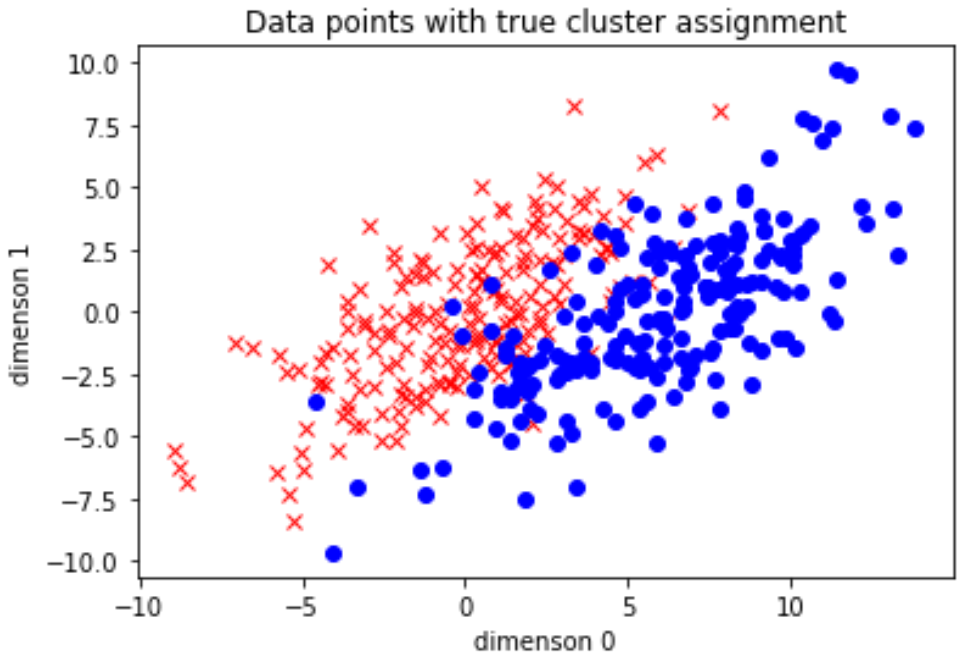
HOMEWORK 4
CSC311 Fall 2019

Nicole Xin Yue Wang
Utorid: wangnic8, #1004235339

1. Unsupervised Learning

(a) Scatter plot of the data points showing the true cluster assignment of each points. First class is red 'x', second class is blue 'o'

