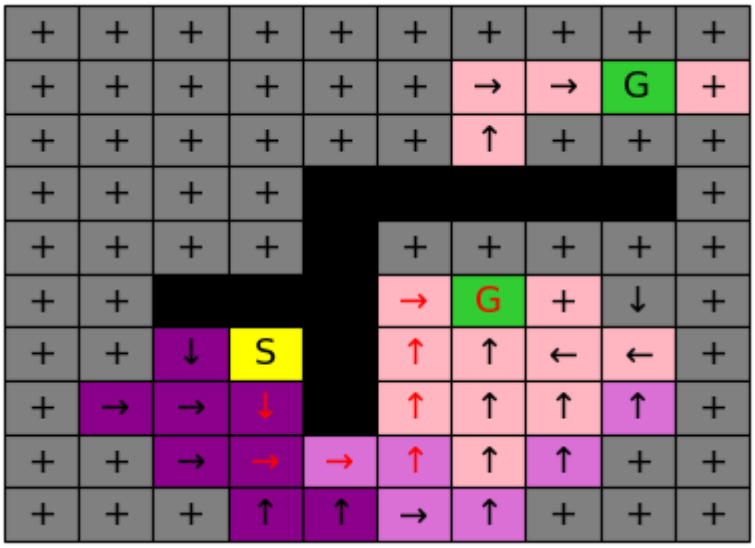
```
# TODO: Plot the steps vs iterations
plot_steps_vs_iters(steps_vs_iters)
```



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

In [11]:

```
# TODO: plot the policy from the Q values
plot_policy_from_q(q_hat, env)
```




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

Summary

- Given two different goal locations, the algorithm returns the shortest path to the nearest goal location.

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

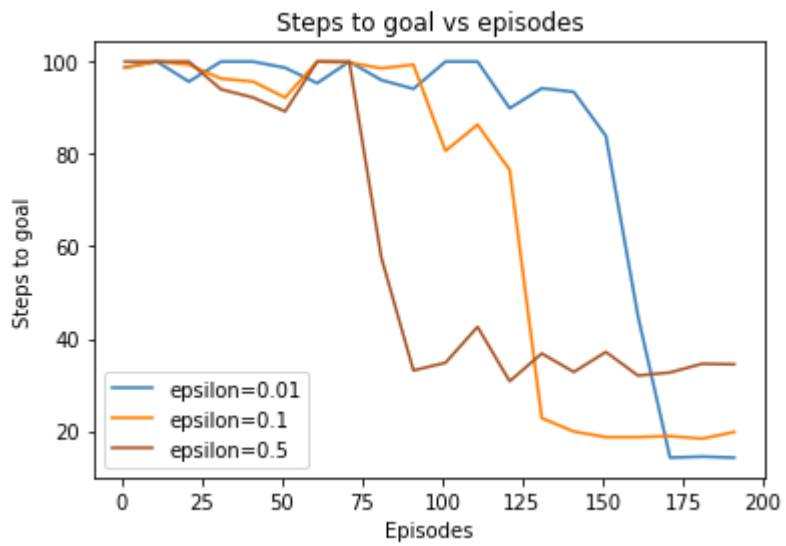
(a) Try different ϵ values in ϵ-greedy exploration: We asked you to use a rate of ϵ=10%, but try also 50% and 1%. Graph the results (for 3 epsilon values) and discuss the costs and benefits of higher and lower exploration rates.

In [18]:

```
# TODO: Fill this in (same as before)
num_iters = 200
alpha = 1.0
gamma = 0.9
max_steps = 100
use_softmax_policy = False

# TODO: set the epsilon lists in increasing order:
epsilon_list = [0.01, 0.1, 0.5]

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_softmax_policy)
    steps_vs_iters_list.append(steps_vs_iters)
```

In [19]:

```
# 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)
```



Summary

- Given different epsilons. The algorithm's performance varies a lot.
- For higher exploration rate, the steps drops much earlier. But when it converges, the steps is the highest.
- For lower exploration rate, the steps drops very late. But when it converges, the steps is the lowest.

