1. Unsupervised Learning

In [1]:

```
%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 [2]:

```
# 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]

#print(data)
#print(data == data_full[:, 0:2])
#print(x_class1)
#print(x_class1[:,1])
```

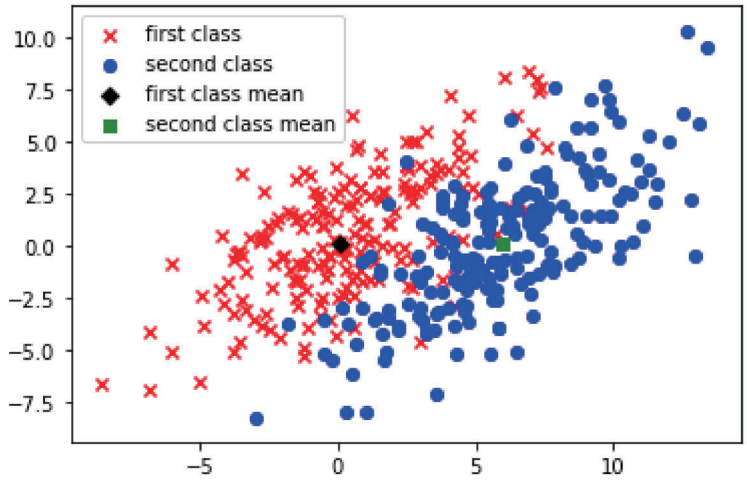
Make a scatter plot of the data points showing the true cluster assignment of each point using different color codes and shape (x for first class and circles for second class):

In [3]:

```
# TODO: Make a scatterplot for the data points showing the true cluster assignments of
 each point
#print(data)
plt.figure()
plt.scatter(xy_class1[:,0], xy_class1[:,1], marker = "x", c = 'red', label = "first cla
ss") # first class, x shape
plt.scatter(xy_class2[:,0], xy_class2[:,1], marker = "o", c = 'blue', label = "second c
lass") # second class, circle shape
plt.scatter(mean_1[0], mean_1[1], marker = "D", c = 'black', label = "first class mean"
)
plt.scatter(mean_2[0], mean_2[1], marker = "s", c = 'green', label = "second class mea
n")
plt.legend()
```

Out[3]:

```
<matplotlib.legend.Legend at 0x2a408821148>
```



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 [8]:

```
def cost(data, R, Mu):
    N, D = data.shape

    #print(Mu)
    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 [9]:

```
# 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.empty([N, K]) #a matrix of NxK dimension for the distances of the the N data
points to each of the K cluster centers.
    for k in range(K):
        r[:, k] = np.linalg.norm(Mu[:,k] - data, 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)

  # Piazza: If you initialize with zeros and use argmax, you just need to invert (e.g.
 1-argmax) the indices and then plug in to R_new.
   # print(np.linalg.norm(Mu[:,k] - data)**2)
    for i in range(len(arg_min)):
        R_new[i, arg_min[i]] = 1 # Assign to 1
    return R_new
```

In [10]:

```
# 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.transpose(data).dot(R)/np.sum(R, axis=0)
    return Mu_new
```

In [11]:

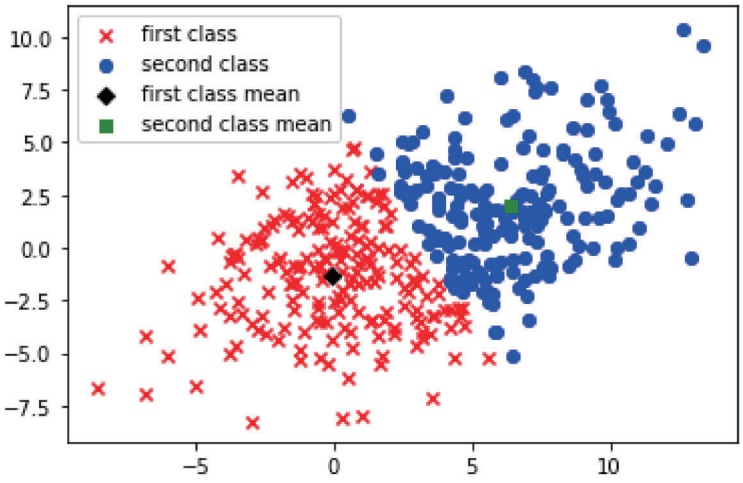
```
# TODO: Run this cell to call the K-means algorithm
N, D = data.shape
K = 2
max_iter = 100
class_init = np.random.binomial(1., .5, size=N)
R = np.vstack([class_init, 1 - class_init]).T


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

num_iter = []
cost_per_iter = []

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))
    num_iter.append(it)
    cost_per_iter.append(cost(data, R, Mu))

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

class_1_points = []
class_2_points = []

for i in class_1:
    class_1_points.append(data[i])

for i in class_2:
    class_2_points.append(data[i])
```

In [12]:

```
# TODO: Make a scatterplot for the data points showing the K-Means cluster assignments
 of each point
plt.figure()

plt.scatter(class_1_points[0][:,0], class_1_points[0][:,1], marker = "x", c = 'red', la
bel = "first class") # first class, x shape
plt.scatter(class_2_points[0][:,0], class_2_points[0][:,1], marker = "o", c = 'blue', l
abel = "second class") # second class, circle shape

plt.scatter(Mu[:,0][0], Mu[:,0][1], marker = "D", c = 'black', label = "first class mea
n")
plt.scatter(Mu[:,1][0], Mu[:,1][1], marker = "s", c = 'green', label = "second class me
an")
plt.legend()
```