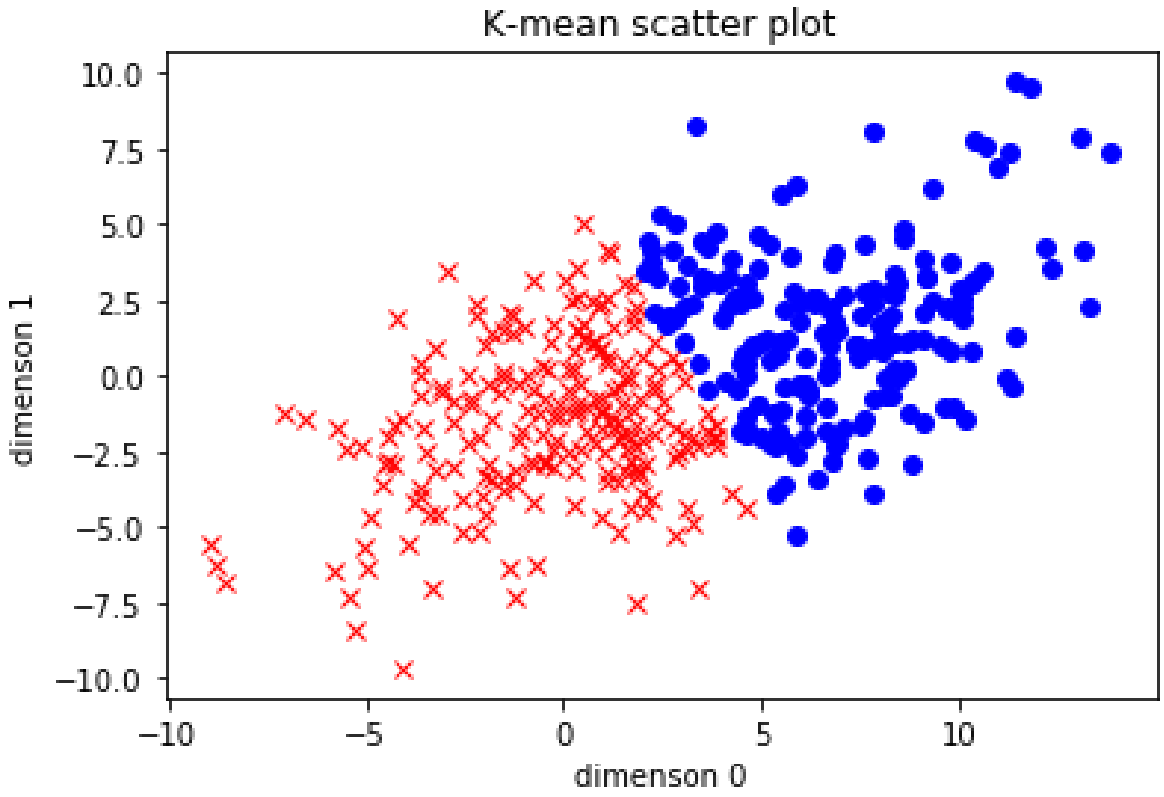


(b) Written two functions for k-mean:

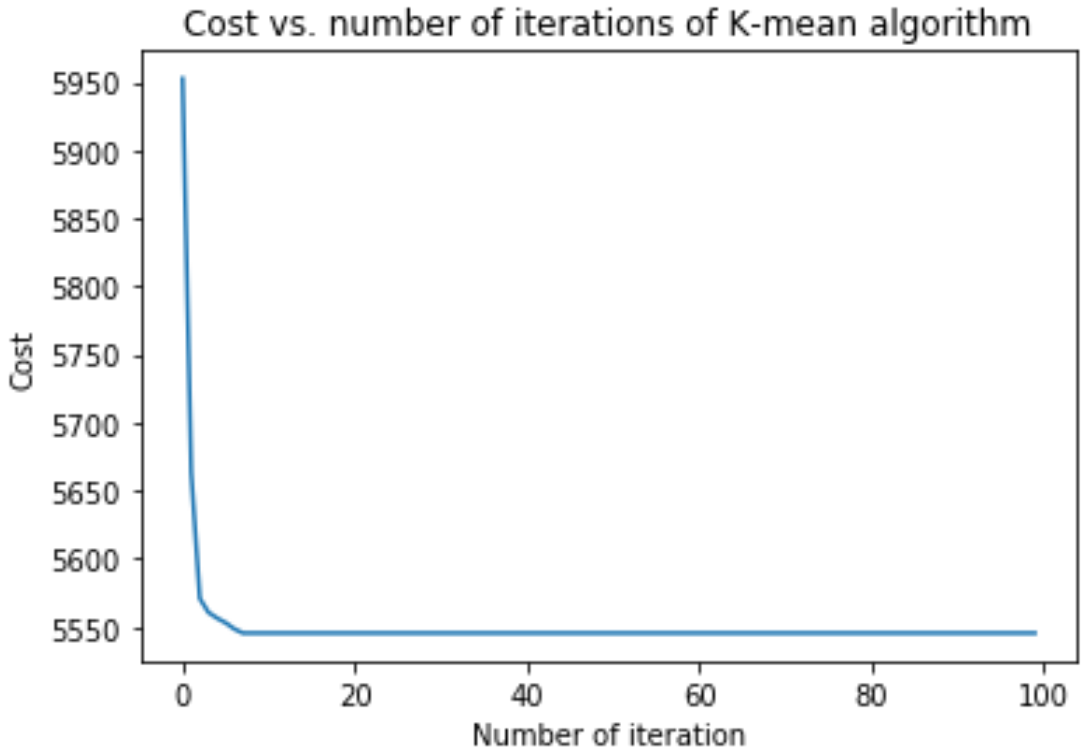
```
def km_assignment_step(data, Mu):
    """ Compute K-Means assignment step

    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
    """
    N, D = data.shape
    K = np.shape(Mu)[1]
    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
```

```
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 = np.shape(Mu)[1]  # number of clusters
    Mu_new = np.zeros((D,K))
    for k in range(K):
        Mu_new[:, k] = np.sum((data.T * R[:, k]).T , axis=0) / np.sum(R[:, k])
    return Mu_new
```

