hw4_sol

December 5, 2019

1 1. Unsupervised Learning

[549]:
```
%matplotlib inline
import scipy
import numpy as np
import itertools
import matplotlib.pyplot as plt
```

1.1 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

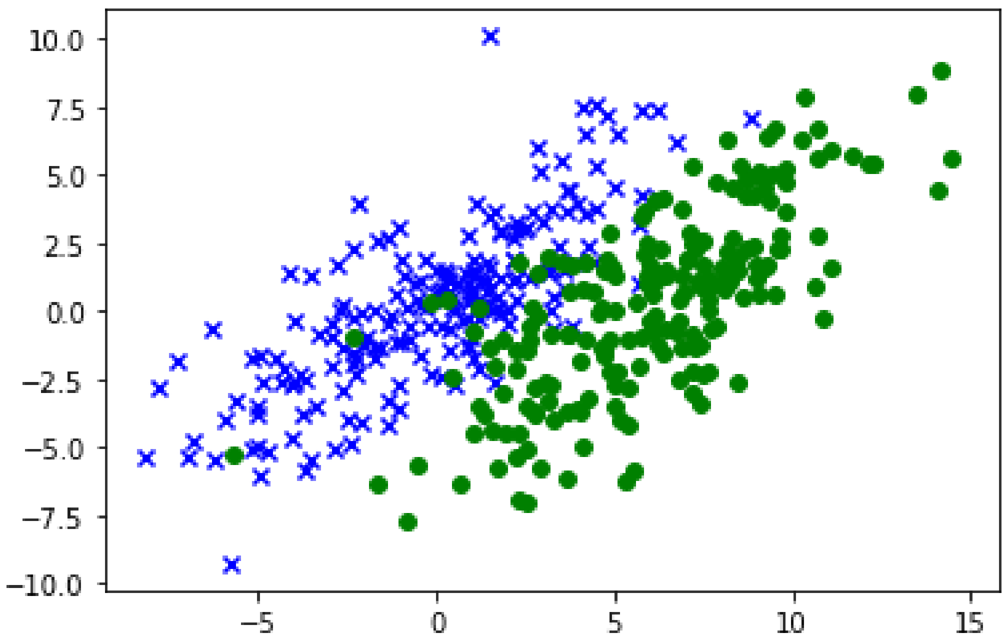
$$\mu_1 = \begin{bmatrix} 0.1 \\ 0.1 \end{bmatrix}, \mu_2 = \begin{bmatrix} 6.0 \\ 0.1 \end{bmatrix} \text{ and } \Sigma_1 = \Sigma_2 = \begin{bmatrix} 10 & 7 \\ 7 & 10 \end{bmatrix}$$

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.

[550]:
```
# 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]

# np.random.seed(311)

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

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

```
[551]: # TODO: Make a scatterplot for the data points showing the true cluster␣
        ↪assignments of each point
       # plt.plot(...) # first class, x shape
       # plt.plot(...) # second class, circle shape
       plt.figure()
       plt.scatter(x_class1[:, 0], x_class1[:, 1], marker='x', c='blue')
       plt.scatter(x_class2[:, 0], x_class2[:, 1], marker='o', c='green')
       plt.show()
```



1.2 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.

[552]:
```
def cost(data, R, Mu):
    N, D = data.shape
    K = Mu.shape[1]
    J = 0
    for k in range(K):
        J += np.dot(np.linalg.norm(data - np.array([Mu[:, k], ] * N),␣
↪axis=1)**2, R[:, k])
    return J
```

[553]:
```
# TODO: K-Means Assignment Step
def km_assignment_step(data, Mu):
    """ Compute K-Means assignment step

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

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

    # Fill this in:
    N, D = data.shape[0], data.shape[1] # Number of datapoints and dimension of␣
↪datapoint
    K = Mu.shape[1] # number of clusters
    R_new = np.zeros((N, K))
    for k in range(K):
        for n in range(N):
            tmps = [np.linalg.norm(data[n,] - Mu.T[k,]) ** 2 for k in range(K)]
            arg_min =  np.argmin(tmps)
            R_new[n,k] = 1 if k == arg_min else 0
    return R_new
```

[554]:
```
# TODO: K-means Refitting Step
def km_refitting_step(data, R, Mu):
    """ Compute K-Means refitting step.

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

    Returns:
        Mu_new: a DxK matrix for the new cluster means locations
    """
    N, D = data.shape[0], data.shape[1] # Number of datapoints and dimension of␣
↪datapoint
```

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

[555]:
```
# 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)

costr = []
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))
    costr.append(cost(data, R, Mu))

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

```
0 30496.113218791685
1 9058.012385540143
2 8068.895699884421
3 7869.281417410331
4 7751.048117101591
5 7714.464199079827
6 7662.450055357289
7 7662.450055357289
8 7662.450055357289
9 7662.450055357289
10 7662.450055357289
11 7662.450055357289
12 7662.450055357289
13 7662.450055357289
14 7662.450055357289
15 7662.450055357289
16 7662.450055357289
```

17 7662.450055357289
18 7662.450055357289
19 7662.450055357289
20 7662.450055357289
21 7662.450055357289
22 7662.450055357289
23 7662.450055357289
24 7662.450055357289
25 7662.450055357289
26 7662.450055357289
27 7662.450055357289
28 7662.450055357289
29 7662.450055357289
30 7662.450055357289
31 7662.450055357289
32 7662.450055357289
33 7662.450055357289
34 7662.450055357289
35 7662.450055357289
36 7662.450055357289
37 7662.450055357289
38 7662.450055357289
39 7662.450055357289
40 7662.450055357289
41 7662.450055357289
42 7662.450055357289
43 7662.450055357289
44 7662.450055357289
45 7662.450055357289
46 7662.450055357289
47 7662.450055357289
48 7662.450055357289
49 7662.450055357289
50 7662.450055357289
51 7662.450055357289
52 7662.450055357289
53 7662.450055357289
54 7662.450055357289
55 7662.450055357289
56 7662.450055357289
57 7662.450055357289
58 7662.450055357289
59 7662.450055357289
60 7662.450055357289
61 7662.450055357289
62 7662.450055357289
63 7662.450055357289
64 7662.450055357289

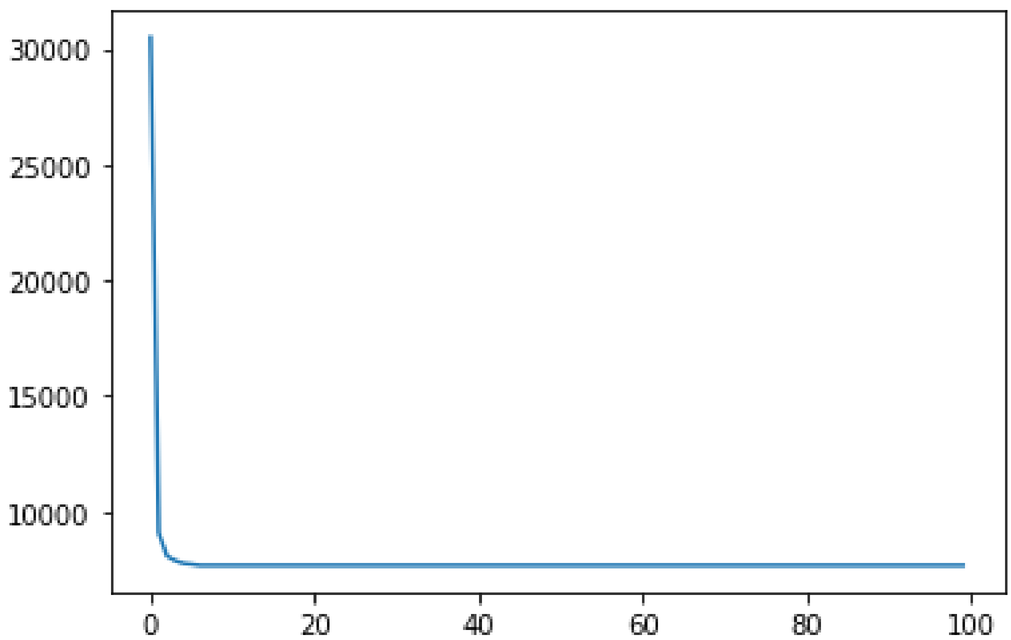
```
65 7662.450055357289
66 7662.450055357289
67 7662.450055357289
68 7662.450055357289
69 7662.450055357289
70 7662.450055357289
71 7662.450055357289
72 7662.450055357289
73 7662.450055357289
74 7662.450055357289
75 7662.450055357289
76 7662.450055357289
77 7662.450055357289
78 7662.450055357289
79 7662.450055357289
80 7662.450055357289
81 7662.450055357289
82 7662.450055357289
83 7662.450055357289
84 7662.450055357289
85 7662.450055357289
86 7662.450055357289
87 7662.450055357289
88 7662.450055357289
89 7662.450055357289
90 7662.450055357289
91 7662.450055357289
92 7662.450055357289
93 7662.450055357289
94 7662.450055357289
95 7662.450055357289
96 7662.450055357289
97 7662.450055357289
98 7662.450055357289
99 7662.450055357289
```

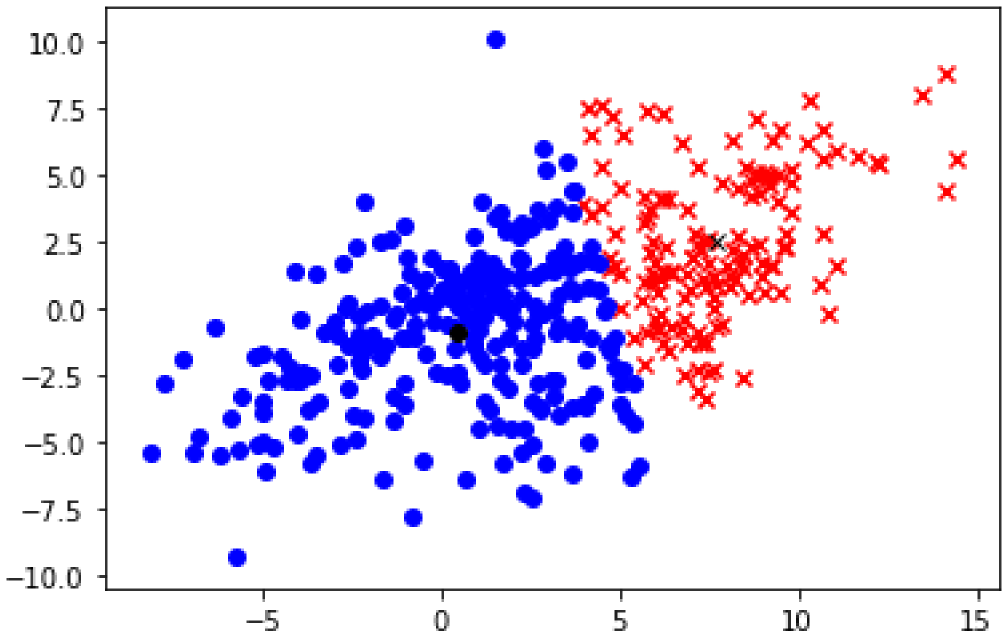
[556]:

```
# TODO: Make a scatterplot for the data points showing the K-Means cluster␣
↪assignments of each point
count = 0
for i in range(N):
    if labels[i] == 0 and np.isin(i, class_1):
        count += 1
    if labels[i] == 1 and np.isin(i, class_2):
        count += 1
count /= data.shape[0]
print("The misclassification rate is: " + str(1 - count))
```

```
plt.plot(costr, label='cost vs iterations')
print('cost vs iterations')
x_1 = data[:, 0][class_1]
x_2 = data[:, 0][class_2]
y_1 = data[:, 1][class_1]
y_2 = data[:, 1][class_2]
plt.figure()
plt.plot(Mu[0,0],Mu[0,1], 'x', color='black')
plt.scatter(x_1, y_1, marker='x', c='red') # first class, x shape
plt.plot(Mu[1,0],Mu[1,1], 'o', color='black')
plt.scatter(x_2, y_2, marker='o', c='blue') # second class, circle shape
plt.show()
```

```
The misclassification rate is: 0.76
cost vs iterations
```





1.3 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:

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

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.

