Assignment 4

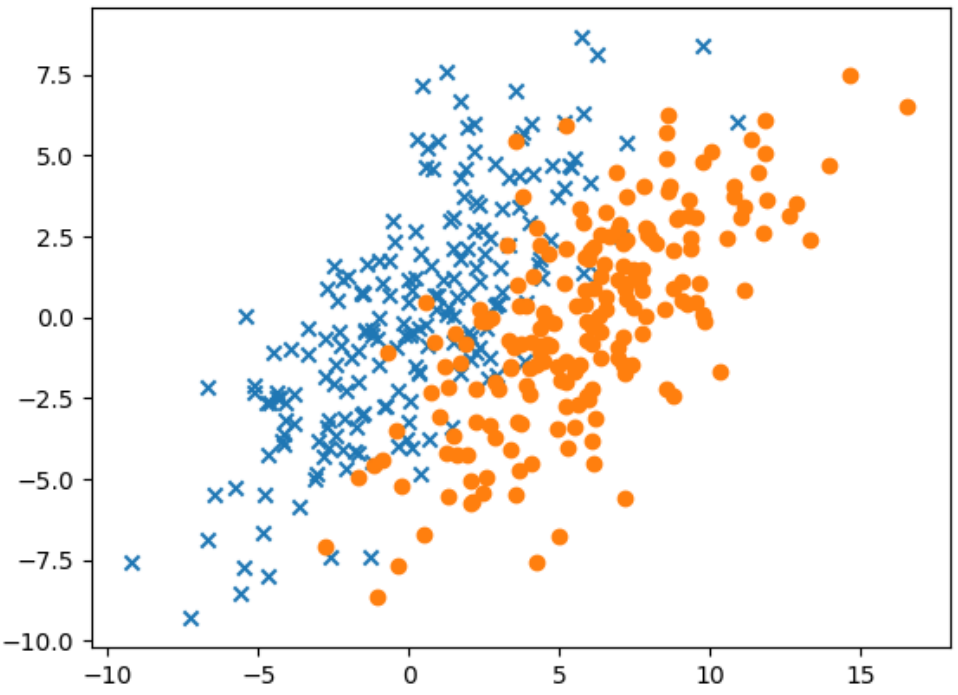
1.

(a)

Code for (a):

```
import ...

# 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]

# TODO: Make a scatterplot for the data points showing the true cluster assignments of each p
plt.scatter(x_class1[:, 0], x_class1[:, 1], marker="x")  # first class, x shape
plt.scatter(x_class2[:, 0], x_class2[:, 1], marker="o")  # second class, circle shape
plt.show()
```

Visualize the generated data points:



(b)

Code for (b):

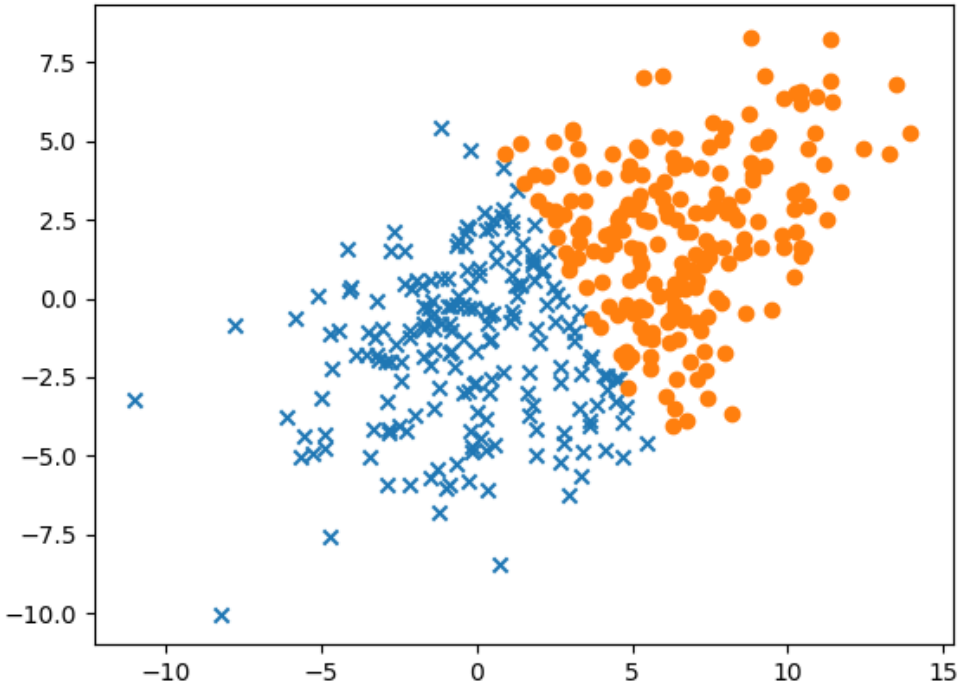
```python
# TODO: K-Means Assignment Step
def km_assignment_step(data, Mu):
    """ Compute K-Means assignment step

    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 - Mu[:, k], axis=1)
    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[np.arange(N), arg_min] = 1  # Assign to 1
    return R_new
```

```python
# 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 = R.shape[1]  # number of clusters
    Mu_new = data.T.dot(R) / sum(R)
    return Mu_new


# 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)
```

```python
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 = []
for it in range(max_iter):
    R = km_assignment_step(data, Mu)
    Mu = km_refitting_step(data, R, Mu)
    costs.append(cost(data, R, Mu))


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

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

class_1_labels = labels[class_1[0]]
match_1 = max(sum(class_1_labels), len(class_1[0]) - sum(class_1_labels))
class_2_labels = labels[class_2[0]]
match_2 = max(sum(class_2_labels), len(class_2[0]) - sum(class_2_labels))
accuracy = (match_1 + match_2) / N
print("Misclassification error is ", 1 - accuracy)

# TODO: Make a scatterplot for the data points showing the K-Means cluster assignments of each point
plt.scatter(data[class_1[0], np.zeros(class_1[0].shape[0], dtype=int)],
            data[class_1[0], np.ones(class_1[0].shape[0], dtype=int)], marker="x")  # first class, x shape
plt.scatter(data[class_2[0], np.zeros(class_2[0].shape[0], dtype=int)],
            data[class_2[0], np.ones(class_2[0].shape[0], dtype=int)], marker="o")  # second class, circle shape
plt.show()
plt.plot(np.arange(max_iter), costs)
plt.show()
```