Scatter plot for output:



Cost vs. number of iterations:

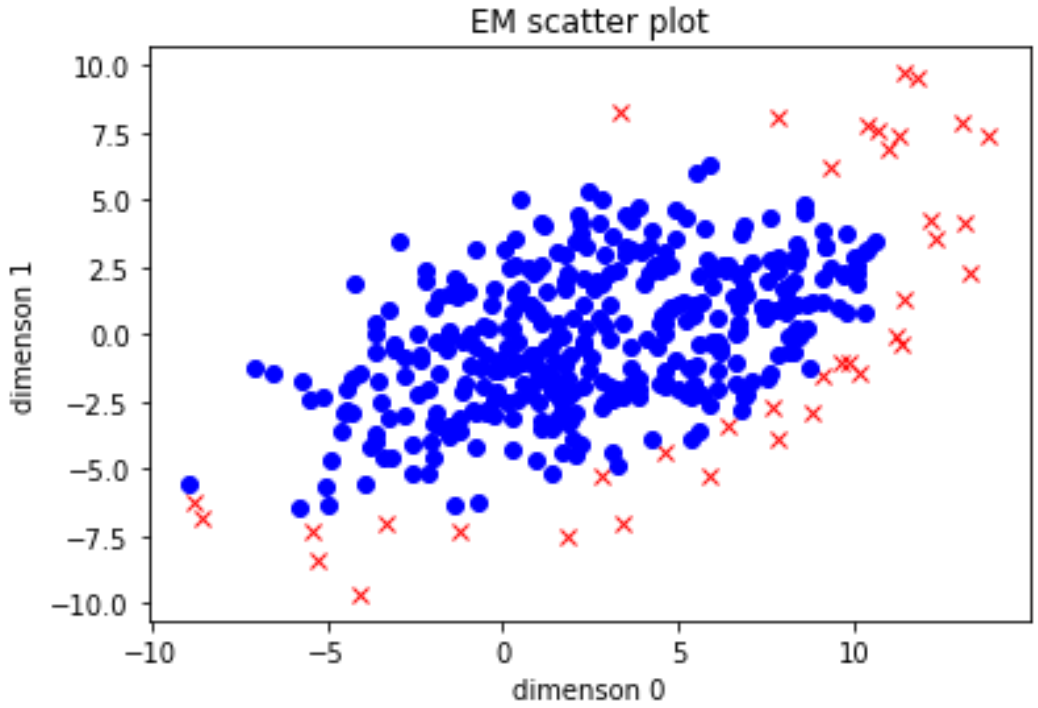


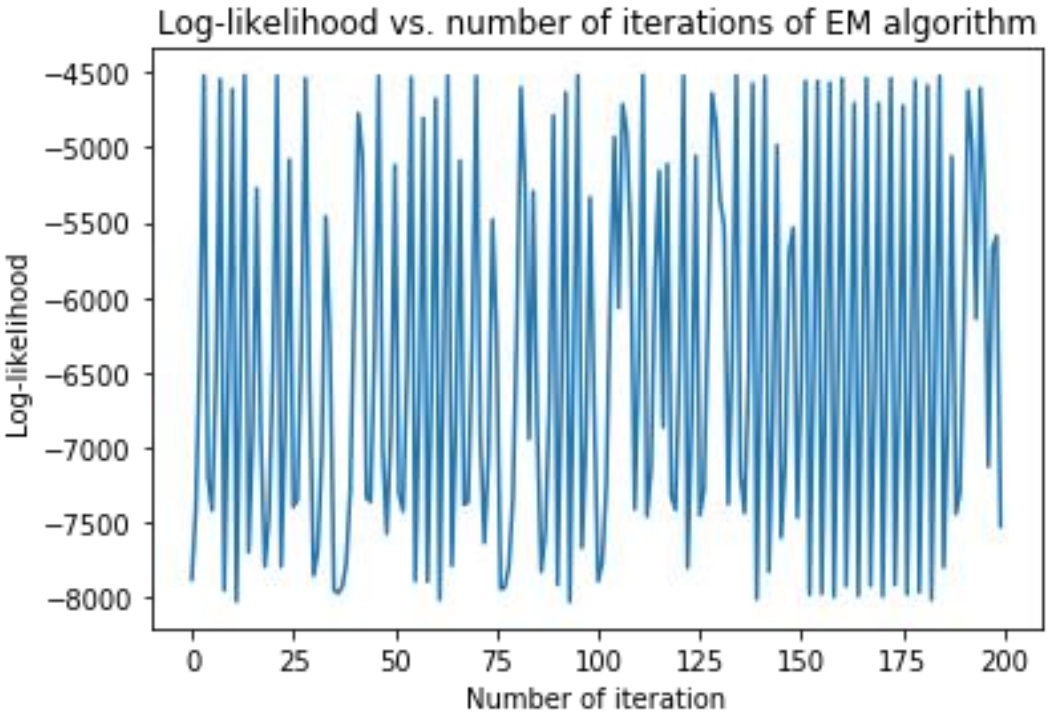
Misclassification error is: 0.24

(c) Written functions for EM:

```
def log_likelihood(data, Mu, Sigma, Pi):
    N, D = data.shape  # Number of datapoints and dimension of datapoint
    K = np.shape(Mu)[1] # number of mixtures
    L, T = 0., 0.
    for n in range(N):
        T = 0
        for k in range(K):
            # Compute the likelihood from the k-th Gaussian weighted by the mixing coefficients
            T += Pi[k] * multivariate_normal(mean=Mu[:,k], cov=Sigma[k]).pdf(data[n])
        L += np.log(T)
    return L

def gm_e_step(data, Mu, Sigma, Pi):
    N, D = data.shape # Number of datapoints and dimension of datapoint
    K = np.shape(Mu)[1] # number of mixtures
    Gamma = np.zeros((N,K)) # zeros of shape (N,K), matrix of responsibilities

    for k in range(K):
        exp = 1 / np.sqrt((2 * np.pi) ** D * np.linalg.det(Sigma[k])) * np.diag(np.exp(-0.5 * np.dot(
            np.dot(data - Mu[:,k], np.linalg.inv(Sigma[k])), (data - Mu[:,k]).T)))
        Gamma[:,k] = Pi[k] * exp
    Gamma = (Gamma.T / np.sum(Gamma, axis=1)).T
    return Gamma

def gm_m_step(data, Gamma):
        N, D = data.shape # Number of datapoints and dimension of datapoint
        K = Gamma.shape[1]  # number of mixtures
        Nk = np.sum(Gamma, axis=1) # Sum along first axis
        Mu = np.zeros((D, K))
        Sigma = [np.eye(2), np.eye(2)]

        for k in range(K):
            Mu[:,k] = np.sum(Gamma[:,k] * data.T, axis=1).T / Nk[k]
            x_miu = data - Mu[:,k]
            Sigma[k] = np.dot(np.multiply(x_miu.T, Gamma[:,k]), x_miu) / Nk[k]

        Pi = Nk / N
        return Mu, Sigma, Pi
```