[557]:
```
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))
```

[558]:
```
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):
        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
```

[559]:
```
# 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):
        pi = np.zeros(K)
        for k in range(K):
            pi[k] = Pi[k] * normal_density(data[n], Mu[k], Sigma[k])
        Gamma[n] = pi / np.sum(pi) # Normalize by sum across second dimension␣
↪(mixtures)
    return Gamma
```

[560]:
```
# 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 = np.sum(Gamma, axis=0) # Sum along first axis
    Mu = np.zeros((K, D))
    Sigma = np.zeros((K, D, D))
    comb = np.matmul(Gamma.T, data)
    Pi = Nk / N

    for k in range(K):
        Mu[k] = np.matmul(Gamma.T, data)[k] / Nk[k]
        sig = np.zeros((D, D))
        for n in range(N):
            sig+= Gamma[n, k] * np.matmul((data[n:n+1]-Mu[k:k+1]).T, (data[n:
 ↪n+1]-Mu[k:k+1]))
        Sigma[k] = sig / Nk[k]
    return Mu, Sigma, Pi
```

[561]:
```
# 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.
Mu = Mu.T
Sigma = [np.eye(2), np.eye(2)]
Pi = np.ones(K) / K
Gamma = np.zeros([N, K]) # Gamma is the matrix of responsibilities

max_iter  = 200
log = []

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␣
↪computation longer, but good for debugging
    log.append(log_likelihood(data, Mu, Sigma, Pi))

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

```
0 -37.942332343308024
1 -34.03403874404508
2 -32.18528606866345
3 -31.04404154947671
4 -30.209637033666407
5 -29.50929769193563
6 -28.84073186070178
7 -28.1211293552002
8 -27.263817104837745
9 -26.164588114480917
10 -24.695751668720316
11 -22.713990349178665
12 -20.09264836254087
13 -16.786668006236937
14 -12.922398536453132
15 -8.85874451771884
16 -5.111815607186489
17 -2.1180758796360264
18 -0.030160842079079653
19 1.268443160752335
20 2.006703760945684
21 2.397437548380495
22 2.591171266124376
23 2.680178350011035
24 2.7164584074188594
25 2.727635402041084
26 2.727547914195301
27 2.7225358013147023
28 2.714961911041886
29 2.705108279436382
30 2.6921512743876135
31 2.674619577216179
32 2.6505548590030643
33 2.6174852486295075
34 2.572253468852219
35 2.510693025097954
36 2.427102962930041
37 2.3134287143200902
38 2.1580324846898837
```

39 1.94405136821526
40 1.6481064501342417
41 1.243339683567874
42 0.7204845804184941
43 0.15131791551172735
44 -0.24562233363075403
45 -0.2429972770722908
46 0.14658529679015964
47 0.7545947627245799
48 1.4447726791152342
49 2.14669842902988
50 2.828100926948257
51 3.4735047926914
52 4.074529196819876
53 4.626208764723499
54 5.125732325696482
55 5.572021910619065
56 5.965533638227933
57 6.308060465572112
58 6.602482083896357
59 6.852474345332272
60 7.06221167323336
61 7.236094474402366
62 7.378522817553261
63 7.493725374957793
64 7.585642808214601
65 7.657858665607978
66 7.713568122361243
67 7.75557457030868
68 7.786305135261841
69 7.807837875505243
70 7.82193518197376
71 7.830079485589648
72 7.833508669812061
73 7.833249572808342
74 7.830148677649225
75 7.824899581987742
76 7.818067162890557
77 7.810108553760765
78 7.801391165177152
79 7.792208037976808
80 7.782790835493591
81 7.773320776809043
82 7.76393779428344
83 7.754748172918948
84 7.745830900734956
85 7.73724293077184
86 7.729023528328995

87 7.721197852190443
88 7.7137798964576
89 7.706774900124148
90 7.700181314631063
91 7.693992405135963
92 7.688197548868445
93 7.682783283448106
94 7.677734149238183
95 7.673033362336669
96 7.668663348625107
97 7.664606164063219
98 7.6608438221063535
99 7.657358545481556
100 7.654132956572858
101 7.651150218160243
102 7.648394134185255
103 7.645849218492856
104 7.643500738082179
105 7.6413347362134925
106 7.63933803974122
107 7.637498254242578
108 7.635803749838629
109 7.634243640059527
110 7.632807755647187
111 7.631486614823885
112 7.6302713912401625
113 7.629153880568198
114 7.628126466502627
115 7.627182086760007
116 7.626314199528215
117 7.625516750716726
118 7.624784142252549
119 7.624111201606393
120 7.623493152672515
121 7.622925588080252
122 7.622404442965565
123 7.6219259702352655
124 7.621486717283744
125 7.621083504158782
126 7.620713403129313
127 7.620373719588822
128 7.620061974261273
129 7.619775886636465
130 7.619513359572
131 7.619272464998608
132 7.61905143066799
133 7.618848627875781
134 7.6186625601002325

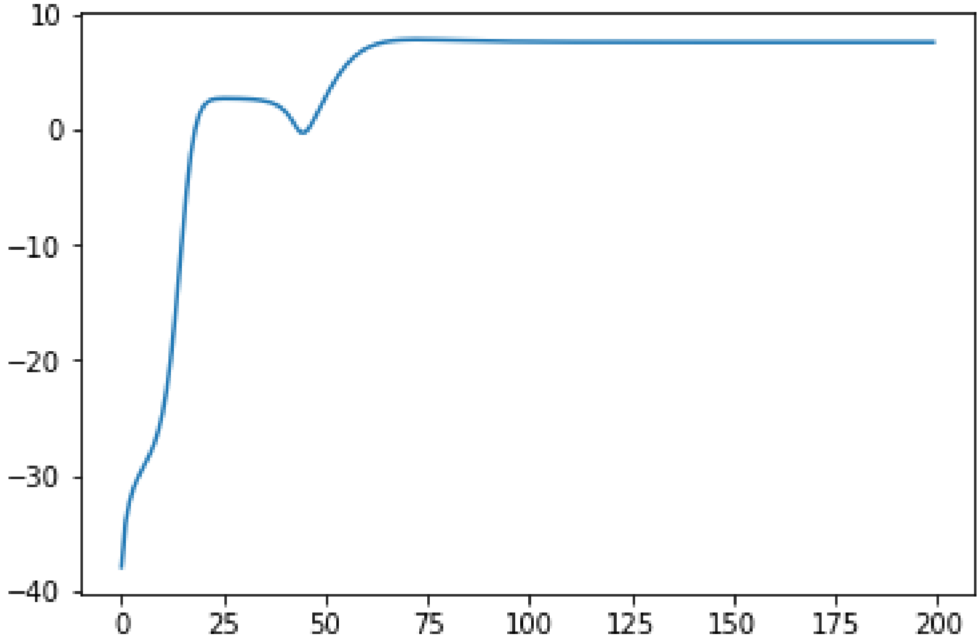
135 7.618491852498847
136 7.618335242201369
137 7.618191569344805
138 7.6180597688076
139 7.617938862578379
140 7.617827952723639
141 7.617726214910622
142 7.617632892442422
143 7.61754729075828
144 7.6174687723805325
145 7.61739675226327
146 7.617330693509303
147 7.617270103436562
148 7.617214529959701
149 7.617163558265953
150 7.617116807754602
151 7.617073929232458
152 7.617034602333392
153 7.616998533145643
154 7.616965452037229
155 7.616935111657346
156 7.616907285100479
157 7.6168817642202065
158 7.616858358085164
159 7.616836891555126
160 7.616817203979339
161 7.616799148002697
162 7.616782588463767
163 7.61676740138858
164 7.616753473065514
165 7.616740699200781
166 7.616728984134201
167 7.616718240127043
168 7.616708386710139
169 7.616699350076001
170 7.6166910625335875
171 7.616683461999299
172 7.616676491532818
173 7.61667009891009
174 7.616664236237041
175 7.61665885958537
176 7.6166539286690105
177 7.616649406540381
178 7.616645259313044
179 7.6166414559082645
180 7.616637967824914
181 7.616634768924193
182 7.6166318352319715

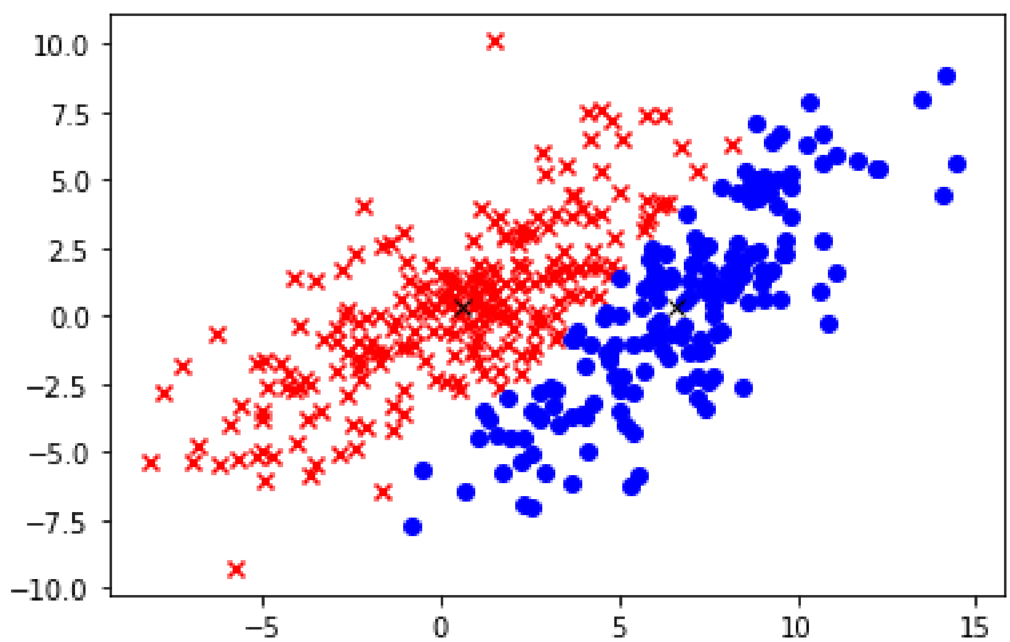
```
183 7.616629144762488
184 7.616626677352287
185 7.6166244145092445
186 7.616622339274562
187 7.616620436092656
188 7.616618690702672
189 7.616617090019975
190 7.616615622050205
191 7.6166142757880735
192 7.616613041144556
193 7.616611908865105
194 7.616610870463505
195 7.616609918155726
196 7.616609044804718
197 7.616608243863287
198 7.616607509329319
199 7.61660683569574
```

[562]:

```python
# TODO: Make a scatterplot for the data points showing the Gaussian Mixture␣
 ↪cluster assignments of each point
count = 0
for i in range(N):
    if labels[i] == 0 and np.isin(i, class_1):
        count += 1
    if labels[i] == 1 and np.isin(i, class_2):
        count += 1
count /= data.shape[0]
print("The misclassification rate is: " + str(1 - count))

plt.plot(log, label='log-likelihood vs iterations')
print('log-likelihood vs iterations')

x_1 = data[:, 0][class_1]
x_2 = data[:, 0][class_2]
y_1 = data[:, 1][class_1]
y_2 = data[:, 1][class_2]
plt.figure()
plt.plot(Mu[0,0],Mu[0,1], 'x', color='black')
plt.scatter(x_1, y_1, marker='x', c='red') # first class, x shape
plt.plot(Mu[1,0],Mu[1,1], 'x', color='black')
plt.scatter(x_2, y_2, marker='o', c='blue') # second class, circle shape
plt.show()
```