Out[12]:

```
<matplotlib.legend.Legend at 0x2a4095ae8c8>
```



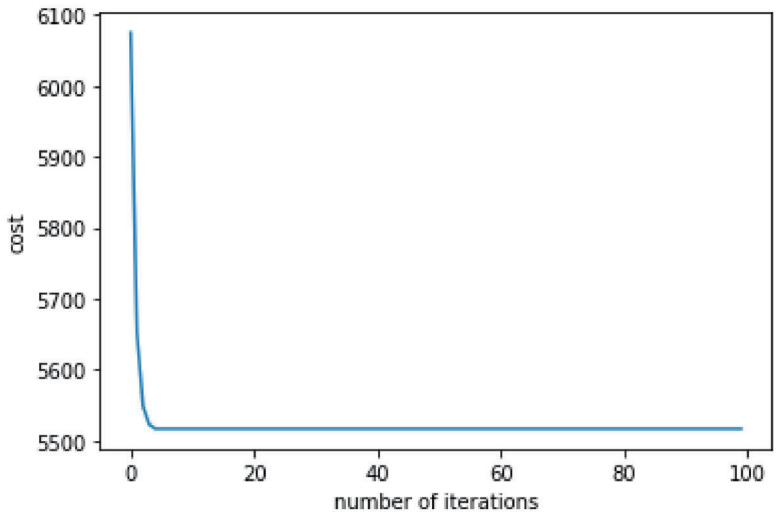
In [13]:

```
count_1 = 0
count_2 = 0
#print(x_class1)
for i in class_1_points[0]:
    if i in x_class1:
        count_1 += 1

#print(count_1)

for i in class_2_points[0]:
    if i in x_class2:
        count_2 += 1
#print(count_2)

accuracy = (count_1+count_2) / N
error = 1 - accuracy
print("Misclassification error: " + str(error))

plt.figure()
plt.plot(num_iter, cost_per_iter)
plt.xlabel('number of iterations')
plt.ylabel('cost')
```

```
Misclassification error: 0.255
```

Out[13]:

```
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 [14]:

```
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 [15]:

```
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 =  0.
    for n in range(N):
        T = 0
        for k in range(K):
            T += Pi[k] * normal_density(data[n,:], Mu[:,k], Sigma[k]) # Compute the lik
elihood from the k-th Gaussian weighted by the mixing coefficients
        L += np.log(T)
    return L
```

In [16]:

```
# 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 (mix
tures)
    return Gamma
```

In [17]:

```
# 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.dot(data.T, Gamma)/Nk
    Sigma = np.zeros([K,D,D])

    for k in range(K):
        sig = np.zeros([2,2])
        for i in range(N):
            sig += Gamma[i,k] * np.outer((data[i]-Mu[:,k]), (np.matrix(data[i] - Mu[:,k
])).T)
        Sigma[k] = sig/Nk[k]
    Pi = Nk/N
    return Mu, Sigma, Pi
```

In [18]:

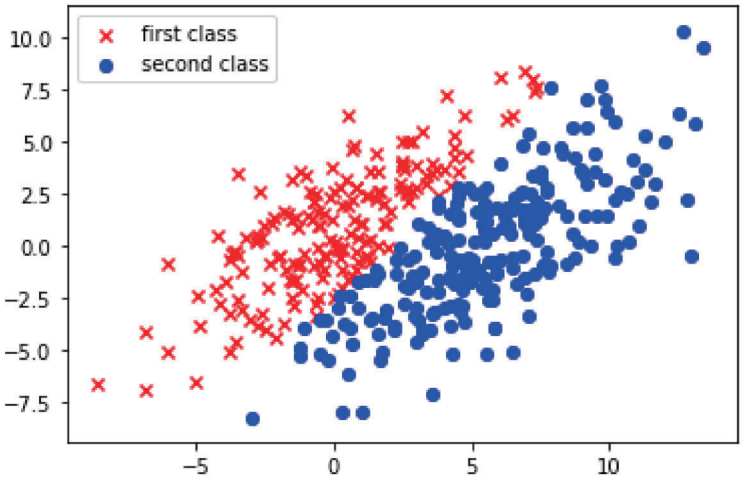
```
# TODO: Run this cell to call the Gaussian Mixture EM algorithm
N, D = data.shape
K = 2
Mu = np.zeros([D, K])
Mu[:, 1] = 1.
Sigma = [np.eye(2), np.eye(2)]
Pi = np.ones(K) / K
Gamma = np.zeros([N, K]) # Gamma is the matrix of responsibilities

max_iter  = 200


num_iter_log = []
log_likelihood_per_iter = []

for it in range(max_iter):
    Gamma = gm_e_step(data, Mu, Sigma, Pi)
    Mu, Sigma, Pi = gm_m_step(data, Gamma)
    # print(it, log_likelihood(data, Mu, Sigma, Pi)) # This function makes the computat
ion longer, but good for debugging
    num_iter_log.append(it)
    log_likelihood_per_iter.append(log_likelihood(data, Mu, Sigma, Pi))

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

class_1_points = []
class_2_points = []

for i in class_1:
    class_1_points.append(data[i])

for i in class_2:
    class_2_points.append(data[i])
```