Scatter plot





Misclassification error is: 1.0

2. Reinforcement Learning

2.1 Implementation

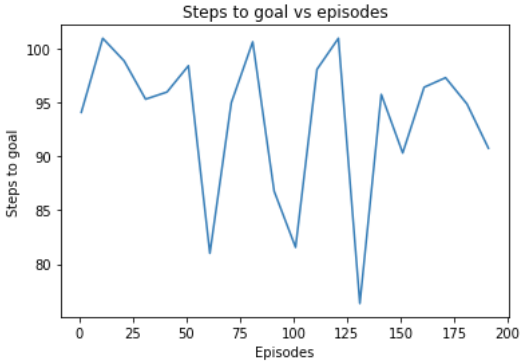
```
def qlearn(env, num_iters, alpha, gamma, epsilon, max_steps, use_softmax_policy, init_beta=None,
k_exp_sched=None):
    action_space_size = env.num_actions
    state_space_size = env.num_states
    q_hat = np.zeros(shape=(state_space_size, action_space_size))
    steps_vs_iters = np.zeros(num_iters)

    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 == False and (num_steps <= max_steps)):
            num_steps += 1

            # Choose an action using policy derived from either softmax Q-value
            # or epsilon greedy
            action = 0
            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, 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])
                # 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] = updated_q_sa(q_hat[curr_state, action],
                                                new_value, alpha)

                # 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
```

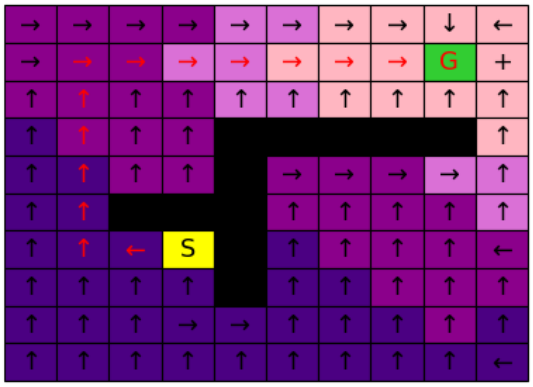
```
def updated_q_sa(q_sa, new_value, alpha):
    """ Update the action-value function at state-action (S_t, A_t)
    Args:
        q_sa: Q(S_t, A_t)
        new_value: reward + gamma * max_a'(S_t+1, a')
        alpha: learning rate
    Returns:
        new_q_sa: new updated Q(S_t, A_t)
    """
    return q_sa + alpha * (new_value - q_sa)


def epsilon_greedy(q_hat, epsilon, state, action_space_size):
    # TODO: Implement your code here
    # Hint: Sample from a uniform distribution and check if the sample is below
    # a certain threshold
    action = 0
    if np.random.random() <= epsilon:
        action = np.argmax(q_hat[state])
    else:
        action = np.random.choice(np.arange(action_space_size))

    return action

def softmax_policy(q_hat, beta, state):
    # TODO: Implement your code here
    # Hint: use the stable_softmax function defined below
    softmax = stable_softmax(beta * q_hat[state], axis=0)
    action = np.random.choice(np.arange(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):
    max_x = np.max(x, axis, keepdims=True)
    z = np.exp(x - max_x)
    output = z / np.sum(z, axis, keepdims=True)

    return output
```

2.2 Experiments

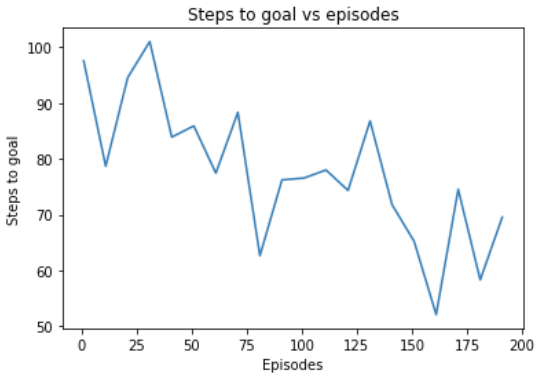
1. Basic Q learning experiments

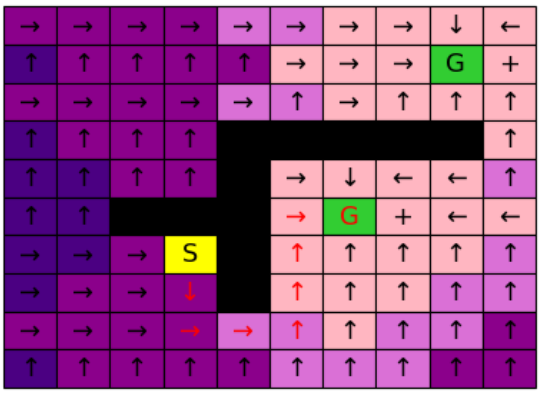
   (a) Running algorithm on given environment