```
The misclassification rate is: 0.10499999999999998
log-likelihood vs iterations
```





1.4 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?

TODO: Your written answer here

a) The performance of k-Means and EM based on the resulting cluster is pretty similar, with K-means has slightly higher misclassification rate and both were aroung 50%.

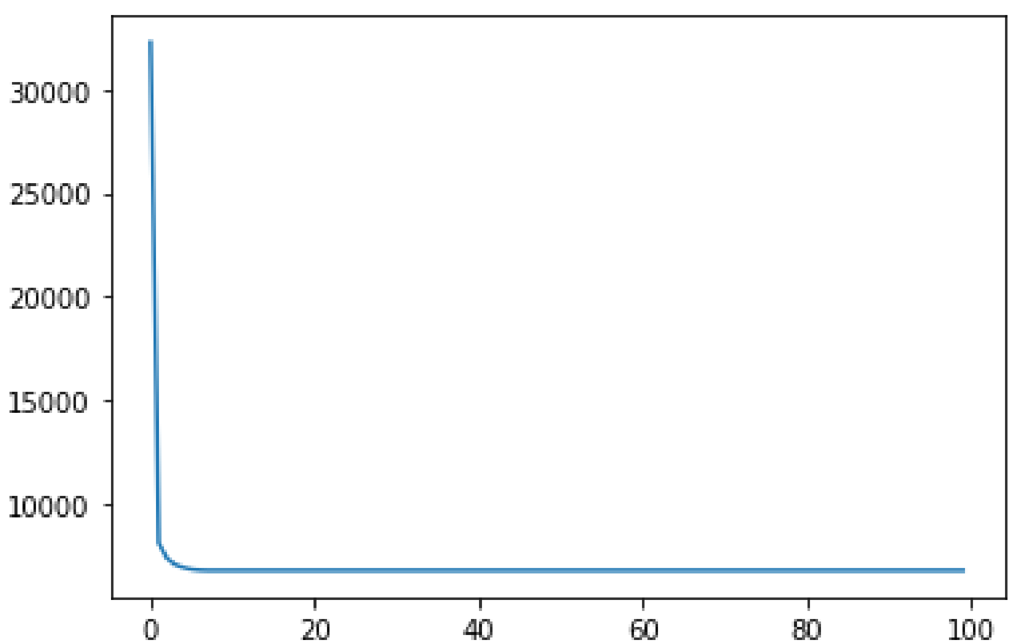
b) From the data above, the K-means algorithm converges much faster than the EM algorithm, while the second algorithm has better performance. According to the algorithm, the bottleneck for K-means is the assignment step since it has two for loops (give the assignments for every points). As for the EM algorithm, the bottleneck can be the M step, since it requires more matrix multiplications than the E step.

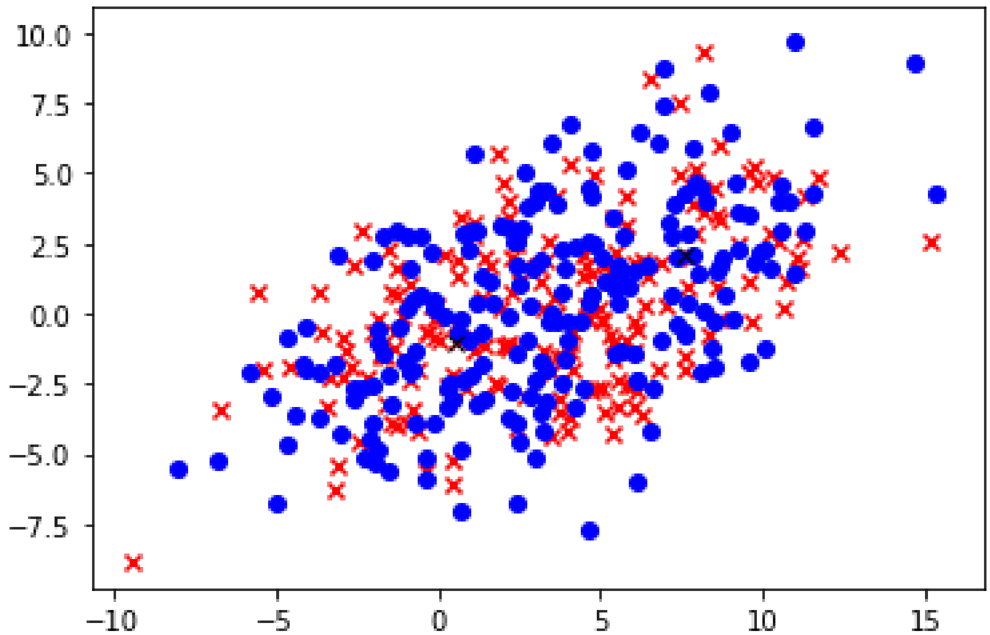
c) After several experimetns, I found out that the algorithm performance do depend on different realization of data. The performances of EM algorithm are generally more stable and better than K-means algorithm. However, in some situations like seed = 373, the EM algorithm produced a weird assignment while K-Means method works normally.

[564]:

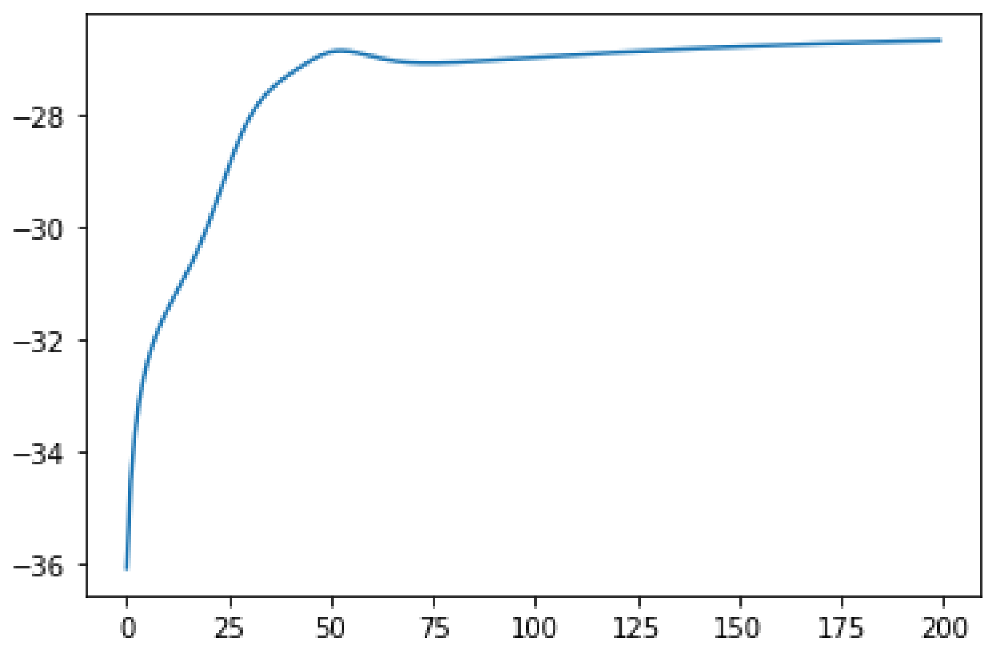
```
```

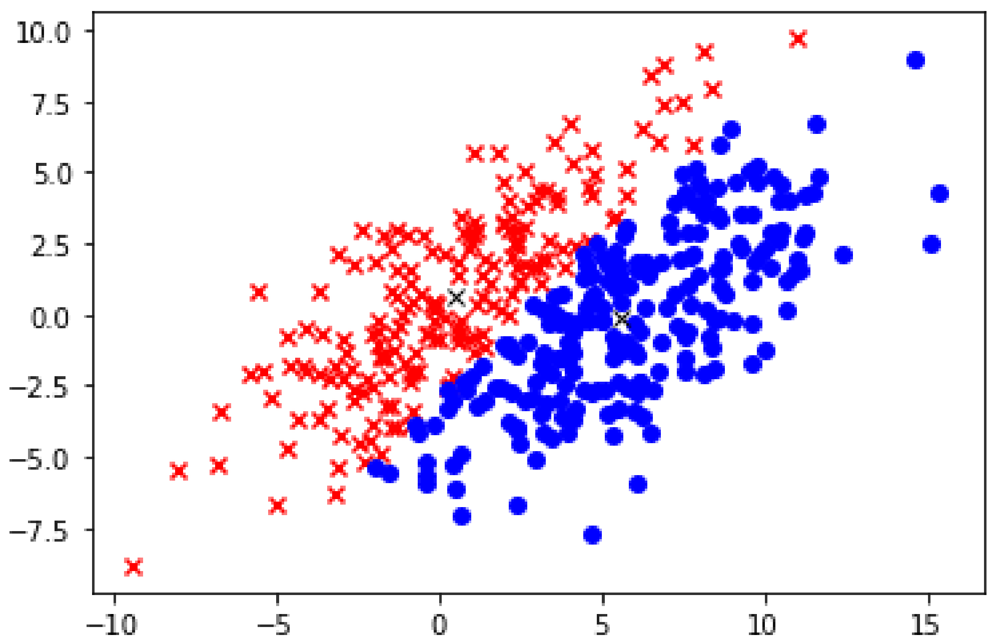
```
Seed = 311
The misclassification rate is: 0.49
cost vs iterations
```



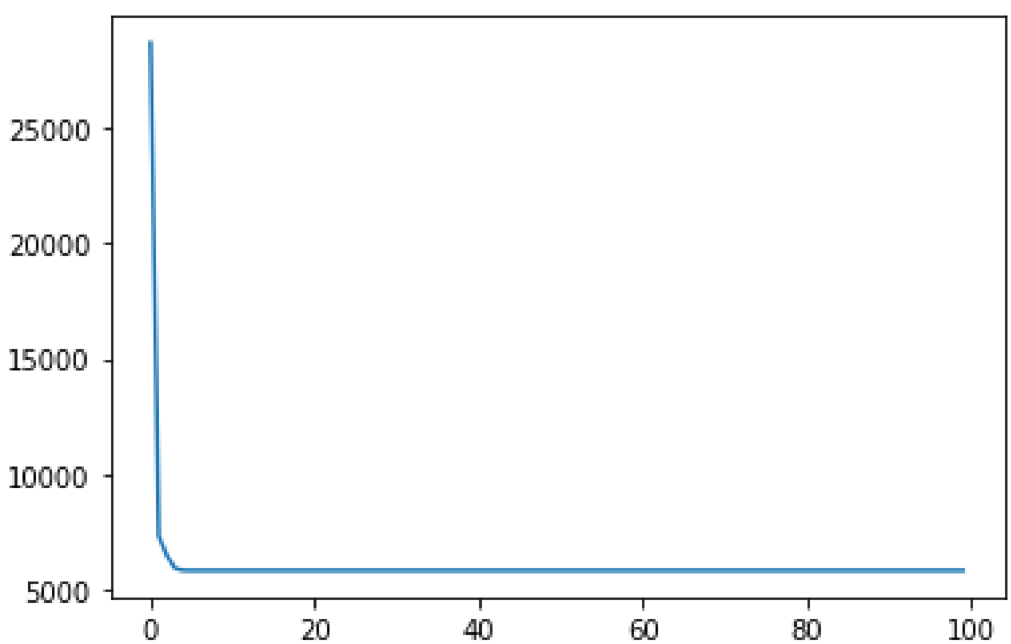


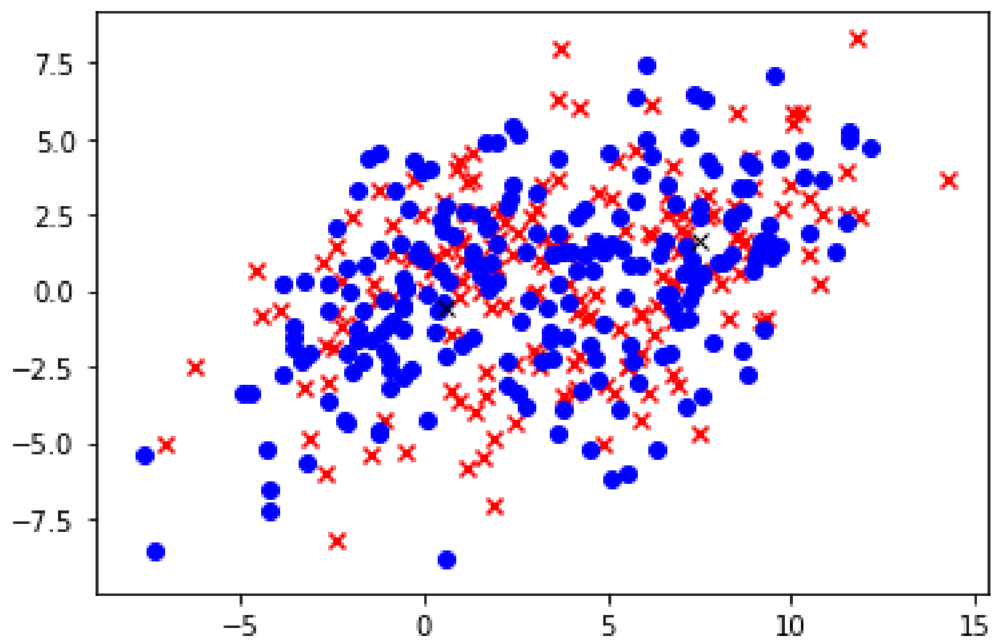