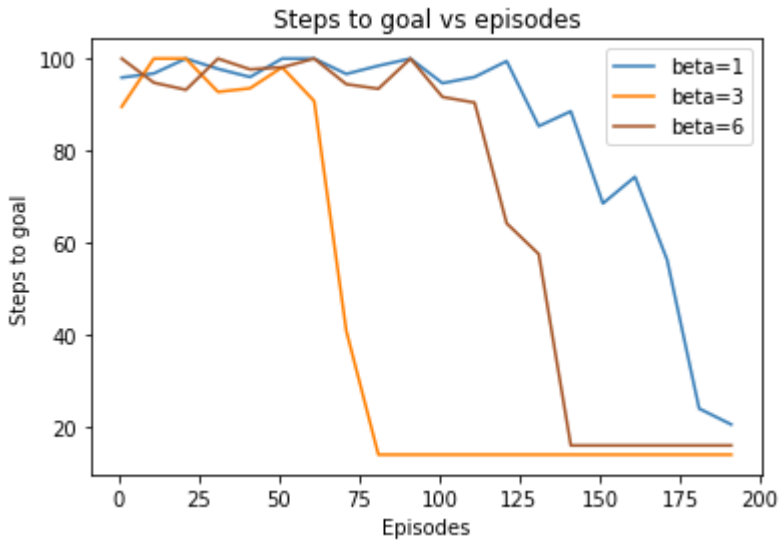
(b) Try exploring with policy derived from **softmax of Q-values** described in the Q learning lecture. Use the values of $\beta \in \{1, 3, 6\}$ for your experiment, keeping β fixed throughout the training.

In [38]:

```
# TODO: Fill this in for Static Beta with softmax of Q-values
num_iters = 200
alpha = 1.0
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 = 0.0 # (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_softmax_policy, beta, k_exp_schedule)
    steps_vs_iters_list.append(steps_vs_iters)
```

In [39]:

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



Summary

- Using softmax policy. Given different betas and keep beta fixed throughout the training. The algorithm's performance varies a lot.
- Beta that is higher or lower takes longer time to converge and the steps when converges is higher. Beta = 3 seems to a better choice.

(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:

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

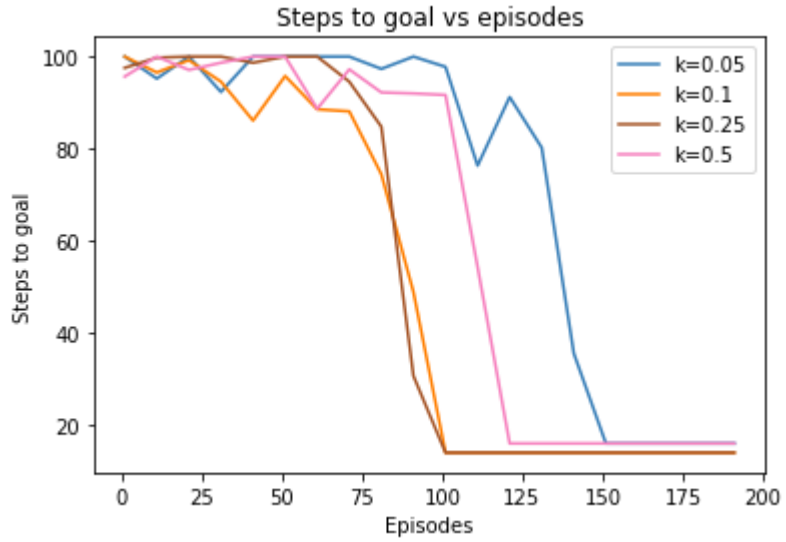
That is, the β value is fixed for a particular episode. Run the training again for different values of $k \in \{0.05, 0.1, 0.25, 0.5\}$, keeping $\beta_0 = 1.0$. Compare the results obtained with this approach to those obtained with a static β value.