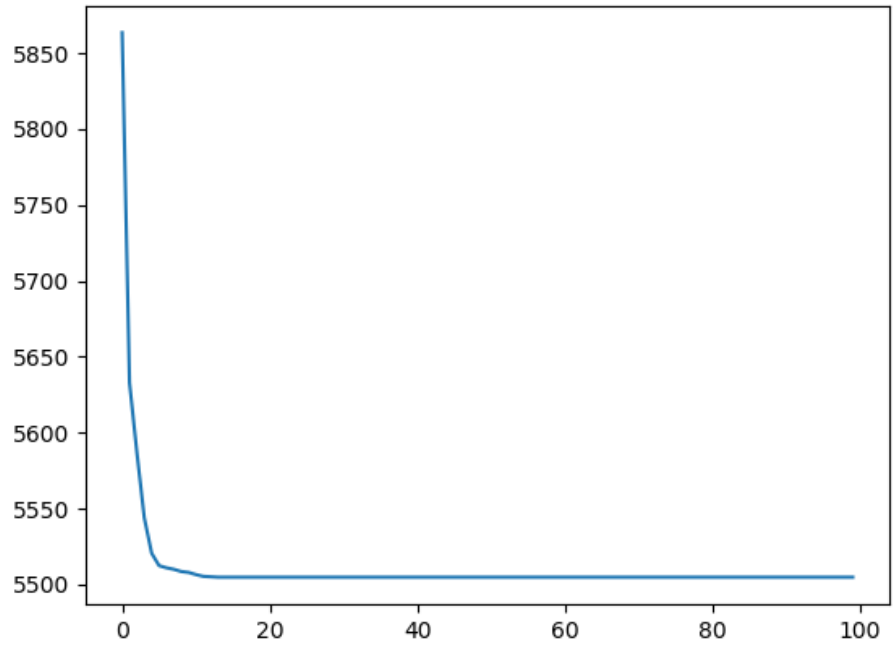
The resulting cluster assignments:



We have the misclassification rate as:

```
Misclassification error is  0.27
```

The cost vs. each iteration:



(c)

Code for (c):

```python
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]  # Number of datapoints and dimension of datapoint
    K = Mu.shape[1]  # number of mixtures
    L, T = 0., 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 likelihood from the k-th Gaussian weighted by the mixing coefficients
        L += np.log(T)
    return L
```

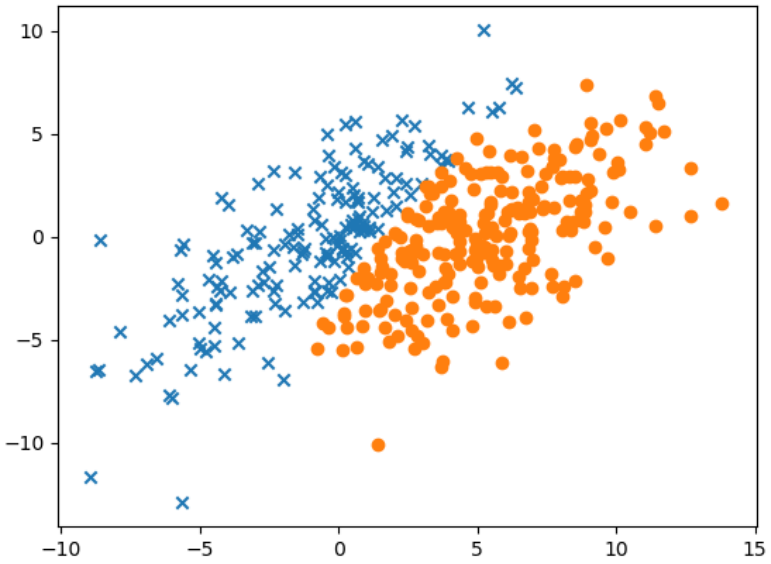
```python
# TODO: Gaussian Mixture Expectation Step
def gm_e_step(data, Mu, Sigma, Pi):
    """ Gaussian Mixture Expectation Step.

    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]  # 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, axis=1)[n]
        # Normalize by sum across second dimension (mixtures)
    return Gamma
```

```python
# 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]  # Number of datapoints and dimension of datapoint
    K = Gamma.shape[1]  # number of mixtures
    Nk = sum(Gamma)  # Sum along first axis
    Mu = data.T.dot(Gamma) / Nk
    Sigma = [0] * K
    for k in range(K):
        gamma_matrix = np.diag(Gamma[:, k])
        Sigma[k] = (data - Mu[:, k]).T.dot(gamma_matrix).dot(data - Mu[:, k]) / Nk[k]
    Pi = Nk / N
    return Mu, Sigma, Pi
```

```python
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

costs = []
log_likelihoods = []
for it in range(max_iter):
    Gamma = gm_e_step(data, Mu, Sigma, Pi)
    Mu, Sigma, Pi = gm_m_step(data, Gamma)
    costs.append(cost(data, Gamma, Mu))
    log_likelihoods.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
# print(Gamma)
```

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

class_1_labels = labels[class_1[0]]
match_1 = max(sum(class_1_labels), len(class_1[0]) - sum(class_1_labels))
class_2_labels = labels[class_2[0]]
match_2 = max(sum(class_2_labels), len(class_2[0]) - sum(class_2_labels))
accuracy = (match_1 + match_2) / N
print("Misclassification error is ", 1 - accuracy)