```
The misclassification rate is: 0.135
log-likelihood vs iterations
```



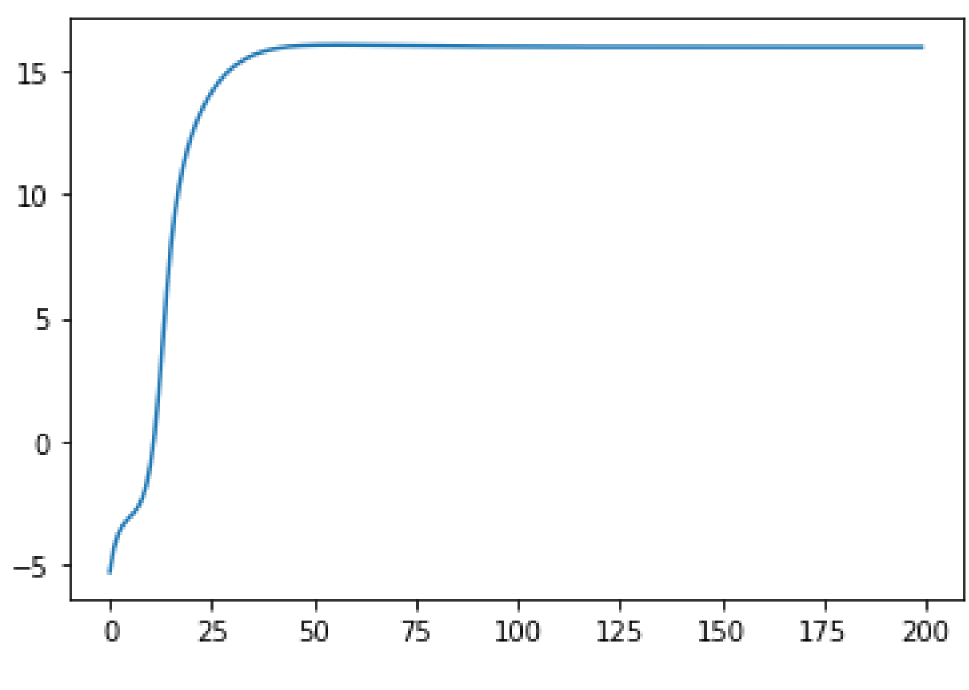


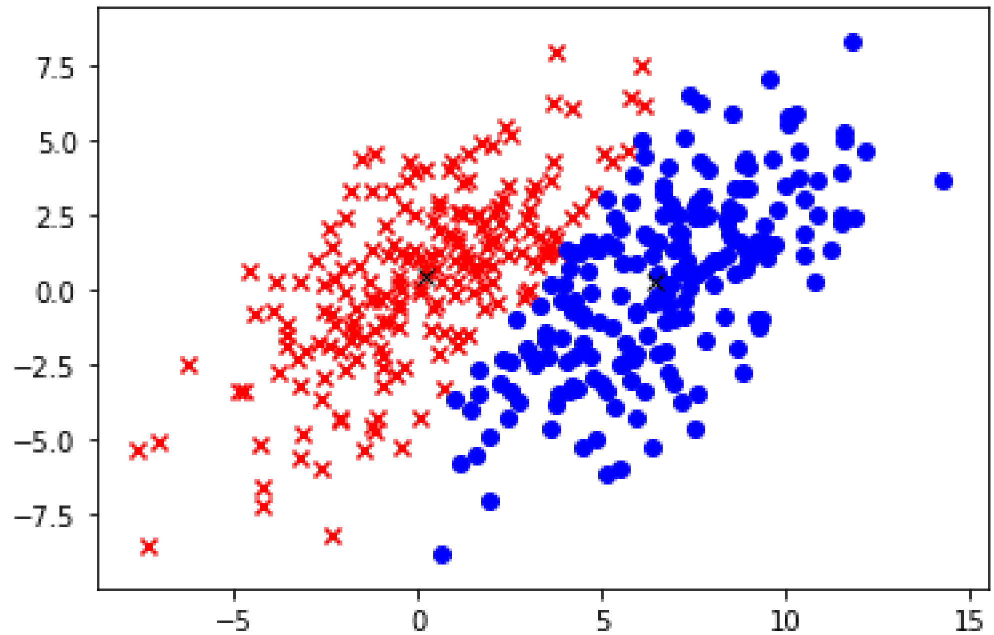

```
Seed = 411
The misclassification rate is: 0.5
cost vs iterations
```



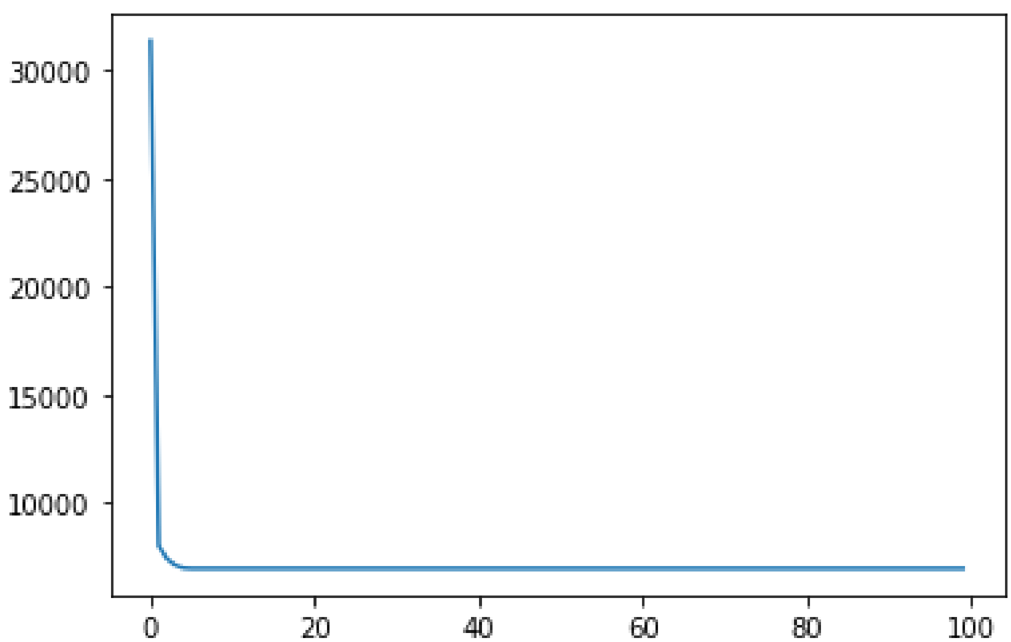


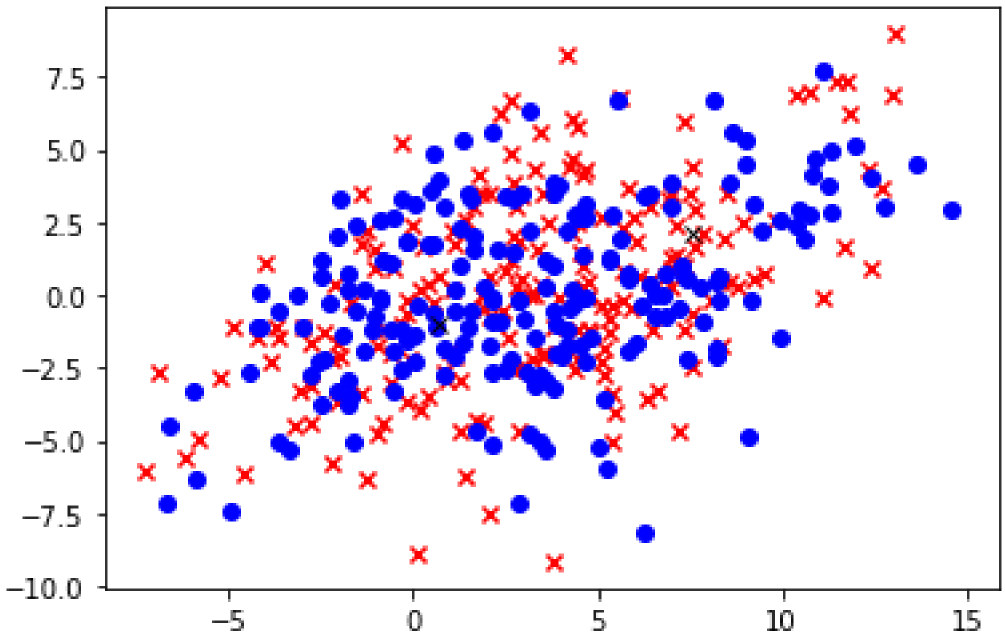

```
The misclassification rate is: 0.06999999999999995
log-likelihood vs iterations
```



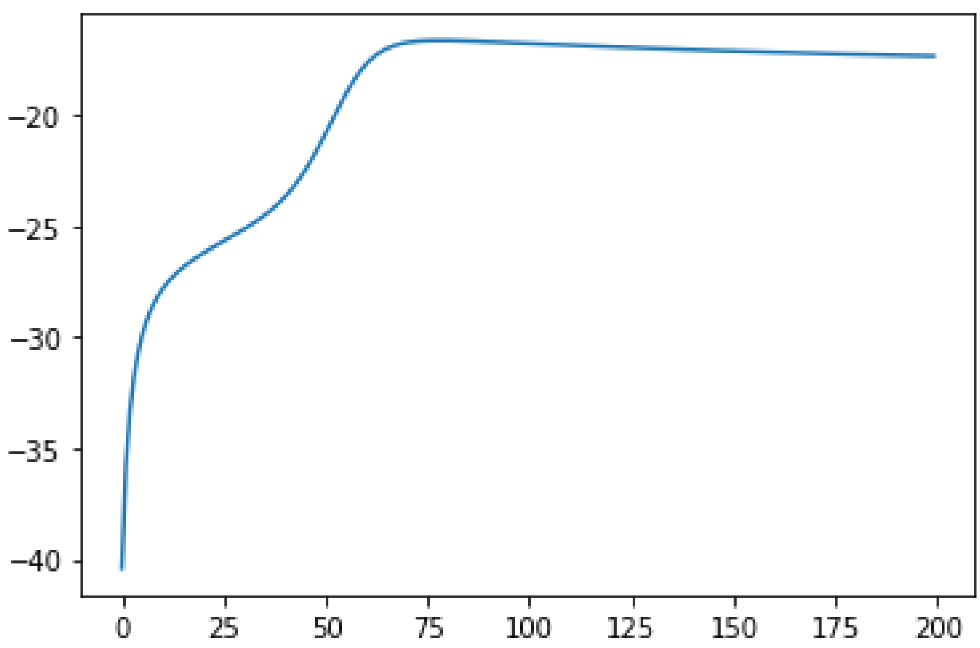


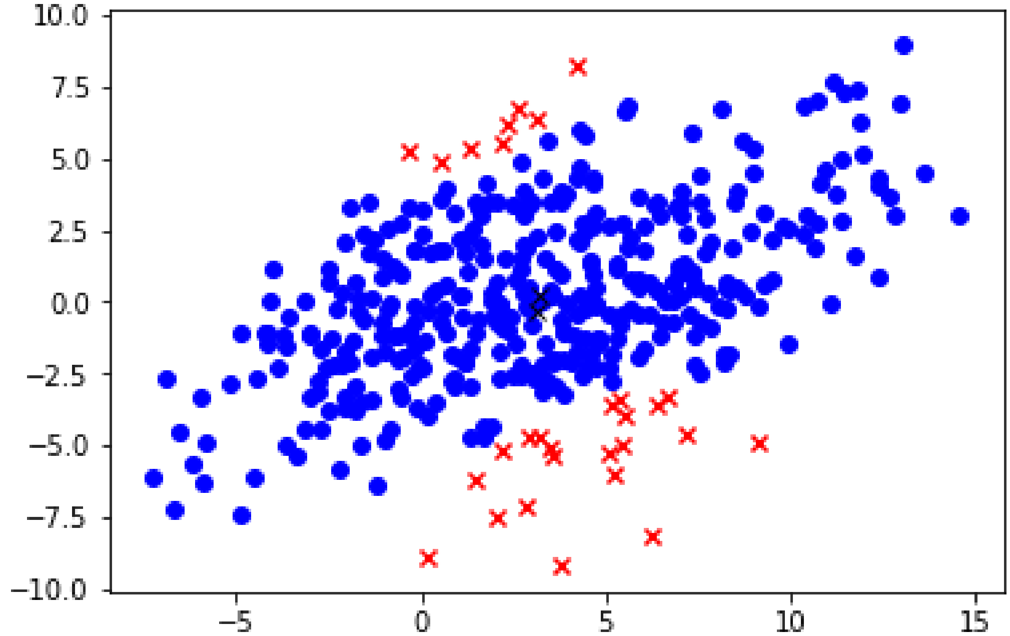