In [56]:

```
# TODO: Fill this in for Dynamic Beta
num_iters = 200
alpha = 1.0
gamma = 0.9
epsilon = 0.1
max_steps = 100

# TODO: Set the beta
beta = 1.0
use_softmax_policy = True
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_softmax_policy, beta, k_exp_schedule)
    steps_vs_iters_list.append(steps_vs_iters)
```

In [57]:

```
# TODO: Plot the steps vs iterations
label_list = ["k={}".format(k_exp_schedule) for k_exp_schedule in k_exp_schedule_list]
plot_several_steps_vs_iters(steps_vs_iters_list, label_list)
```



Summary

- Given different ks. The algorithm's performance varies.
- k that is higher or lower takes longer time to converge and the steps when converges is higher. k = 0.25 seems to a better choice.
- Comparing the results obtained with this approach to those obtained with a static beta value (e.g. beta = 1 from the previous graph): increasing the value of beta as the number of episodes t increase has a better performance.

3. Stochastic Environments

(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 [ ]:

```
# %load maze.py
import numpy as np
import copy
import math

ACTION_MEANING = {
    0: "UP",
    1: "RIGHT",
    2: "LEFT",
    3: "DOWN",
}

SPACE_MEANING = {
    1: "ROAD",
    0: "BARRIER",
    -1: "GOAL",
}


class MazeEnv:

    def __init__(self, start=[6, 3], goals=[[1, 8]]):
        """Deterministic Maze Environment"""

        self.m_size = 10
        self.reward = 10
        self.num_actions = 4
        self.num_states = self.m_size * self.m_size

        self.map = np.ones((self.m_size, self.m_size))
        self.map[3, 4:9] = 0
        self.map[4:8, 4] = 0
        self.map[5, 2:4] = 0

        for goal in goals:
            self.map[goal[0], goal[1]] = -1

        self.start = start
        self.goals = goals
        self.obs = self.start

    def step(self, a):
        """ Perform a action on the environment

            Args:
                a (int): action integer

            Returns:
                obs (list): observation list
                reward (int): reward for such action
                done (int): whether the goal is reached
        """
        done, reward = False, 0.0
        next_obs = copy.copy(self.obs)

        if a == 0:
            next_obs[0] = next_obs[0] - 1
        elif a == 1:
            next_obs[1] = next_obs[1] + 1
```

```
        elif a == 2:
            next_obs[1] = next_obs[1] - 1
        elif a == 3:
            next_obs[0] = next_obs[0] + 1
        else:
            raise Exception("Action is Not Valid")

        if self.is_valid_obs(next_obs):
            self.obs = next_obs

        if self.map[self.obs[0], self.obs[1]] == -1:
            reward = self.reward
            done = True

        state = self.get_state_from_coords(self.obs[0], self.obs[1])

        return state, reward, done

    def is_valid_obs(self, obs):
        """ Check whether the observation is valid

            Args:
                obs (list): observation [x, y]

            Returns:
                is_valid (bool)
        """

        if obs[0] >= self.m_size or obs[0] < 0:
            return False

        if obs[1] >= self.m_size or obs[1] < 0:
            return False

        if self.map[obs[0], obs[1]] == 0:
            return False

        return True

    @property
    def _get_obs(self):
        """ Get current observation
        """
        return self.obs

    @property
    def _get_state(self):
        """ Get current observation
        """
        return self.get_state_from_coords(self.obs[0], self.obs[1])

    @property
    def _get_start_state(self):
        """ Get the start state
        """
        return self.get_state_from_coords(self.start[0], self.start[1])

    @property
    def _get_goal_state(self):
        """ Get the start state
        """
```

```
        goals = []
        for goal in self.goals:
            goals.append(self.get_state_from_coords(goal[0], goal[1]))
        return goals

    def reset(self):
        """ Reset the observation into starting point
        """
        self.obs = self.start
        state = self.get_state_from_coords(self.obs[0], self.obs[1])
        return state

    def get_state_from_coords(self, row, col):
        state = row * self.m_size + col
        return state

    def get_coords_from_state(self, state):
        row = math.floor(state/self.m_size)
        col = state % self.m_size
        return row, col


class ProbabilisticMazeEnv(MazeEnv):
    """ (Q2.3) Hints: you can refer the implementation in MazeEnv
    """

    def __init__(self, goals=[[2, 8]], p_random=0.05):
        """ Probabilistic Maze Environment

            Args:
                goals (list): list of goals coordinates
                p_random (float): random action rate
        """

        super().__init__(goals=goals)
        self.p_random = p_random

    def step(self, a):
        rand_prob = np.random.uniform()

        if rand_prob < self.p_random:
            a = np.random.randint(self.num_actions)

        return super().step(a)
```