# TODO: Make a scatterplot for the data points showing the Gaussian Mixture cluster assignments of each point
plt.scatter(data[class_1[0], np.zeros(class_1[0].shape[0], dtype=int)],
            data[class_1[0], np.ones(class_1[0].shape[0], dtype=int)], marker="x")  # first class, x shape
plt.scatter(data[class_2[0], np.zeros(class_2[0].shape[0], dtype=int)],
            data[class_2[0], np.ones(class_2[0].shape[0], dtype=int)], marker="o")  # second class, circle shape
plt.show()

plt.plot(np.arange(max_iter), log_likelihoods)
plt.show()
```

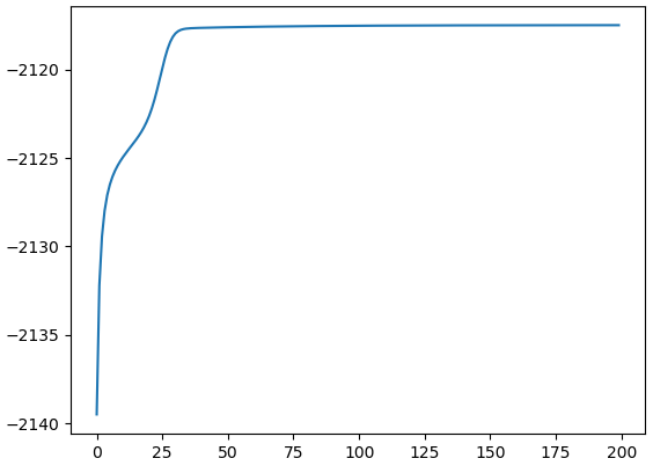
The resulting cluster assignments:



We have the misclassification rate as:

```
Misclassification error is  0.13249999999999995
```

Log-likelihood vs. number of iterations:



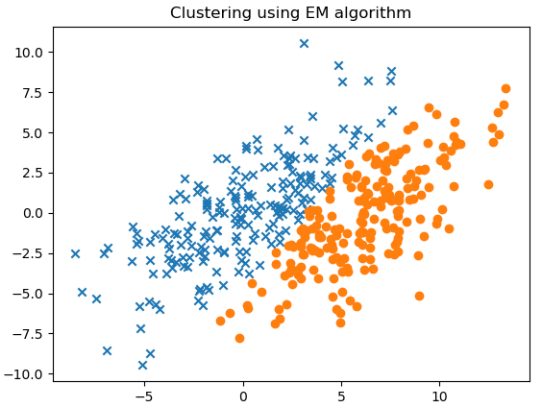
(d)

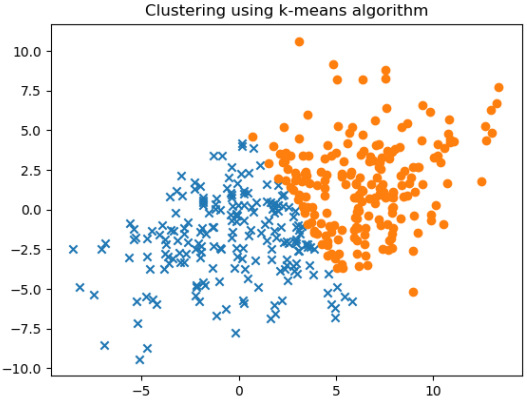
<a> We first compare the misclassification error for the two methods. As EM algorithm compares the probabilities of each cluster, it should get better accuracy. Our test output proved this. Error for EM algorithm is around 10%, which is smaller than k-means (around 25%).

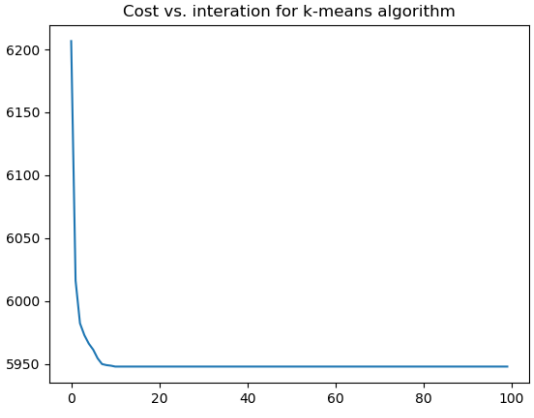
<b> Comparing the cost vs. iteration plot for k-means and log-likelihood vs. iteration for EM algorithm, we see that EM algorithm takes more iteration to converge, since k-means converge at iteration around 10, but EM takes around 30. This may be a bottleneck for EM algorithm. In real practice, EM may take running time.

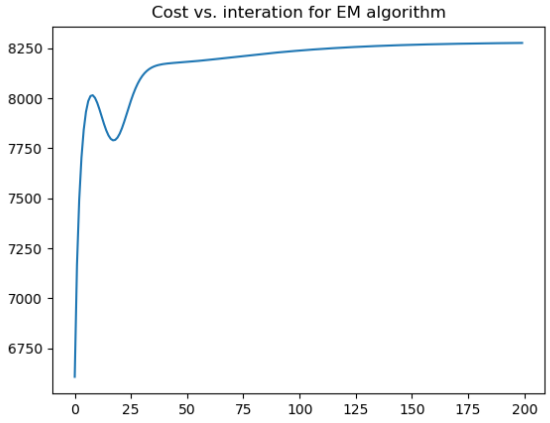
<c> We run the two algorithms on the same generated data for 5 times, we get the following resulting graph:

1st time:



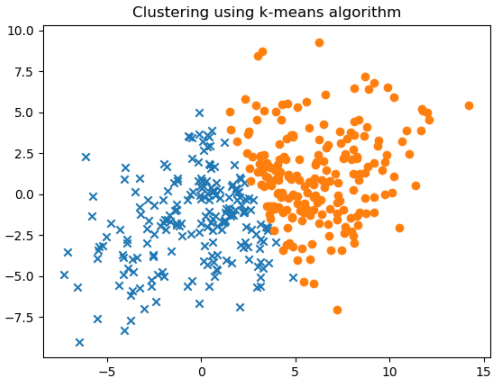


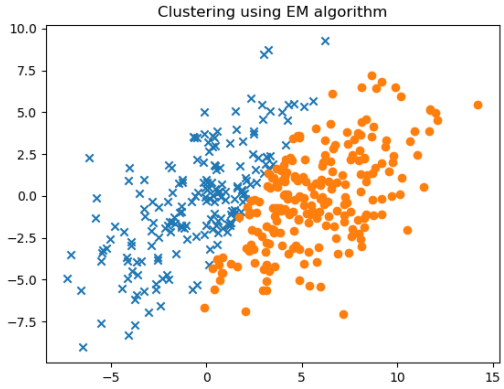


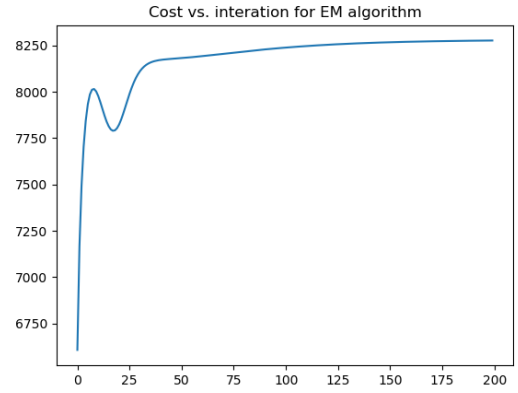


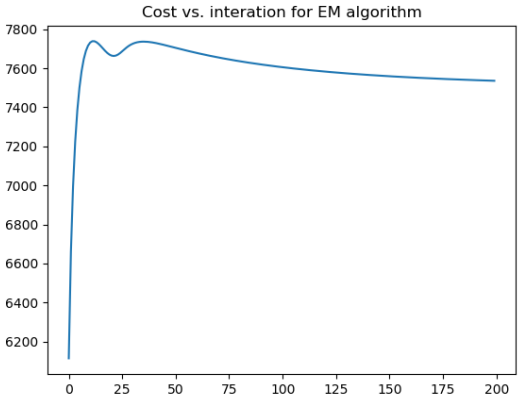