```
Seed = 373
The misclassification rate is: 0.515
cost vs iterations
```



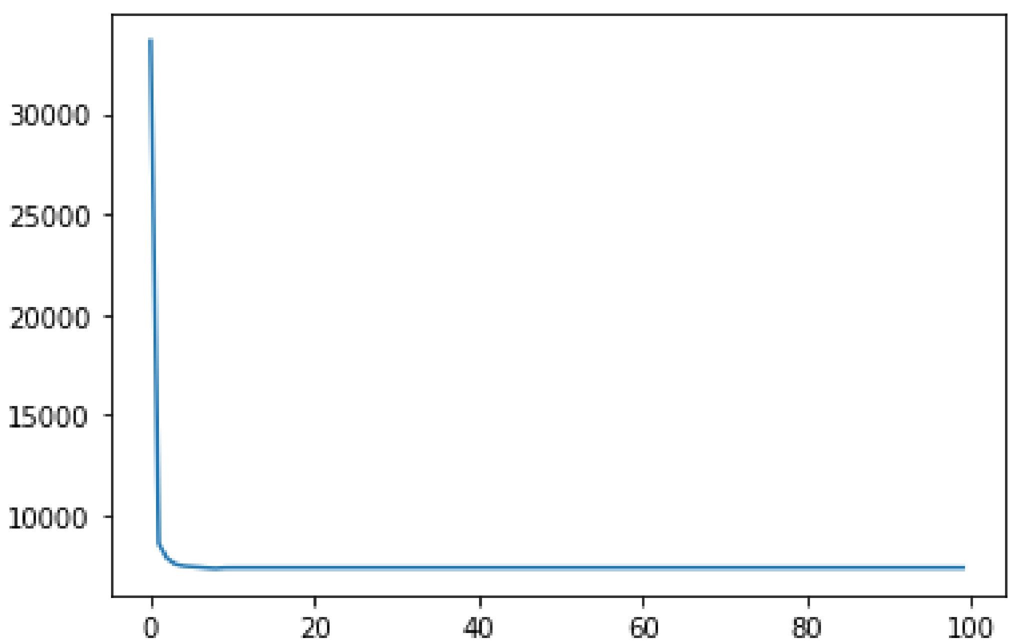


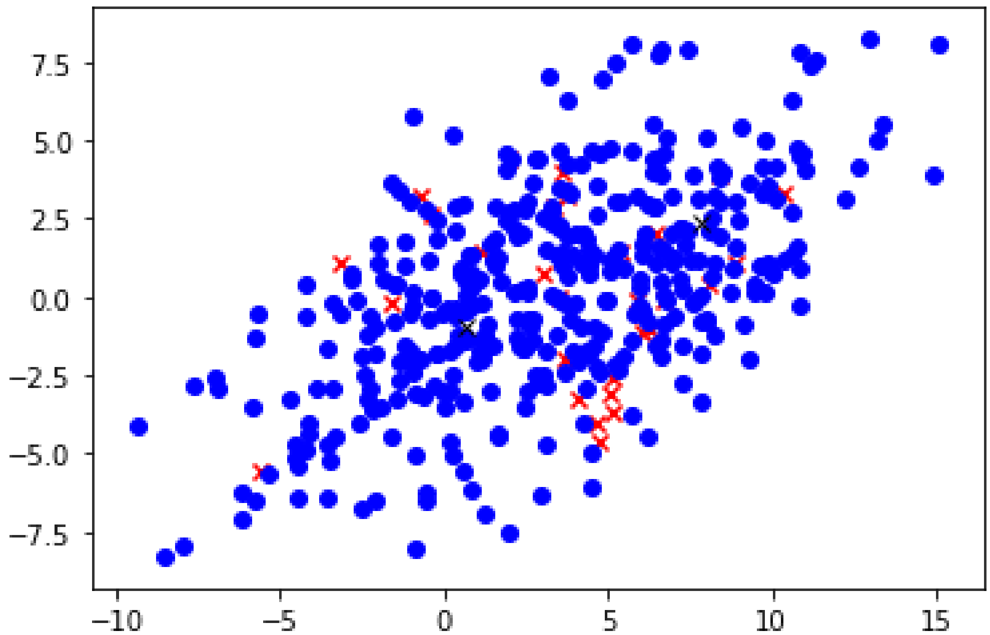

```
The misclassification rate is: 0.5325
log-likelihood vs iterations
```



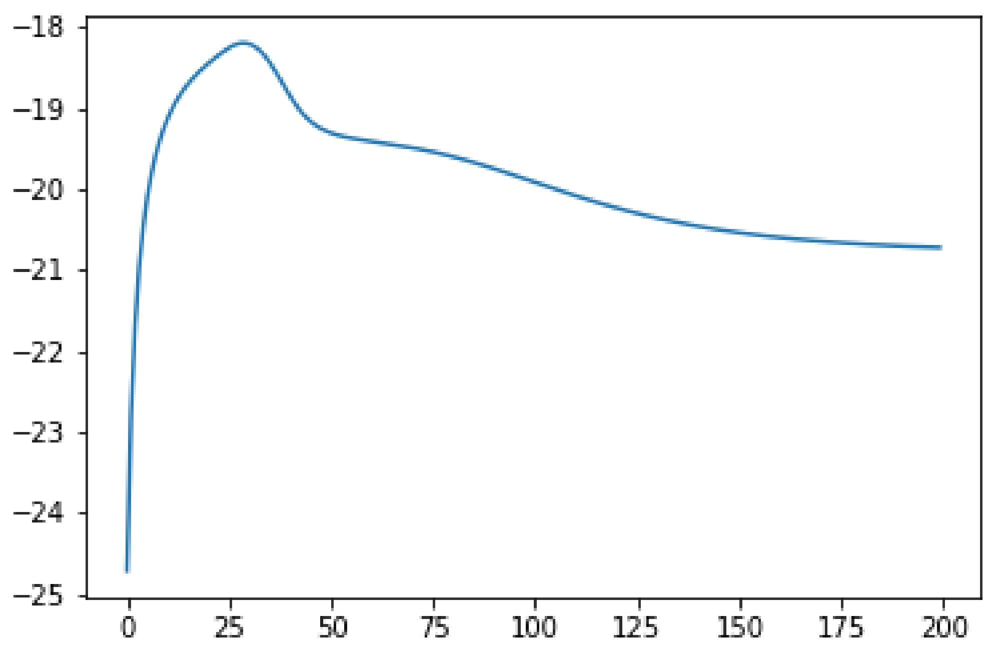


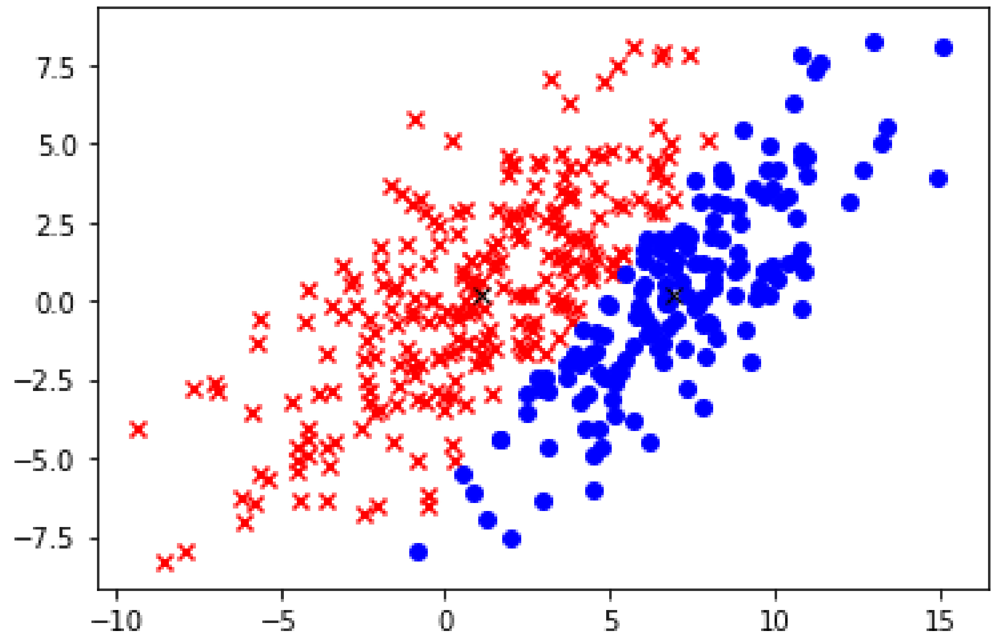

```
Seed = 320
The misclassification rate is: 0.5125
cost vs iterations
```



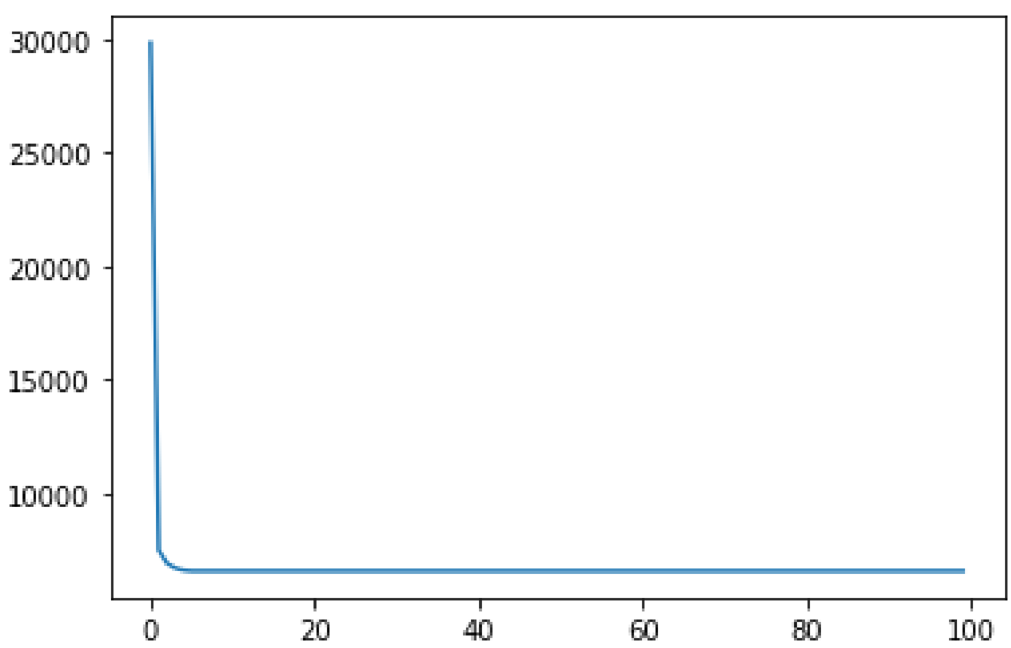


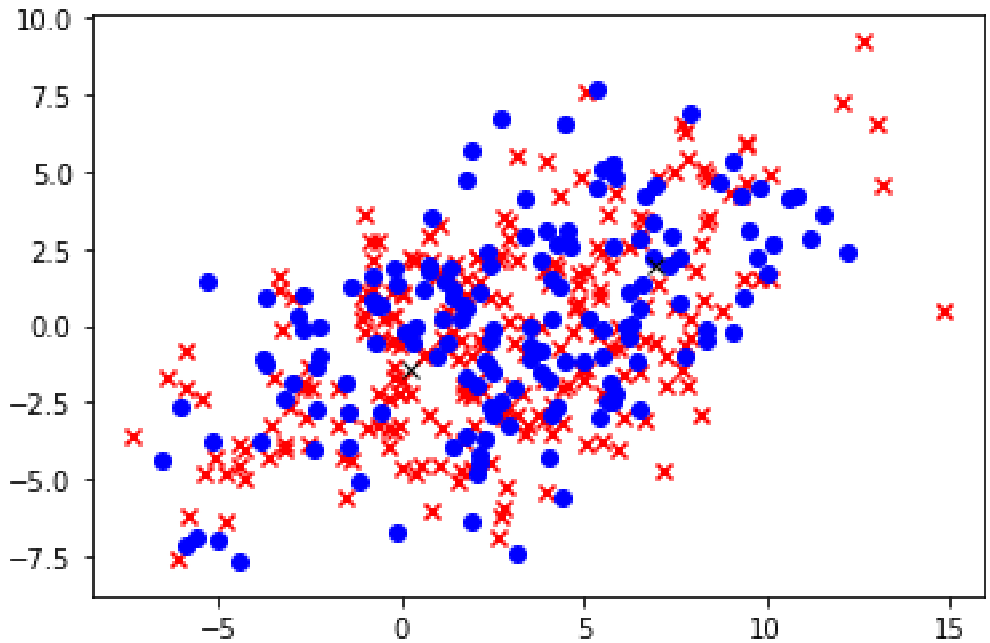