(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?

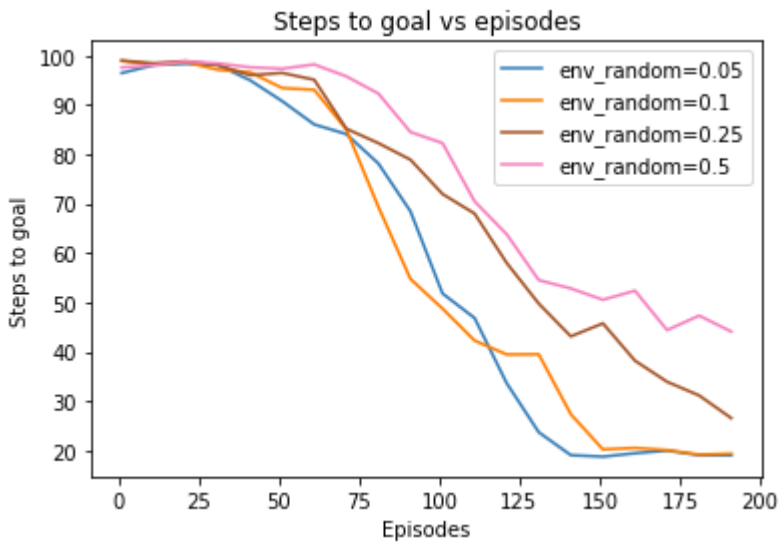
Use the same parameters as in first part, except change the alpha (α) value to be **less than 1**, e.g. 0.5.

In [6]:

```
# 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(p_random=env_p_rand)

    # Note: We will repeat for several runs of the algorithm to make the result 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_softmax_policy)
        avg_steps_vs_iters += steps_vs_iters
    avg_steps_vs_iters /= 10
    steps_vs_iters_list.append(avg_steps_vs_iters)
```

In [7]:

```
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)
```



Summary

- Given different env_random(s). The algorithm's performance varies slightly.
- As the environment becomes more stochastic, the algorithm performs worse. A very small env_random seems to better a choice.

3. Did you complete the course evaluation?

Yes.

UNIVERSITY OF TORONTO

Your List of Course Evaluations to Complete

Task Owner: Zhuozi Zou
Project Title: FAS Fall 2019 Undergrad
Category: Arts and Science
Subcategory: 2019 Fall

Subject	Due date	Status
Intro to Soft Eng CSC301H1-F-LEC0101	Friday, December 6, 2019	Completed
Intro Machine Learning CSC311H1-F-LEC0201	Friday, December 6, 2019	Completed
Prog Languages CSC324H1-F-LEC0101	Friday, December 6, 2019	Completed
Operating Systems CSC369H1-F-LEC5101	Friday, December 6, 2019	Completed

Mobile Version | Standard Version

Powered by blue

In [ ]:

```

```