In [19]:

```
# TODO: Make a scatterplot for the data points showing the Gaussian Mixture cluster ass
ignments of each point
plt.figure()
plt.scatter(class_1_points[0][:,0], class_1_points[0][:,1], marker = "x", c = 'red', la
bel = "first class") # first class, x shape
plt.scatter(class_2_points[0][:,0], class_2_points[0][:,1], marker = "o", c = 'blue', l
abel = "second class") # second class, circle shape
plt.legend()
```

Out[19]:

```
<matplotlib.legend.Legend at 0x2a4096a3348>
```



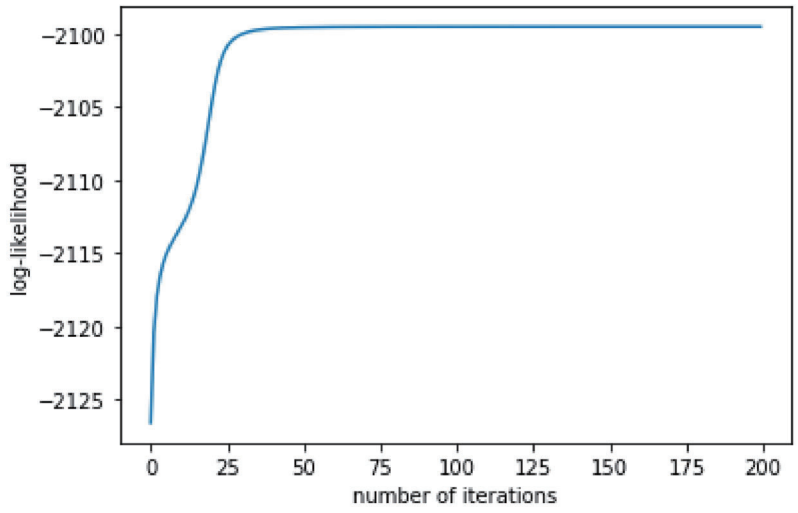
In [20]:

```
count_1 = 0
count_2 = 0
#print(x_class1)
for i in class_1_points[0]:
    if i in x_class1:
        count_1 += 1

#print(count_1)

for i in class_2_points[0]:
    if i in x_class2:
        count_2 += 1
#print(count_2)

accuracy = (count_1+count_2) / N
error = 1 - accuracy
print("Misclassification error:" + str(error))

plt.plot(num_iter_log, log_likelihood_per_iter)
plt.xlabel('number of iterations')
plt.ylabel('log-likelihood')
```

```
Misclassification error:0.10499999999999998
```

Out[20]:

```
Text(0, 0.5, 'log-likelihood')
```



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?

Comments:

- From the above models for k-Means and EM, we can see that EM algorithm performs significantly better in terms of cluster assignments. This is also reflected in a lower misclassification rate. We have a misclassification error of EM at 0.10 but an error rate of 0.255 for k-Means. This is also reflected in the visual representations of the splits, where the splitting of two classes for EM is much more similar to that of the original data.
- However, in terms of convergence rate, we can see that k-Means performs better. Based on the plots for convergence in terms of number of iterations, we can see that k_Means converges at around less than 5 iterations. For EM, the plot converges at around 30 iterations. This is especially a key and decisive factor when performing classification on a larger data set, since on my local machine, performing EM with default hyper paramters have already caused a time duration for 3-5 seconds (while k-Means finished performing almost instantly). So tradeoff or bottleneck for EM is to gain high accuracy but result in higher time complexity as well, while for k-Means, we have a worse performance in exchange for a faster time complexity.
- In general, my findings were consistent (as shown below). The percentage difference for EM accuracy is almost always significantly higher. With my data (where I randomize sample_size, means, and covariance), my results (of percentage difference between the two algorithms on the same data) were as follows [20.500000000000007, 7.333333333333336, -3.749999999999998, 10.799999999999999, 23.83333333333333]. We can see that the only outlier was the 3rd data realization with K-means performing better in terms of accuracy with about 0.3%. This might be due to the selection of means, since I chose the mean of 5 for both dimensions for both data set, which might make the data to be clustered together. This is also reinforced by the fact that both models resulted in a very high misclassification rate (about 40%). In general, since EM aglorithm soft assigns a point to a certain clusters using softmax probability (instead of hard assignments used by k-Means), it is supposed to perform better in terms of accuracy as k-Means might be more biased towards certain "shapes" of data clustering. The only drawback is the longer time complexity required for the classification.