   (b) Run algorithm with 2 goal locations

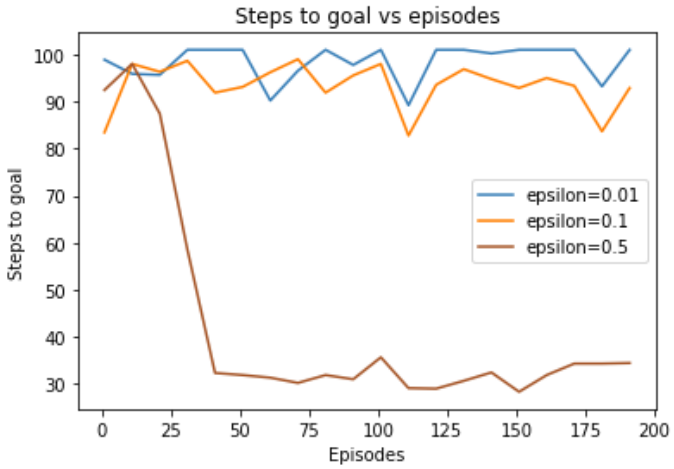




We can see that when the environment has to different goals, the steps to finding a goal decreases as more episodes are gained. This could be due to the agent has found the goal that is easier to approach. In these two experiments, the original given environment has a goal that is far and has lots of barriers from initial space, while the second environment also has that goal but an easier goal to achieve. By learning in the environment, the agent finds that easy goal and by learning more in the environment, it finds a way to achieve the goal in an optimal way.

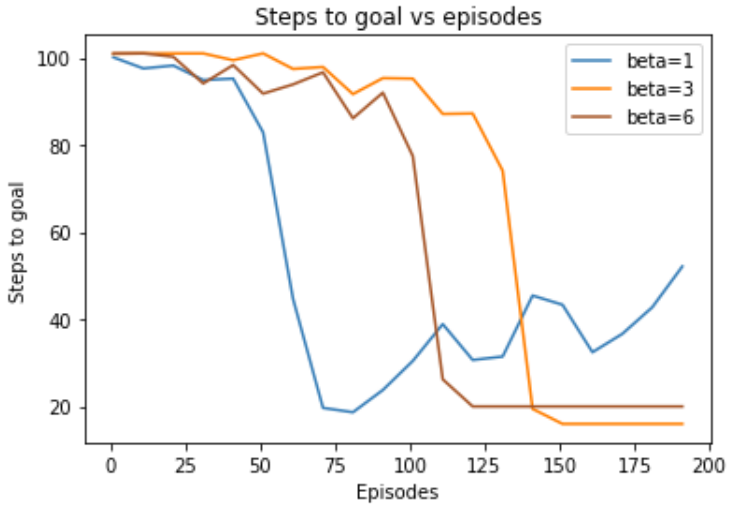
2. Experiment with the exploration strategy.

   (a) Different exploration rates:



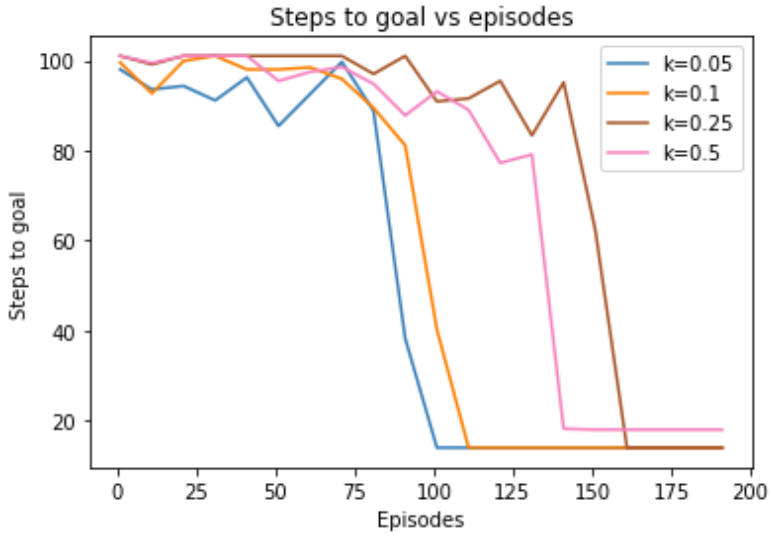
We can see that from the beginning, when not much episodes went on, epsilon rate being 0.1 does the best job, then later on, epsilon being 0.5 does the greatest job when more episodes iterated. This makes sense since at the beginning, more exploration is needed, so a smaller epsilon would make more explorations. But when some explorations are done, the agent should start exploring places closer to a goal. Then a bigger epsilon would perform better since then the agent would have bigger chance of moving toward the goal.