```
Misclassification error for k-means algorithm is  0.28
Misclassification error for EM algorithm is  0.10499999999999998
```

2nd time:



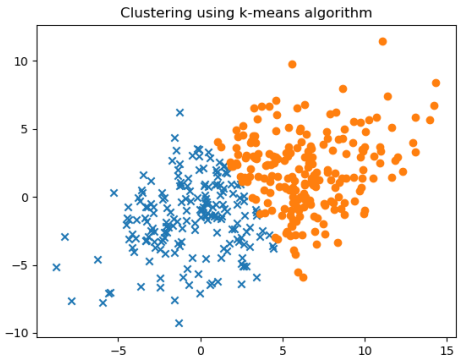


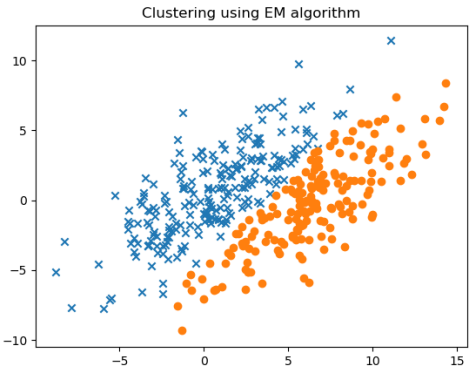


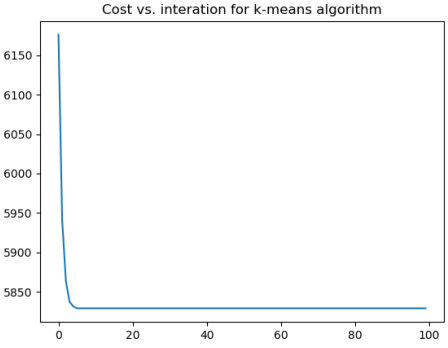


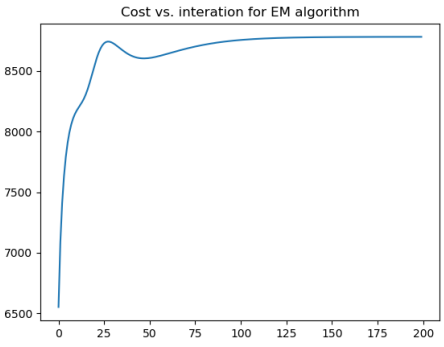