```
The misclassification rate is: 0.13749999999999996
log-likelihood vs iterations
```



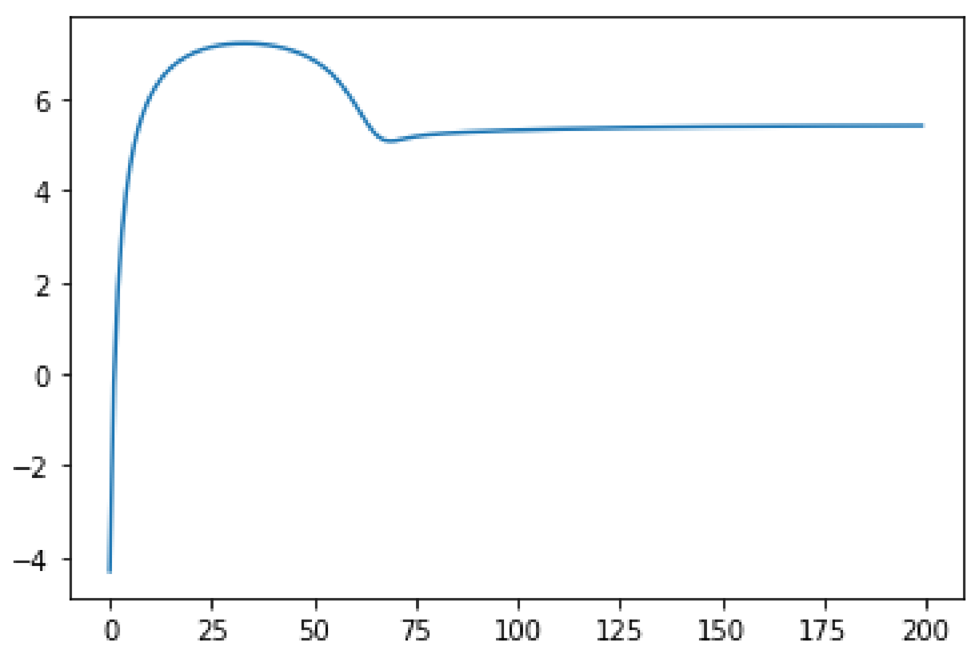


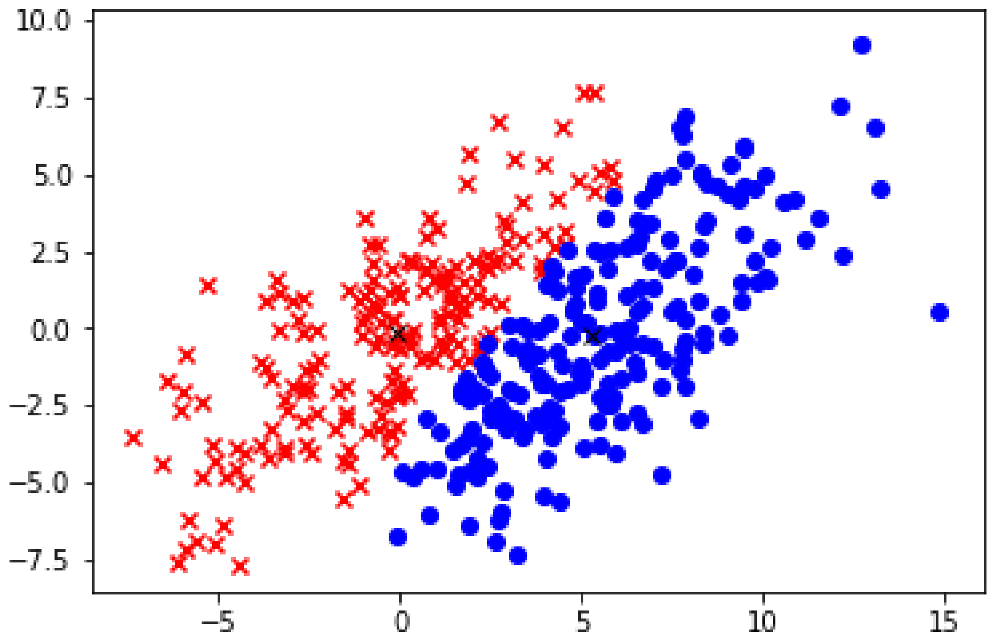

```
Seed = 384
The misclassification rate is: 0.49750000000000005
cost vs iterations
```






```
The misclassification rate is: 0.09499999999999997
log-likelihood vs iterations
```





2 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`

[875]:
```
from qlearning import qlearn
from maze import MazeEnv, ProbabilisticMazeEnv
from plotting_utils import plot_steps_vs_iters, plot_several_steps_vs_iters,␣
 ↪plot_policy_from_q

import numpy as np
import math
import copy
from random import randint

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

        # 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
            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_steps, 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 = np.argmax(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] = q_hat[curr_state, action]  + alpha␣
↪* (reward + gamma * q_hat[next_state, new_value] - q_hat[curr_state, action])

                # TODO: Update the current staet 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
    # ...
    if np.sum(q_hat[state]) == 0 or np.random.uniform(0, 1) < epsilon:
        action = randint(0, action_space_size - 1)
    else:
        action = np.argmax(q_hat[state])
    return action

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

    Args:
        q_hat: A Q-value table shaped [num_rows, num_col, num_actions] for
            grid environment with num_rows rows and num_col columns
        beta (float): Parameter for controlling the stochasticity of the action
        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
    if np.sum(q_hat[state]) == 0:
        return randint(0, 3)
    l = stable_softmax(beta * q_hat[state], 0)
    r = np.random.choice(4, 1, p=l)
    return r

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

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

```
                = e^x / (e^max(x) * sum(e^x/e^max(x)))

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

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

    return output
```

2.1 1. Basic Q Learning experiments

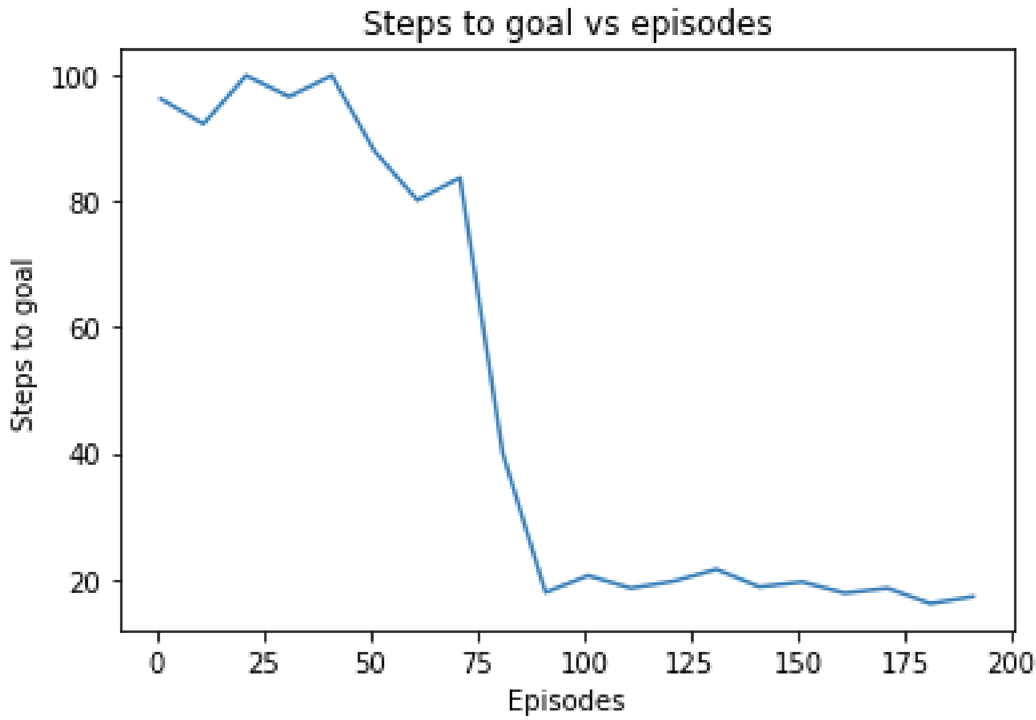
(a) Run your algorithm several times on the given environment. Use the following hyperparameters:

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.

[839]:
```
# TODO: Fill this in
num_iters = 200
alpha = 1.0
gamma = 0.9
epsilon = 0.1
max_steps = 100
use_softmax_policy = False

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

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

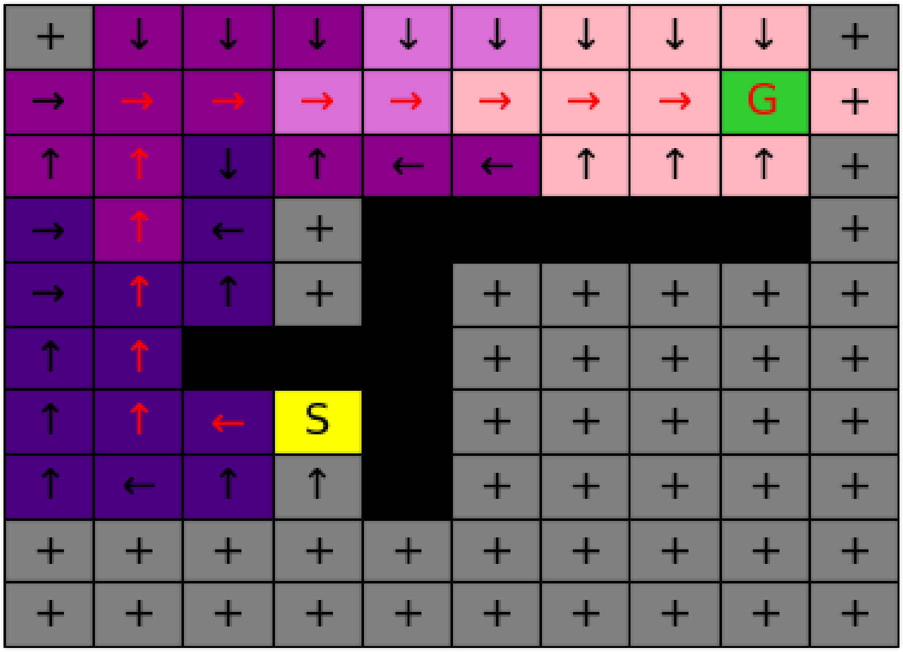
Plot the steps to goal vs training iterations (episodes):

[840]:
```
# TODO: Plot the steps vs iterations
plot_steps_vs_iters(steps_vs_iters)
```



Visualize the learned greedy policy from the Q values:

[841]:
```
# TODO: plot the policy from the Q value
plot_policy_from_q(q_hat, env)
```



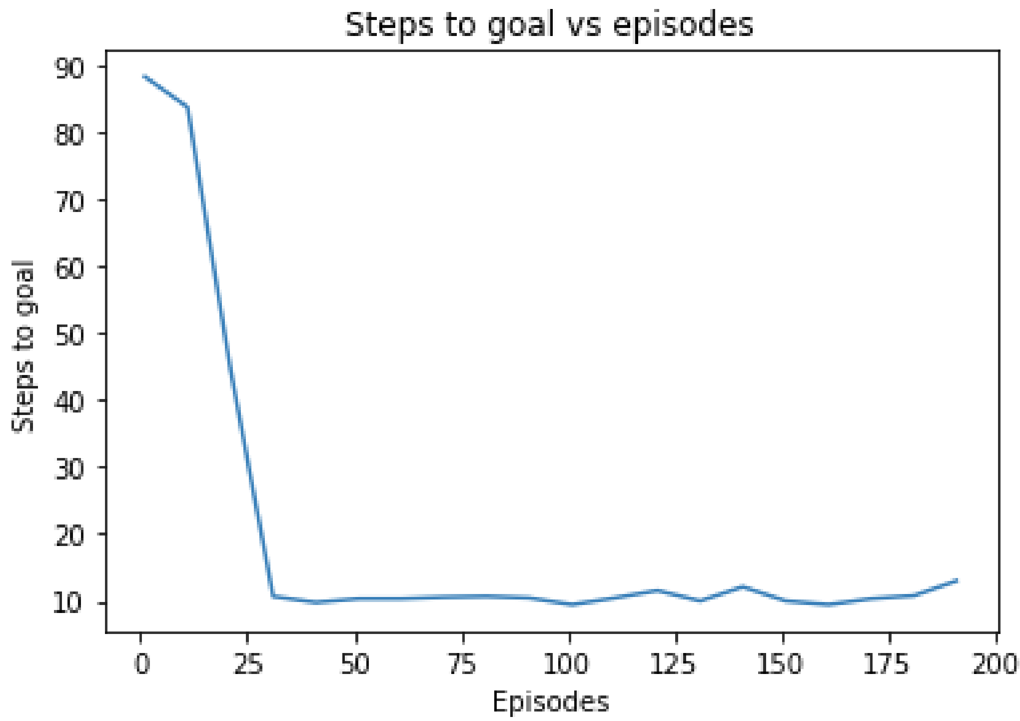
<Figure size 720x720 with 0 Axes>

(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.

[842]:
```
# 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(start=[0, 0], goals=goal_locs)