In [71]:

```
def samples_em(num_samples_1, cov_1, mean_1_1, mean_2_1):
    x_class1_1 = np.random.multivariate_normal(mean_1_1, cov_1, num_samples_1 // 2)
    x_class2_1 = np.random.multivariate_normal(mean_2_1, cov_1, num_samples_1 // 2)
    xy_class1_1 = np.column_stack((x_class1_1, np.zeros(num_samples_1 // 2)))
    xy_class2_1 = np.column_stack((x_class2_1, np.ones(num_samples_1 // 2)))
    data_full_1 = np.row_stack([xy_class1_1, xy_class2_1])
    np.random.shuffle(data_full_1)
    data_1 = data_full_1[:, :2]
    labels_1 = data_full_1[:, 2]

    N, D = data_1.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
    num_iter_log = []
    log_likelihood_per_iter = []

    for it in range(max_iter):
        Gamma = gm_e_step(data_1, Mu, Sigma, Pi)
        Mu, Sigma, Pi = gm_m_step(data_1, Gamma)
        # print(it, log_likelihood(data, Mu, Sigma, Pi)) # This function makes the comp
utation longer, but good for debugging
        num_iter_log.append(it)
        log_likelihood_per_iter.append(log_likelihood(data_1, Mu, Sigma, Pi))
    class_1 = np.where(Gamma[:, 0] >= .5)
    class_2 = np.where(Gamma[:, 1] >= .5)
    class_1_points = []
    class_2_points = []
    for i in class_1:
        class_1_points.append(data_1[i])
    for i in class_2:
        class_2_points.append(data_1[i])
    count_1 = 0
    count_2 = 0
    #print(x_class1)
    for i in class_1_points[0]:
        if i in x_class1_1:
            count_1 += 1
    #print(count_1)
    for i in class_2_points[0]:
        if i in x_class2_1:
            count_2 += 1
    #print(count_2)

    accuracy = (count_1+count_2) / N
    error = 1 - accuracy
    return error
```

In [79]:

```
def samples_km(num_samples_1, cov_1, mean_1_1, mean_2_1):
    x_class1_1 = np.random.multivariate_normal(mean_1_1, cov_1, num_samples_1 // 2)
    x_class2_1 = np.random.multivariate_normal(mean_2_1, cov_1, num_samples_1 // 2)
    xy_class1_1 = np.column_stack((x_class1_1, np.zeros(num_samples_1 // 2)))
    xy_class2_1 = np.column_stack((x_class2_1, np.ones(num_samples_1 // 2)))
    data_full_1 = np.row_stack([xy_class1_1, xy_class2_1])
    np.random.shuffle(data_full_1)
    data_1 = data_full_1[:, :2]
    labels_1 = data_full_1[:, 2]

    N, D = data_1.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_1), np.sum(R, axis=0)

    num_iter = []
    cost_per_iter = []

    for it in range(max_iter):
        R = km_assignment_step(data_1, Mu)
        Mu = km_refitting_step(data_1, R, Mu)
    #    print(it, cost(data, R, Mu))
        num_iter.append(it)
        cost_per_iter.append(cost(data_1, R, Mu))

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

    class_1_points = []
    class_2_points = []

    for i in class_1:
        class_1_points.append(data_1[i])

    for i in class_2:
        class_2_points.append(data_1[i])
        count_1 = 0
    count_2 = 0
    #print(x_class1)
    for i in class_1_points[0]:
        if i in x_class1_1:
            count_1 += 1

    #print(count_1)

    for i in class_2_points[0]:
        if i in x_class2_1:
            count_2 += 1
    #print(count_2)

    accuracy = (count_1+count_2) / N
    error = 1 - accuracy
    return error
```

```
#samples_km(num_samples_1, cov_1, mean_1_1, mean_2_1)
```

In [80]:

```
num_samples_1 = 200
#cov_1 = np.array([[.1, .3], [.7, .4]]) * 10
cov = np.array([[1., .7], [.7, 1.]]) * 10
cov_1 = np.array([[1., 0.7], [0.7, 1.]]) * 10
# cov_1 = cov
mean_1_1 = [.1, 1]
mean_2_1 = [6., .1]

num_samples_list = [200, 300, 400, 500, 600]
cov_list = [np.array([[1., .7], [.7, 1.]]) * 10, np.array([[1., 0.6], [0.6, 1.]]) * 10,
           np.array([[1., 0.7], [0.7, 1.]]) * 10, np.array([[1., 0.8], [0.8, 1.]]) * 10
,
           np.array([[1., 0.9], [0.9, 1.]]) * 10]
mean_1_list = [[.1, 1], [4, .1], [5, 5], [-4, .7], [5., .3]]
mean_2_list = [[6., .1], [3, 9], [5, 5], [4, .9], [10., 1.]]

em_error_list = []
km_error_list = []
# num_samples = 400
# cov = np.array([[1., .7], [.7, 1.]]) * 10
# mean_1 = [.1, .1]
# mean_2 = [6., .1]
# print(samples_em(num_samples, cov, mean_1, mean_2))

i = 0;
while i < 5:
#     print(i)

    num_samples_1 = num_samples_list[i]
    cov_1 = cov_list[i]
    mean_1_1 = mean_1_list[i]
    mean_2_1 = mean_2_list[i]

    em_error = samples_em(num_samples_1, cov_1, mean_1_1, mean_2_1)
    km_error = samples_km(num_samples_1, cov_1, mean_1_1, mean_2_1)
#     print("em")
#     print(em_error)
#     print("km")
#     print(km_error)
    em_error_list.append(em_error)
    km_error_list.append(km_error)
    i = i + 1

print("EM error rates for various data realizations")
# for i in em_error_list:
#     print(i)
print(em_error_list)
print("KM error rates for various data realizations")
# for i in km_error_list:
#     print(i)
print(km_error_list)

print("Percentage Difference of KM minus EM error rates")
percentage_list = []
i = 0
while i < 5:
    percentage_list.append((km_error_list[i]-em_error_list[i])*100)
    i +=1
print(percentage_list)
```

```
EM error rates for various data realizations
[0.07999999999999996, 0.026666666666666616, 0.5, 0.018000000000000016, 0.0
6833333333333336]
KM error rates for various data realizations
[0.28500000000000003, 0.09999999999999998, 0.4625, 0.126, 0.30666666666666
664]
Percentage Difference of KM minus EM error rates
[20.500000000000007, 7.333333333333336, -3.749999999999998, 10.79999999999
9999, 23.83333333333333]
```