```
Misclassification error for k-means algorithm is  0.245
Misclassification error for EM algorithm is  0.11750000000000005
```

3rd time:



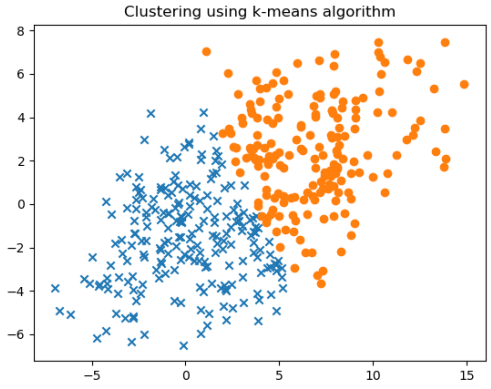


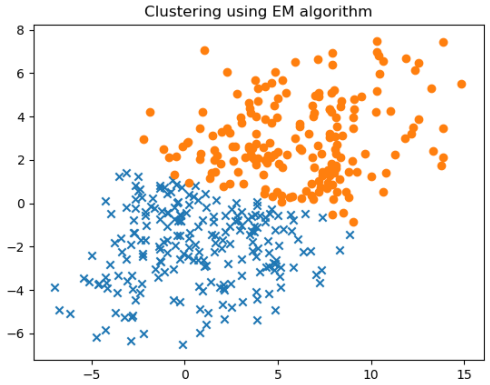


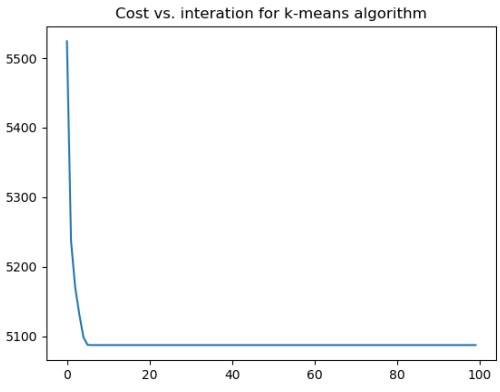


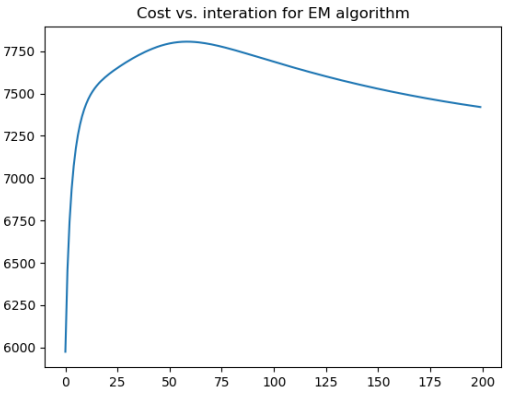
```
Misclassification error for k-means algorithm is  0.25249999999999995
Misclassification error for EM algorithm is  0.08999999999999997
```

4th time:



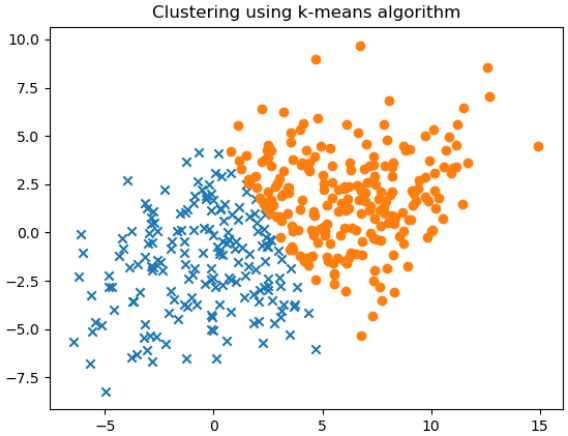


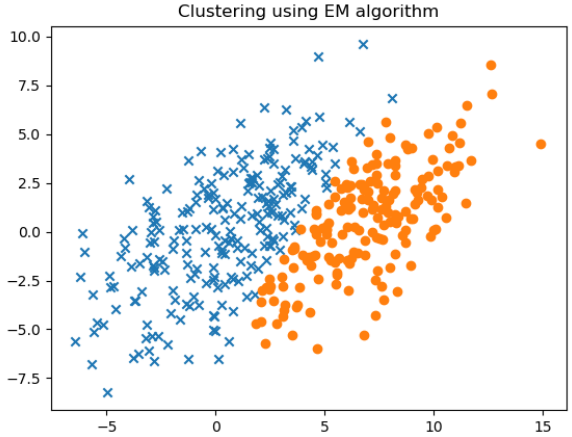


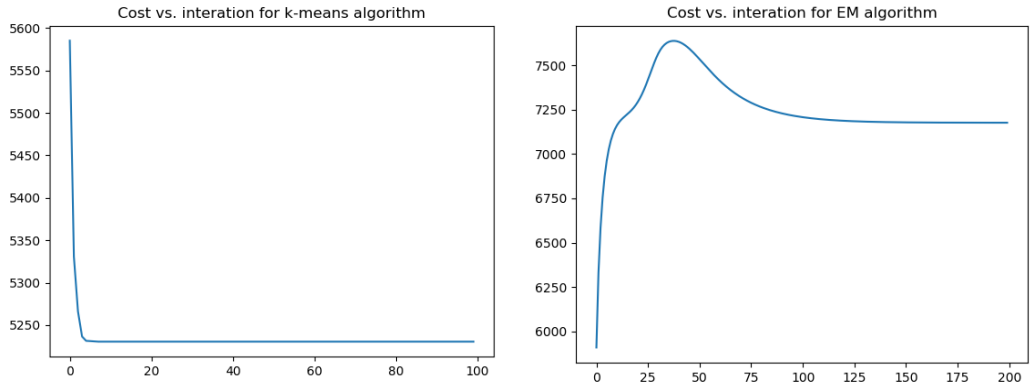



```
Misclassification error for k-means algorithm is  0.30000000000000004
Misclassification error for EM algorithm is  0.41000000000000003
```

5th time:








```
Misclassification error for k-means algorithm is  0.24
Misclassification error for EM algorithm is  0.08999999999999997
```

K-means algorithm's performance is stable as the classification errors are all around 25%. However, in the 4th running, the classification error for the EM algorithm rises up to 41%, which does not follow the trend of the other 4 experiments. Cost functions converge for both algorithms. We conclude that the performance for k-means is stable but a bit low in accuracy, EM algorithm usually has higher accuracy, but depends on data realizations. For some "unfortunate" data, EM perform worse than k-means.

2.

2.1

Code for 2.1:

```python
import ...