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

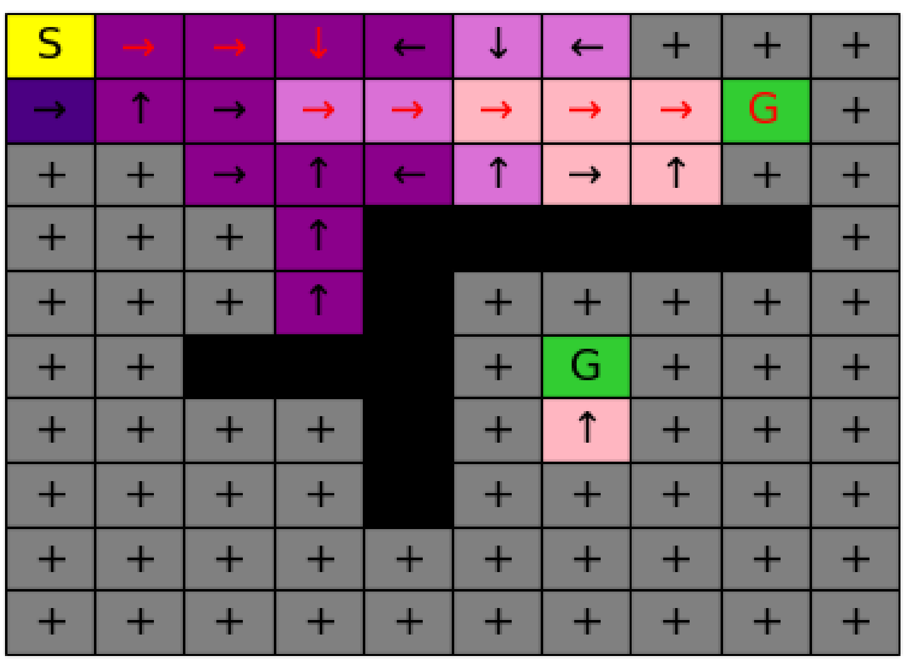
Plot the steps to goal vs training iterations (episodes):

[843]:
```
# TODO: Plot the steps vs iterations
plot_steps_vs_iters(steps_vs_iters)
```



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

[844]:
```
# TODO: plot the policy from the Q values
plot_policy_from_q(q_hat, env)
```



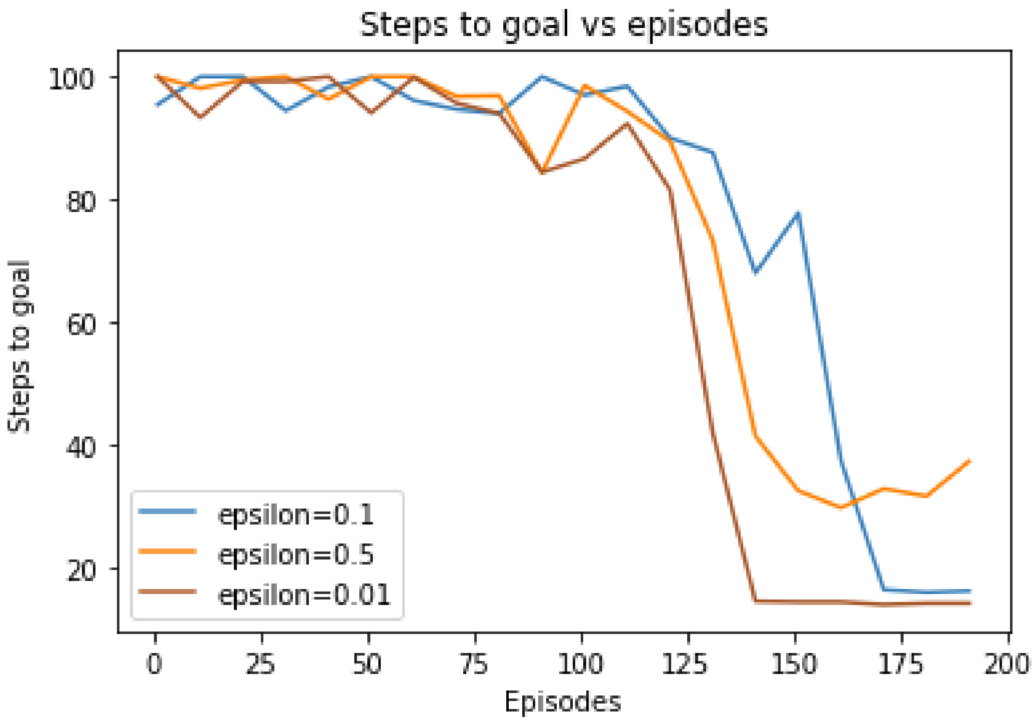
```
<Figure size 720x720 with 0 Axes>
```

3 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.

[873]:
```
# 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 = [0.1, 0.5, 0.01]

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, init_beta=None, k_exp_sched=None)
    steps_vs_iters_list.append(steps_vs_iters)
```

[874]:
```
# 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)
```



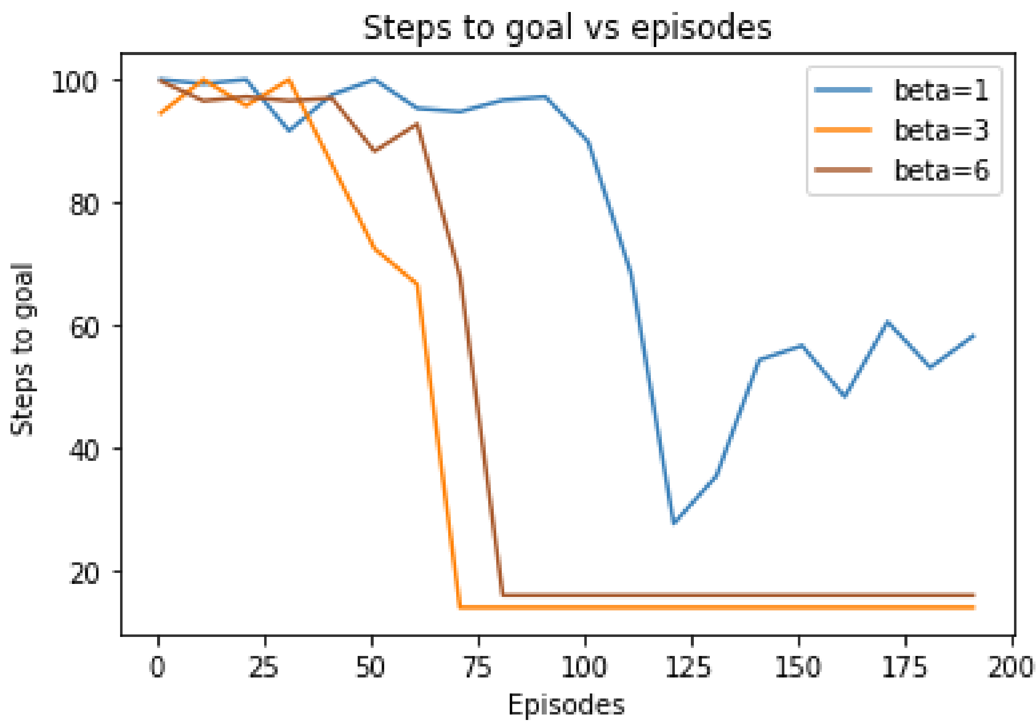
From the graph shown above, the benefit for lower epsilon values is that they are faster to get to the goal location. The cost for them is that they are more deterministic. For the high epsilon values, the cost is that it generally takes longer for than to get the correct location, while the benefit can be they are more random, so it has larger probability to find some shortcuts and get to the goal faster.

(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.

[876]:
```
# 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 # (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)
```

[877]:
```
label_list = ["beta={}".format(beta) for beta in beta_list]
# TODO:
plot_several_steps_vs_iters(steps_vs_iters_list, label_list)
```



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

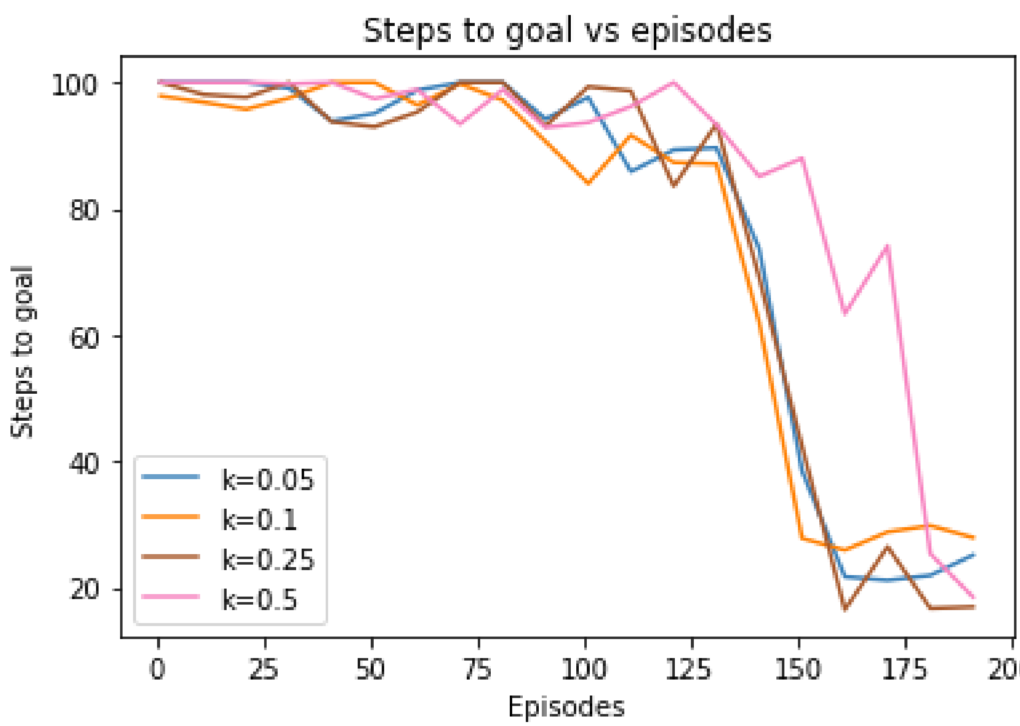
$$\beta(t) = \beta_0 e^{kt}$$

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

[878]:
```
# TODO: Fill this in for Dynamic Beta
num_iters = 200
alpha = 1.0
gamma = 0.9
```

```
epsilon = 0.1
max_steps = 100

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

steps_vs_iters_list = []
for k_exp_schedule in k_exp_schedule_list:
    q_hat, steps_vs_iters = qlearn(env, num_iters, alpha, gamma, epsilon,␣
 ↪max_steps, use_softmax_policy, beta, k_exp_schedule)

    steps_vs_iters_list.append(steps_vs_iters)
```

[879]:
```
# 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)
```



With fixed beta value, not every beta value can get to the goal location, at least with 200 steps. For example, the beta=1 curve ends up with going upwards, which means it goes far away from the correct location. However, with dynamic beta, the learning process can be slower than the fixed beta, but step to goal for all k values are decreasing.

3.1 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.

[884]:
```python
# TODO: Implement ProbabilisticMazeEnv in maze.py
import numpy as np
import copy
import math

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

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


class MazeEnv:

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

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

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

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

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

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

        Args:
            a (int): action integer

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

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

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

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

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

    return state, reward, done

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

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

        Returns:
            is_valid (bool)
    """
```

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

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

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

        return True

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

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

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

    @property
    def _get_goal_state(self):
        """ Get the start state
        """
        goals = []
        for goal in self.goals:
            goals.append(self.get_state_from_coords(goal[0], goal[1]))
        return goals

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

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

```python
        return state

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


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

    def __init__(self, goals=[[2, 8]], p_random=0):
        """ Probabilistic Maze Environment

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

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

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

        if np.random.uniform(0, 1) <= self.p_random:
            a = randint(0, 3)

        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
```

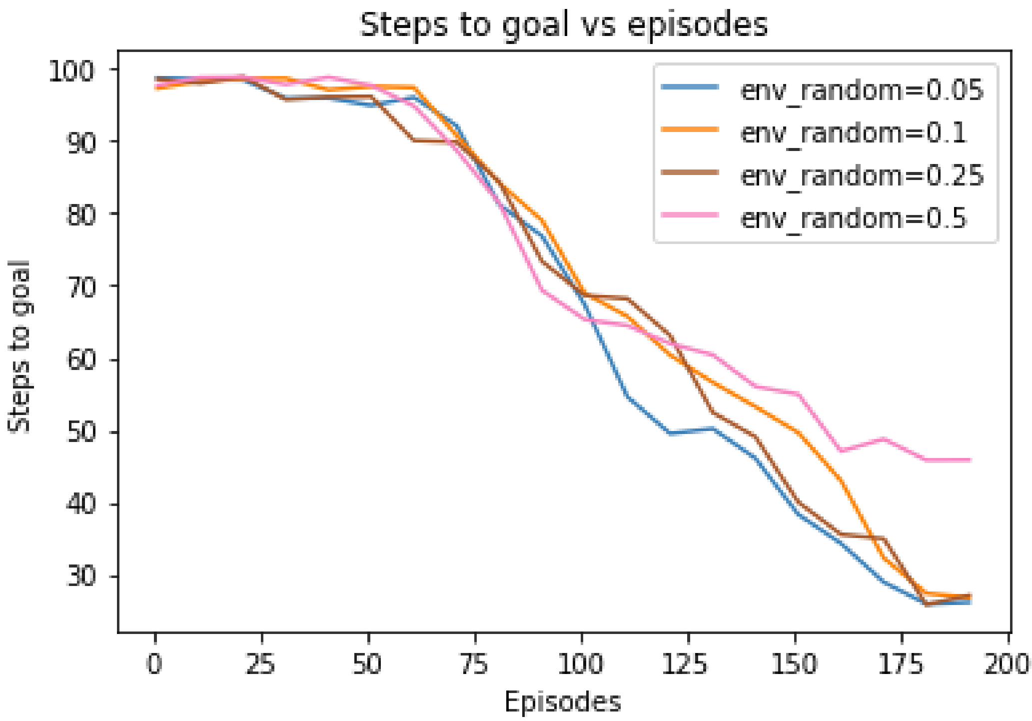
```
        state = self.get_state_from_coords(self.obs[0], self.obs[1])

        return state, reward, done
```

(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.