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 [82]:

```
from qlearning import qlearn
from maze import MazeEnv, ProbabilisticMazeEnv
from plotting_utils import plot_steps_vs_iters, plot_several_steps_vs_iters, plot_polic
y_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.

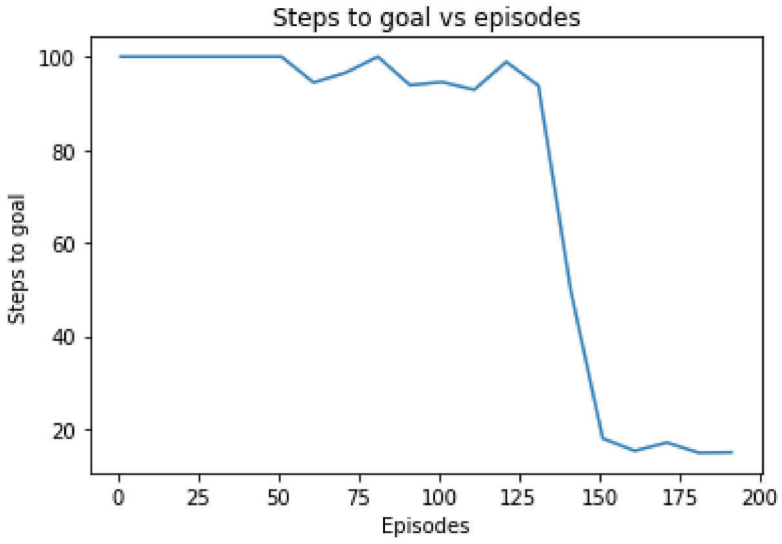
In [83]:

```
# 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_so
ftmax_policy)
```

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

In [84]:

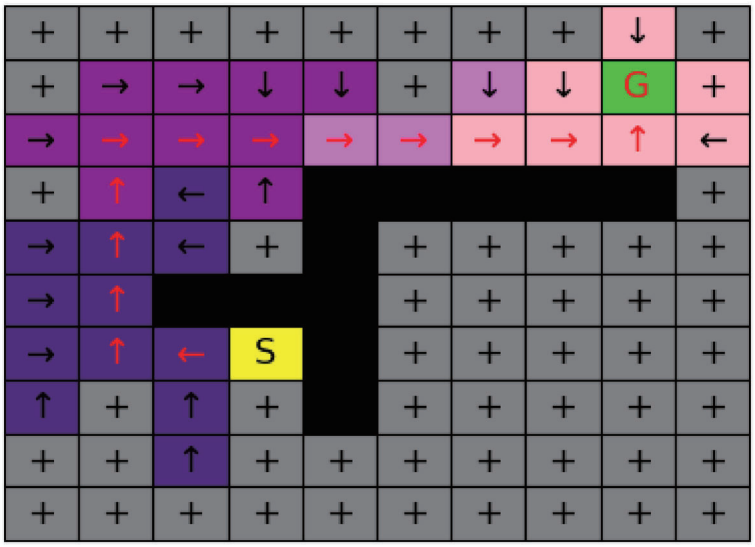
```
# TODO: Plot the steps vs iterations
plot_steps_vs_iters(steps_vs_iters)
```



Visualize the learned greedy policy from the Q values:

In [85]:

```
# 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.

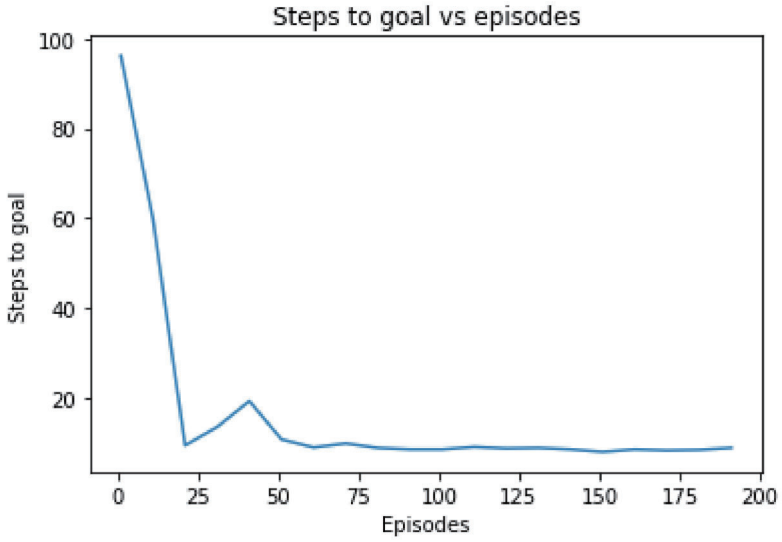
In [88]:

```
# TODO: Fill this in
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_so
ftmax_policy)
```

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

In [89]:

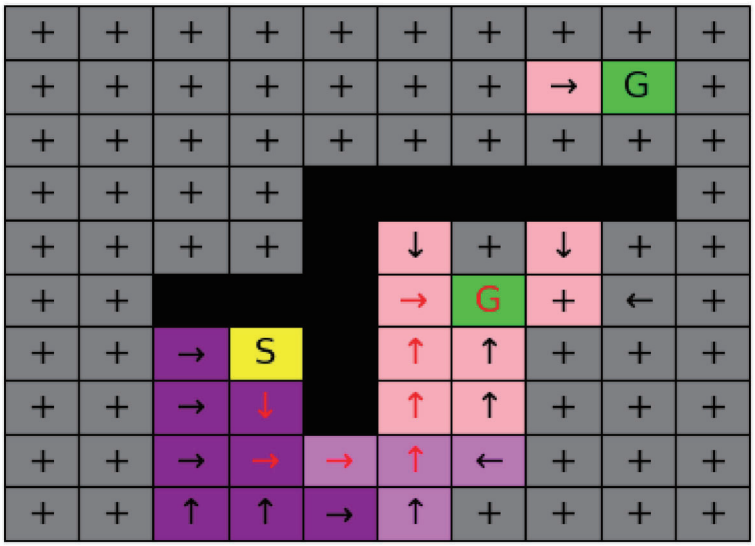
```
# TODO: Plot the steps vs iterations
plot_steps_vs_iters(steps_vs_iters)
```



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

In [90]:

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




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

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

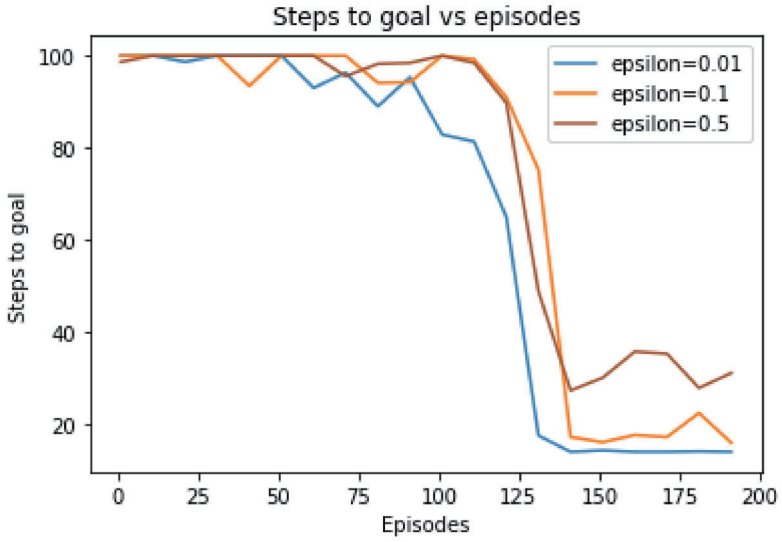
(a) Try different ϵ values in ϵ-greedy exploration: We asked you to use a rate of ϵ=10%, but try also 50% and 1%. Graph the results (for 3 epsilon values) and discuss the costs and benefits of higher and lower exploration rates.

In [116]:

```
# TODO: Fill this in (same as before)
# num_iters = ...
# alpha = ...
# gamma = ...
# max_steps = ...
# use_softmax_policy = ...

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 [117]:

```
# 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, block_size=10)
```



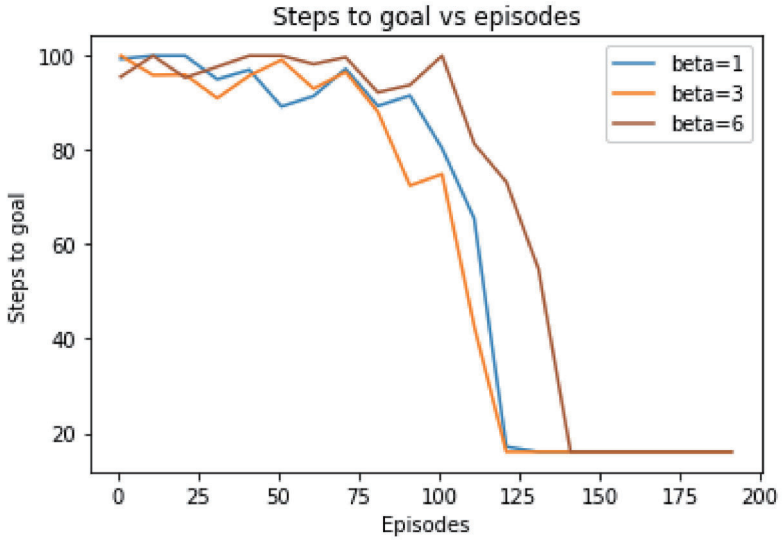
(b) Try exploring with policy derived from **softmax of Q-values** described in the Q learning lecture. Use the values of $\beta \in \{1, 3, 6\}$ for your experiment, keeping β fixed throughout the training.

In [108]:

```
# TODO: Fill this in for Static Beta with softmax of Q-values
# num_iters = ...
# alpha = ...
# gamma = ...
# epsilon = ...
# max_steps = ...
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 = [0.05, 0.1, 0.25, 0.5] # (float) choose k such that we have a constan
t 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, us
e_softmax_policy, init_beta=beta, k_exp_sched=0)
    steps_vs_iters_list.append(steps_vs_iters)
```

In [109]:

```
label_list = ["beta={}".format(beta) for beta in beta_list]
# TODO:
plot_several_steps_vs_iters(steps_vs_iters_list, label_list, block_size=10)
```



(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:

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

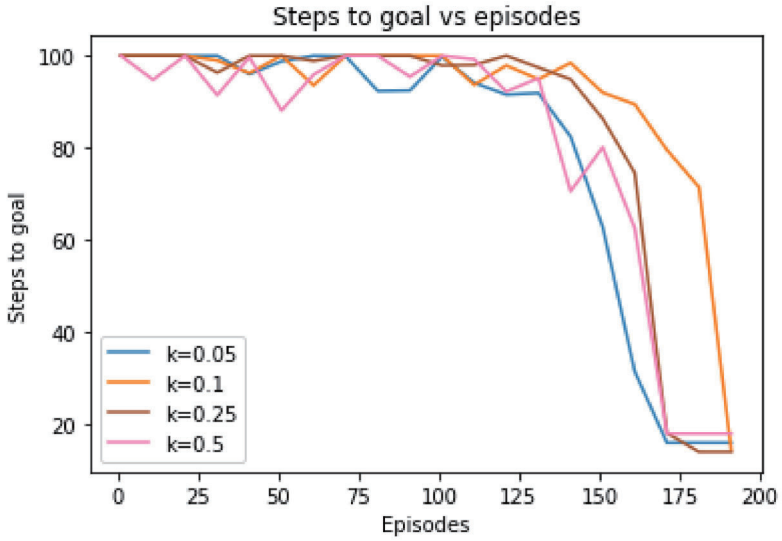
That is, the β value is fixed for a particular episode. Run the training again for different values of $k \in \{0.05, 0.1, 0.25, 0.5\}$, keeping $\beta_0 = 1.0$. Compare the results obtained with this approach to those obtained with a static β value.

In [127]:

```
# TODO: Fill this in for Dynamic Beta
# num_iters = ...
# alpha = ...
# gamma = ...
# epsilon = ...
# max_steps = ...

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, us
e_softmax_policy, init_beta=1, k_exp_sched=k_exp_schedule)
    steps_vs_iters_list.append(steps_vs_iters)
```

In [128]:

```
# 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, block_size=10)
```



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 [95]:

```
# 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?

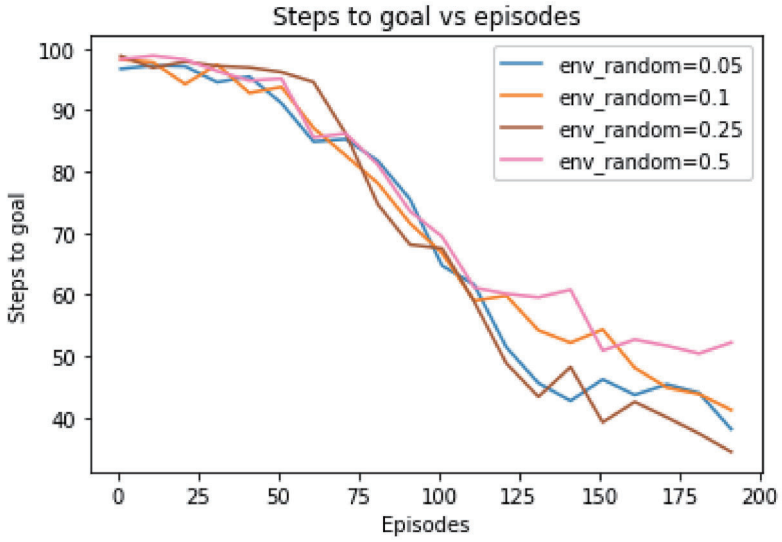
Use the same parameters as in first part, except change the alpha (α) value to be **less than 1**, e.g. 0.5.

In [96]:

```
# TODO: Use the same parameters as in the first part, except change alpha
# num_iters = ...
# alpha = ...
# gamma = ...
# epsilon = ...
# max_steps = ...
# use_softmax_policy = ...

num_iters = 200
alpha = 0.5
gamma = 0.9
epsilon = 0.1
max_steps = 100
use_softmax_policy = True
import numpy as np

# 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()

    # Note: We will repeat for several runs of the algorithm to make the result less no
isy
    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, init_beta=1, k_exp_sched=0.1)
        avg_steps_vs_iters += steps_vs_iters
    avg_steps_vs_iters /= 10
    steps_vs_iters_list.append(avg_steps_vs_iters)
```

In [97]:

```
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, block_size=10)
```