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
```

```
    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

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

```
        while done is False 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)
                beta = beta_exp_schedule(init_beta, i, 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 * np.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)]
```

```
            # TODO: Update Q_value
            if next_state != curr_state:
                new_value = reward + gamma * np.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] = 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
    q_state = q_hat[state]
    q_max_move = np.argmax(q_state)
    probabilities = [epsilon, 1 - epsilon]
    random_move = np.random.choice(action_space_size)
    action = np.random.choice([random_move, q_max_move], p=probabilities)

    if np.all(q_state == 0):
        action = random_move

    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
        obs: 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
    q_state = q_hat[state]
    max_q = np.max(q_state)
    z = np.exp(beta * (q_state - max_q))
    probabilities = z / np.sum(z)
    actions = np.arange(q_hat.shape[1])
    action = np.random.choice(actions, p=probabilities)
    return action
```

We resolve the issue of "ties" in Q_hat matrix by adding code in epsilon-greedy policy. For softmax policy, the algorithm will randomly assign a move when we have ties in several 0's in Q_hat.

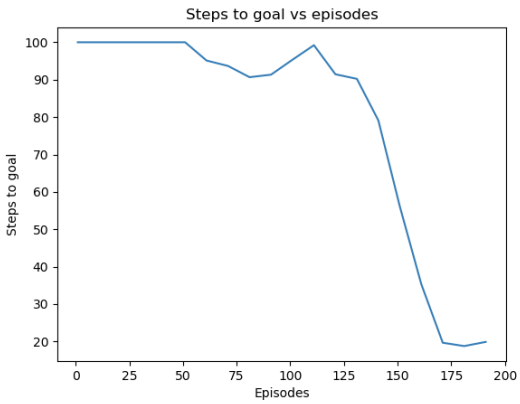
2.2.1

(a)

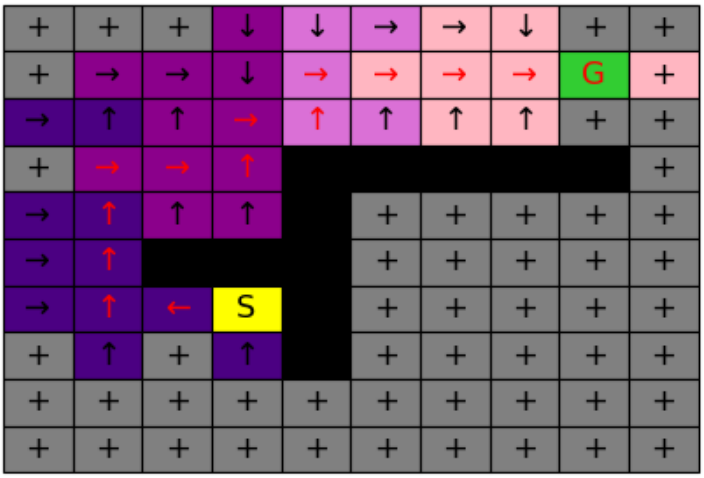
Code for 2.2.1(a):

```python
import ...

# 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_p

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


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

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



Steps to goal converges according to our implementation.
Visualize the learned greedy policy from the Q values:

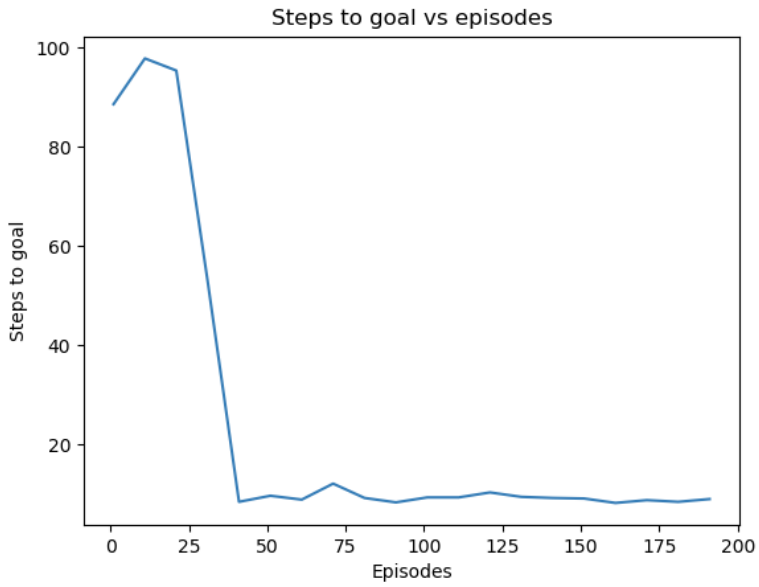


Following the red arrows, we see that the agent found the optimal path to goal.

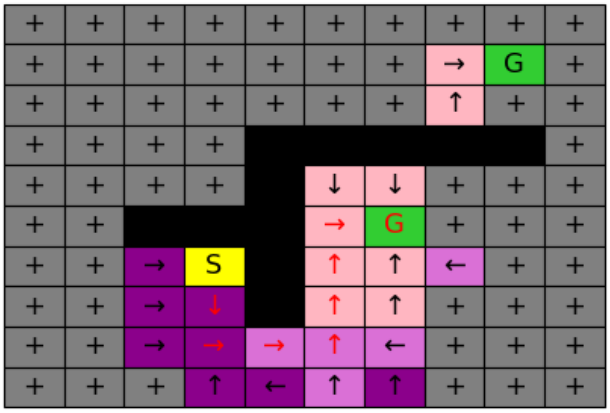
2.2.1(b)
Code for 2.2.1(b):

```
import ...


# 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_p

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

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

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



Visualize the learned greedy policy from the Q values:



Following the red arrows, the agent gets to the nearest goal (5,6). Since the total distance is shorter this time, steps to goal converges faster than 2.2.1(a).