(b) Exploring with fixed beta soft max of Q-values



We can see that making beta small would increase the speed of the agent finding the optimal way to the goal. Although agent with beta = 6 starts to find shorter ways to get to the goal earlier than agent with beta = 3, agent with beta=6 seems to get stuck and couldn't find the shortest way. This could be explained that as beta gets bigger, it's performance starts to act more like greedy action selection. It doesn't have a far sight and thinks it arrived at the optimal way.

(c) Exploring with increasing beta soft max of Q-value



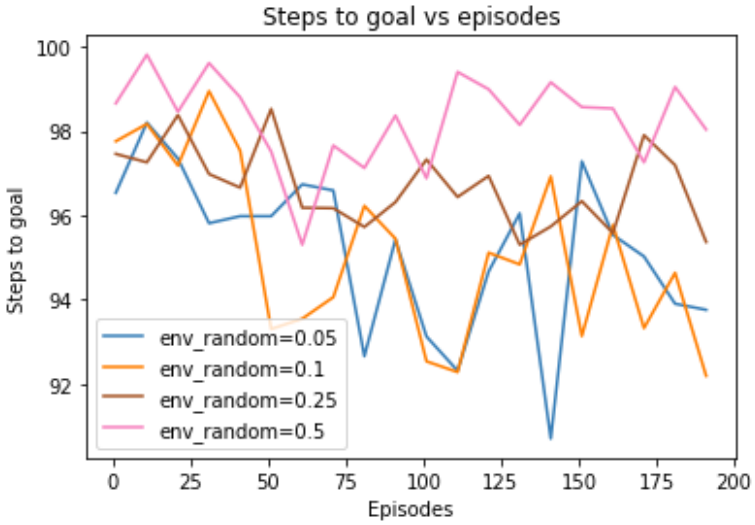
We can see that a smaller k value would find the smallest number of step to get to a goal. This again is explained by the property that as beta approaches infinity, softmax selection will recover greedy action selections. K controls the speed beta approaching infinity as iterations increases, and we can see that the agent with the biggest k value doesn't find a smaller number of steps to goal while all smaller k value agents has found it. The agent with k=0.5 may be stuck since the it starts to select greedily which it will always follow the same pattern of selections. This means that if the agent doesn't find a path with optimal steps before its beta increases to infinity, it will never find the optimal path.

2.3 Stochastic environment

(a) Implemented probabilistic maze environment

```
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
        """
        self.p_random = p_random
        MazeEnv.__init__(self, goals=goals)


    def step(self, a):
        if np.random.random() <= self.p_random:
            a = np.random.choice(np.arange(self.num_actions))
        return MazeEnv.step(self,a)
```

(b) Exploring with different stochastic environments.



It is obviously that a more stochastic environment is harder to explore the optimal path. Since we are using epsilon-greedy policy, sometimes we are taking the best step that we have in our knowledge. However, if sometimes we decided to take the best step, but then due to the stochastic environment and we went on a worst state, it increases our count of steps and also affect our recordings into the agent's knowledge and our future decision makings. All of the agents needs lots of steps to get to the goal, and the shortest path they could get is 91 in the less stochastic environment. This means that agents are either taking steps on every cell on the grid, or making lots of backward movements.

3. Course Evaluation.

I confirm that I submitted my course evaluation.