2.3 Write Up

- For section 2.2.1a, we can see that using the epsilon greedy policy with default hyperparamters, the steps to goal drops drastically to around 20 at about 150 episodes/iterations, and from that it appears that the agent more or less converges. The sudden drop in terms of step to goals might be due to a new path explored (which could be due to the randomness given by epsilon) and that path turns out to be the optimal shortest path.
- This is also the same case for 2.2.1b where steps to goal also show a dramatic drop at around 20 epsiodes and appear to converge for the rest of the iterations for the case of 2 goals. This could again be explained by the agent discovering a shortest path to the closest goal (from the 2 goals), then the agent's strategy would be to get to the closest of the 2 goals consistently and ignoring the further goals. This might be the reason why the 2-goals setting had a faster convergence rate than the 1 goal settings. The "bumps/spikes" at around 30 episodes might be due to the epsilon choosing to explore a new path that is around the "further" goal, which is also resulted in a pink arrow head from the policy plot.
- For 2.2.2a, we can see that epsilon 0.01 has the fastest convergence time and also most consistent as well at around 130 episodes. It is followed by epsilon of 0.1, which also seems to converge but had a consistently slightly higher steps to goal compared to 0.01 epsilon. The highest number of steps taken to goal is for epsilon 0.5 and it is also the one with most fluctations and spikes. This is due to the higher probability given by the epsilon value where the agent attempts to try out random actions 50% of the time, instead of choosing ones with lowest q-values (especially for explored paths already).
- For 2.2.2b, we used the softmax policy with 3 different values of beta, but fixing the k_exp_schedule to 0 (since only when k=0, beta will be equal to beta_0). As for the results, we can see that for beta values 1 and 3, the graph is more or less the same converging at around 120 episodes. For beta=6, the graph takes longer episodes (around 140) to converge, but it also converges to around 20 steps. Hence, by using the softmax function, a convergence is guaranteed.
- For 2.2.2c, we have increased the value of beta as the number of episodes increases by training with the different values of k. For all values of k, we can see that it takes a lot longer to converge at around 170 episodes for all k except for k=0.1 This is longer than the 120 episodes compared to previous where static beta is fixed. There is also some inconsistency regarding the number of steps to goal (compared to static beta) as there is not much convergence of steps for the different values of k. For k=0.1, there is also a possibility of no convergence as it had reached the maximum number of iterations already. In general, static beta performed better in terms of nubmer of steps to goal after convergence and also in terms of epsiodes taken to convergence.
- For section 3, a stochastic environment is used and we can observe a steadily decreasing slope of all 4 random environment probabilities. This is a "smoother" decrease compared to all previous experiments, however there is also more inconsistency in terms of convergence of the number of steps to goal (as the graphs seem to continue to decrease even when we reach 200 episodes and there is also lots of fluctuations and spikes for each env_random). In general, env_random of 0.05 and 0.25 appear to reach the lowest number of steps to goal at around 30, followed by env_random=0.1. When env_random=0.5, we see that althought the graph seems to decrease and converge the number of steps to goal is as high as 50 and the flucatutions are also more extreme during each epsiodes. Intuitively, this makes sense as our environment now performs a random action 50% of the time and our agent appear to have a hard time learning and choosing the optimal path with the non-determinism of this new environment.

3. Did you complete the course evaluation?

yes! :)

Codes for maze.py and q.learning

In [ ]:

```
```

Your List of Course Evaluations to Complete

Task Owner: Timothy Grace Lee
Project Title: FAS Fall 2019 Undergrad
Category: Arts and Science
Subcategory: 2019 Fall

Subject	Due date	Status
Methods of Data Analysis I STA302H1-F-LEC0101	Friday, December 6, 2019	Completed
Data Structures & Analysis CSC263H1-F-LEC0101	Friday, December 6, 2019	Completed
Intro Machine Learning CSC311H1-F-LEC0301	Friday, December 6, 2019	Completed
Intro to Databases CSC343H1-F-LEC0101	Friday, December 6, 2019	Completed

Mobile Version | Standard Version

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

    def __init__(self, goals=[[2, 8]], p_random=0.05):
        """Deterministic Maze Environment"""
        MazeEnv.__init__(self,start=[6,3], goals=goals) #inheritance
        self.goals = goals #Override
        self.p_random = p_random

    @override
    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)

        p_uniform = np.random.random_sample() #uniform form 0 to 1

        if p_uniform <= self.p_random:
            a = np.random.random_integers(0, self.num_actions-1)

        #extending from MazeEnv
        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
```

```python
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 of
                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 (False)
        init_beta (float): If using stochastic policy, sets the initial beta as the parameter for the softmax
        k_exp_sched (float): If using stochastic policy, sets hyperparameter for exponential schedule
            on beta

    Returns:
        q_hat: A Q-value table shaped [num_states, num_actions] for environment with with num_states
            number of states (e.g. num rows * num columns for grid) and num_actions number of possible
            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
        if k_exp_sched is None: #use default k of 0.1
            k_exp_sched = 0.1
        # TODO: Keep looping while environment isn't done and less than maximum steps
        while (num_steps < max_steps) and not done:
            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)
                beta = beta_exp_schedule(init_beta, num_iters, k_exp_sched) # Call beta_exp_schedule to get the current beta value
                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, and done flag
            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])
                # 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 - q_hat[curr_state, action] )
                # 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
```

```python
def find_max(states_list):
    """
    A helper function that returns the action with highest Q value for
    current observation/state. Return a random action if all identical.
    """
    # print(len(states_list))
    # print(states_list)
    if np.unique(states_list).size == 1:
        # print("all same")
#         print(np.random.random_integers(0, len(states_list)-1))
        return np.random.random_integers(0, len(states_list)-1)
    else:
        # print("max")
        # print(np.argmax(states_list))
        return np.argmax(states_list)


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

    if np.random.rand() < epsilon:
        action = np.random.choice(action_space_size)
    else:
        states_list = q_hat[state]
        action = find_max(states_list)
    return action

```

```python

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
    action_table = stable_softmax(q_hat * beta) #all q-values for action of each state
    action_list_state = action_table[state] #q-value for actions for this specific state
    return find_max(action_list_state)

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

def stable_softmax(x, axis=1):
    """ 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

```