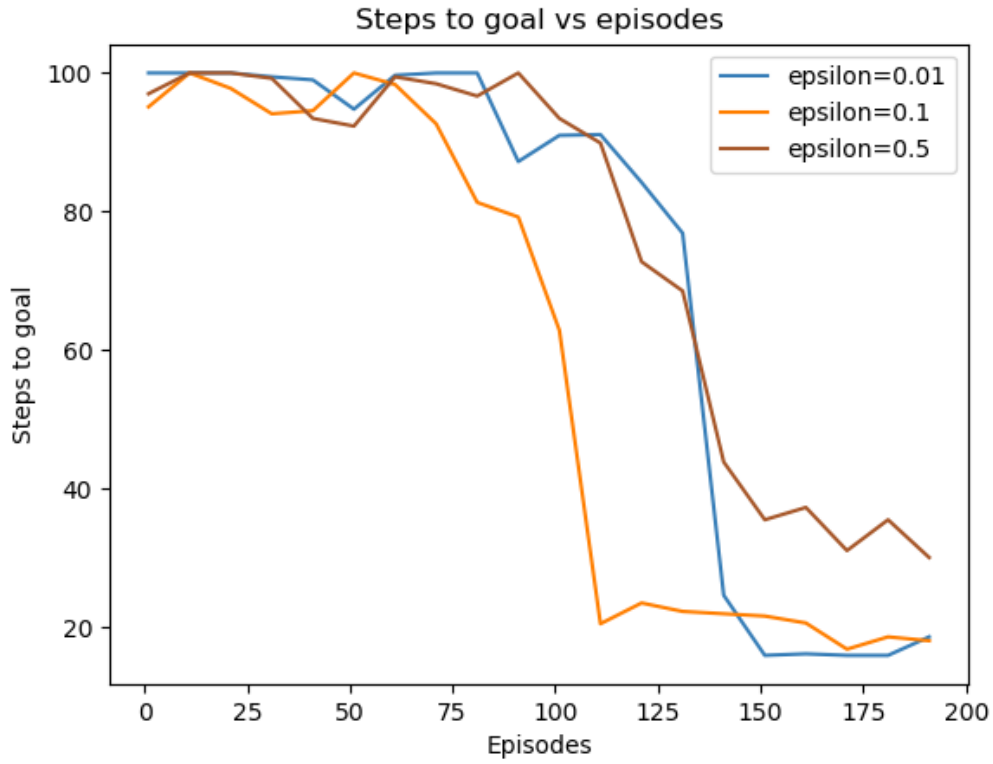
2.2.2

2.2.2(a)

Code for 2.2.2(a):

```python
import ...


# 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 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_softm
    steps_vs_iters_list.append(steps_vs_iters)
# 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)
```

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

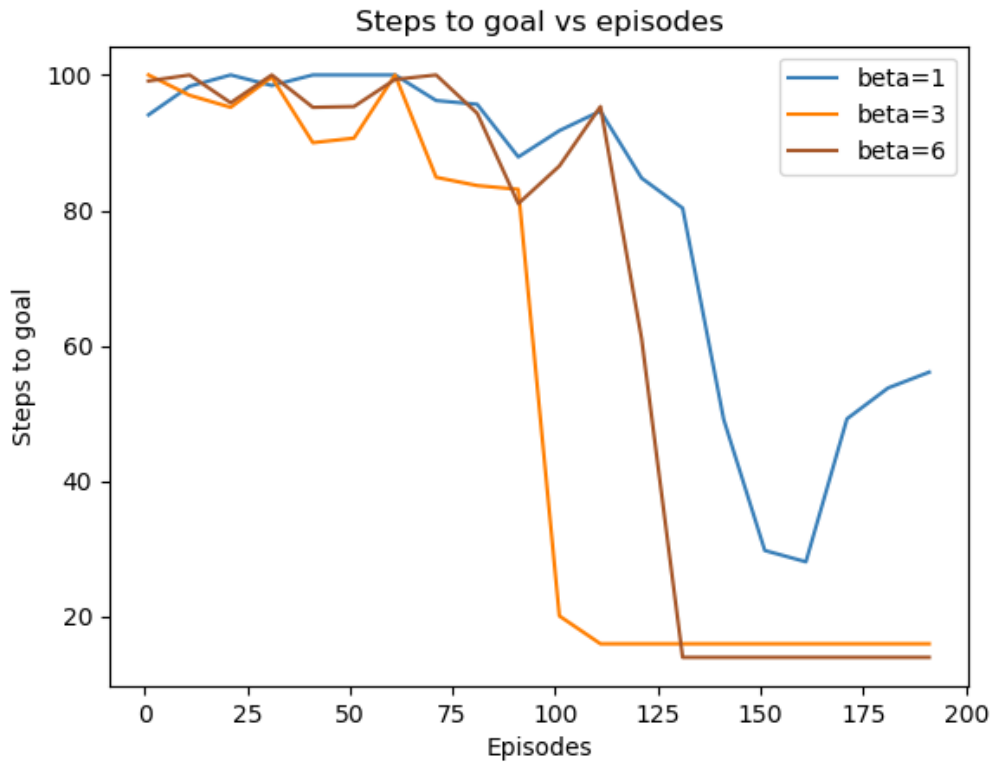


It seems that epsilon = 0.1 works best, since it converges the fastest. This indicates that too low and too high exploration rates will influence performance negatively.

2.2.2(b)
Code for 2.2.2(b):

```python
import ...
# 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)

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

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



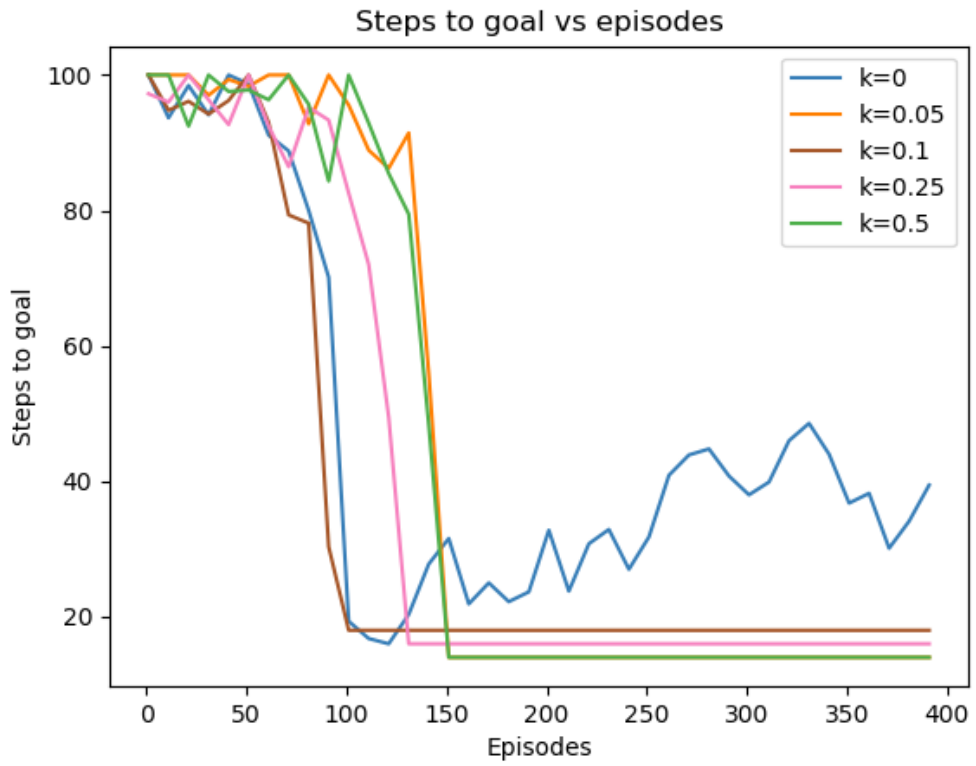
Beta = 3 has the best performance according to our experiments. Again, like the previous problem, too small and too large beta also negatively affect performance.

2.2.2(c)

Code for 2.2.2(c):

```
import ...
# TODO: Fill this in for Dynamic Beta
num_iters = 600
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, 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)

# 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)
```

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



Clearly from the plot, it is better to have $k \neq 0$, since in the case $k = 0$, steps to goal does not converge even after 400 iterations. It seems that as long as $k \neq 0$, the algorithm works well. If we want to converge as quick as possible, we can choose $k = 0.1$. This

also indicates that the convergence rate depends on the moderately chose of hyperparameters, since small k or large k won't converge as fast as k = 0.1, the medium value.

2.2.3

Code for 2.2.3(a):

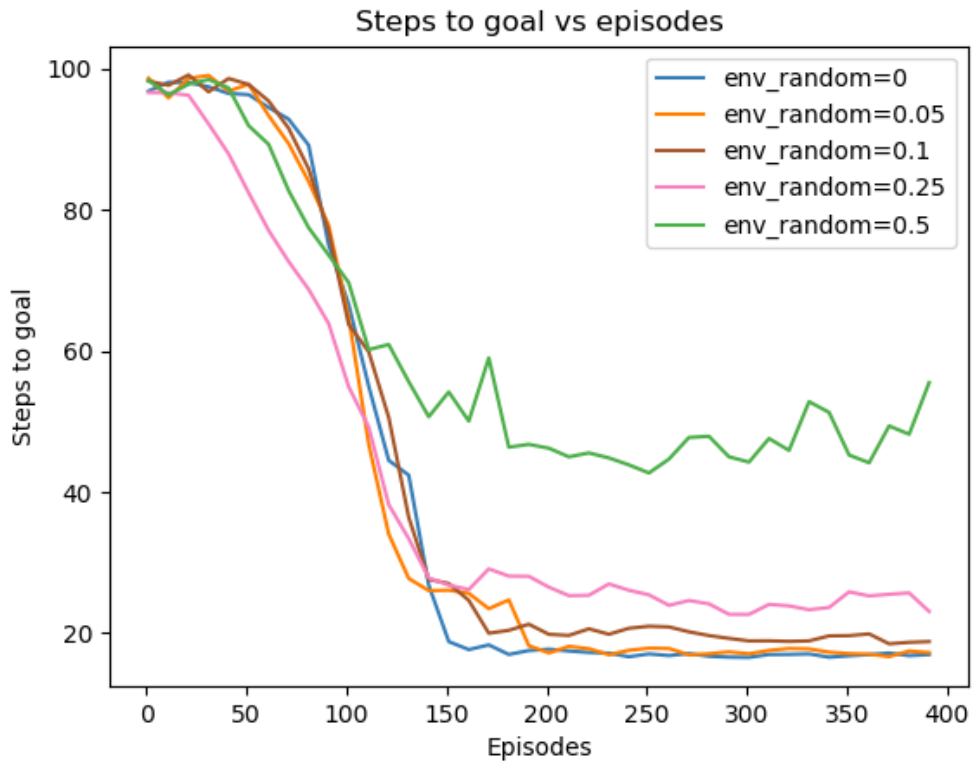
```python
class ProbabilisticMazeEnv(MazeEnv):
    """ (Q2.3) Hints: you can refer the implementation in MazeEnv
    """

    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):
        done, reward = False, 0.0
        next_obs = copy.copy(self.obs)

        random_action = np.random.choice(self.num_actions)
        a = np.random.choice([a, random_action], p=[1-self.p_random, self.p_random])

        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
```

We override the initializer and the step() method in the subclass ProbabilisticMazeEnv.

Code for 2.2.3(b):

```
import ...
# TODO: Use the same parameters as in the first part, except change alpha
num_iters = 400
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, 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_s
        avg_steps_vs_iters += steps_vs_iters
    avg_steps_vs_iters /= 10
    steps_vs_iters_list.append(avg_steps_vs_iters)
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)
```

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



Take env_random = 0.05 or 0.1 if we want the fastest convergence. It should not be too large, since env_random = 0. 5 does not converge after 400 iterations.

2.3

As a summary, we realize that if we want the fastest convergence rate, we have to set the parameters carefully, not too large and not too small. In general, it’s better for them to be non-zero to make the algorithm converge.