[887]:
```
# TODO: Use the same parameters as in the first part, except change alpha
num_iters = 200
alpha = 0.5
gamma = 0.9
epsilon = 0.1
max_steps = 100
use_softmax_policy = False

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

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

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

[888]:
```
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)
```



With some probabilities to step a random move, the curves are more unstable and fluctuated around the trend, since they are taking accound some non-determinism. Moreover, it is also obvious that instead of suddenly drop from high steps to goal value to low steps to goal value, the shape of the curves with more stochastic environment are more gentle.

[883]:
```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
# from qlearning import *
# from maze import *

#  UTILITY FUNCTIONS


color_cycle = ['#377eb8', '#ff7f00', '#a65628',
               '#f781bf','#4daf4a',  '#984ea3',
               '#999999', '#e41a1c', '#dede00']

def plot_steps_vs_iters(steps_vs_iters, block_size=10):
    num_iters = len(steps_vs_iters)
    block_size = 10
    num_blocks = num_iters // block_size
    smooted_data = np.zeros(shape=(num_blocks, 1))
    for i in range(num_blocks):
```

```
        lower = i * block_size
        upper = lower + 9
        smooted_data[i] = np.mean(steps_vs_iters[lower:upper])

    plt.figure()
    plt.title("Steps to goal vs episodes")
    plt.ylabel("Steps to goal")
    plt.xlabel("Episodes")
    plt.plot(np.arange(1,num_iters,block_size), smooted_data,␣
 ↪color=color_cycle[0])

    return

def plot_several_steps_vs_iters(steps_vs_iters_list, label_list, block_size=10):
    smooted_data_list = []
    for steps_vs_iters in steps_vs_iters_list:
        num_iters = len(steps_vs_iters)
        block_size = 10
        num_blocks = num_iters // block_size
        smooted_data = np.zeros(shape=(num_blocks, 1))
        for i in range(num_blocks):
            lower = i * block_size
            upper = lower + 9
            smooted_data[i] = np.mean(steps_vs_iters[lower:upper])
        smooted_data_list.append(smooted_data)

    plt.figure()
    plt.title("Steps to goal vs episodes")
    plt.ylabel("Steps to goal")
    plt.xlabel("Episodes")
    index = 0
    for label, smooted_data in zip(label_list, smooted_data_list):
        plt.plot(np.arange(1,num_iters,block_size), smooted_data, label=label,␣
 ↪color=color_cycle[index])
        index += 1
    plt.legend()

    return


# this function sets color values for
# Q table cells depending on expected reward value
def get_color(value, min_val, max_val):

    switcher={
                0:'gray',
                1:'indigo',
```

```
                2:'darkmagenta',
                3:'orchid',
                4:'lightpink',
             }

    step = (max_val-min_val)/5
    i = 0
    color='lightpink'

    for limit in np.arange(min_val, max_val, step):
        if limit <= value < limit+step:
            color = switcher.get(i)
        i+=1
    return color



# get first cell out of the start state
def get_next_cell(x1,x2,heatmap,policy_table,xlim=9,ylim=9):
    up_reward=-10000
    down_reward=-10000
    left_reward=-10000
    right_reward=-10000

    if (x1<ylim):
        if (policy_table[x1-1][x2]!=3):
            up_reward = heatmap[x1-1][x2]
    else:
        up_reward = -1000

    if (x1>0):
        if (policy_table[x1+1][x2]!=0):
            down_reward = heatmap[x1+1][x2]
    else:
        down_reward = -1000

    if (x2>0):
        if (policy_table[x1][x2-1]!=1):
            left_reward = heatmap[x1][x2-1]

    else:
        left_reward = -1000

    if (x2<xlim):
        if (policy_table[x1][x2+1]!=2):
            right_reward = heatmap[x1][x2+1]
```

```python
    else:
        right_reward = -1000

    rewards = np.array([up_reward, down_reward, left_reward, right_reward])
    idx = np.argmax(rewards)
    next_cell = [(x1-1,x2), (x1+1,x2), (x1,x2-1), (x1,x2+1)][idx]
    choice = ['up', 'down', 'left', 'right']
    #print ('picking ',choice[idx])
    return next_cell




# get coordinates of the cells
# on the way from the start to goal state
def get_path(x1,x2, policy_table):
    x_coords = [x1]
    y_coords = [x2]
    x1_new = x1
    x2_new = x2

    i=0
    num_steps = 0
    total_cells = len(policy_table)*len(policy_table[0])
    while (policy_table[x1][x2]!='G') and num_steps < total_cells:
        if (policy_table[x1][x2]==1): # right
            x2_new=x2+1
            #print(i, ' - moving right')

        elif (policy_table[x1][x2]==0):
            x1_new=x1-1
            #print(i, ' - moving up')

        elif (policy_table[x1][x2]==3):
            x1_new=x1+1
            #print(i, ' - moving down')

        elif (policy_table[x1][x2]==2):
            x2_new=x2-1
            #print(i, ' - moving left')

        x1 = x1_new
        x2 = x2_new
        x_coords.append(x1)
        y_coords.append(x2)
        num_steps += 1
    return x_coords, y_coords
```

```python
# plot Q table
# optimal path is highlighted and cells colored by their values
def plot_table(env, table_data, heatmap, goal_states, start_state, max_val,␣
 ↪min_val, x_coords, y_coords):
    fig = plt.figure(dpi=80)
    ax = fig.add_subplot(1,1,1)
    plt.figure(figsize=(10,10))

    width = len(table_data[0])
    height = len(table_data)

    new_table = []

    for i in range(height):
        new_row = []

        for j in range(width):
            if env.map[i][j] == 0:
                new_row.append('')
            else:
                digit = table_data[i][j]
                if (digit==0):
                    new_row.append('\u2191') # up
                elif (digit==1):
                    new_row.append('\u2192') # right
                elif (digit==2):
                    new_row.append('\u2190') # left
                elif (digit==3):
                    new_row.append('\u2193') # down
                elif (digit=='G'):
                    new_row.append('G') # goal state
                elif (digit=='S'):
                    new_row.append('S') # goal state
                elif (digit==-1):
                    new_row.append('+') # All four directions
                else:
                    new_row.append('x') # unknown

        new_table.append(new_row)

    table = ax.table(cellText=new_table, loc='center',cellLoc='center')

    table.scale(1,2)
```

```python
    for i in range(height):
        new_row = []

        for j in range(width):
            if new_table[i][j] == '':
                table[i, j].set_facecolor('black')
            else:
                table[i, j].
↪set_facecolor(get_color(heatmap[i][j],min_val,max_val))

    for goal_state in goal_states:
        table[(goal_state[0], goal_state[1])].set_facecolor("limegreen")
    table[(start_state[0], start_state[1])].set_facecolor("yellow")
    ax.axis('off')
    table.set_fontsize(16)

    for i in range(len(x_coords)):
        table[(x_coords[i], y_coords[i])].get_text().set_color('red')
    plt.show()


# this function takes 3D Q table as an input
# and outputs optimal trajectory table (policy table)
# and corresponding excpected reward values of different cells (heatmap)
def get_policy_table(q_hat_3D, start_state, goal_states):
    policy_table = []
    heatmap = []

    for i in range(q_hat_3D.shape[0]):
        row = []
        heatmap_row = []
        for j in range(q_hat_3D.shape[1]):

            heatmap_row.append(np.max(q_hat_3D[i,j,:]))

            for goal_state in goal_states:
                if (goal_state[0]==i) and (goal_state[1]==j):
                    row.append('G')

            if (start_state[0]==i) and (start_state[1]==j):
                row.append('S')
            else:
                if np.max(q_hat_3D[i,j,:]) == 0:
                    row.append(-1) # All zeros
                else:
                    row.append(np.argmax(q_hat_3D[i,j,:]))
        policy_table.append(row)
```

```
        heatmap.append(heatmap_row)

    return policy_table, heatmap

def plot_policy_from_q(q_hat, env):
    q_hat_3D = np.reshape(q_hat, (env.m_size, env.m_size, env.num_actions))
    max_val = q_hat_3D.max()
    min_val = q_hat_3D.min()
    start_state = env.get_coords_from_state(env._get_start_state)
    goal_states = env._get_goal_state
    goal_states = [env.get_coords_from_state(goal_state) for goal_state in␣
 ↪goal_states]
    policy_table, heatmap = get_policy_table(q_hat_3D, start_state, goal_states)
    x,y = get_next_cell(start_state[0],start_state[1],heatmap,policy_table)
    x_coords, y_coords = get_path(x,y,policy_table)
    plot_table(env, policy_table, heatmap, goal_states,␣
 ↪start_state,max_val,min_val, x_coords, y_coords)

    return
```

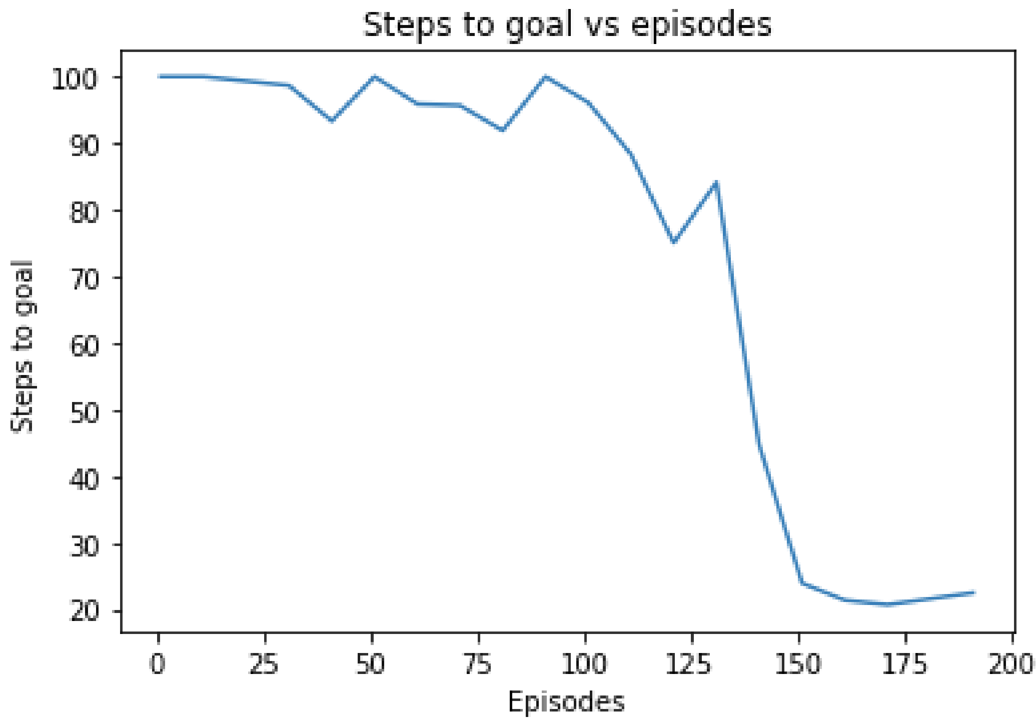
2.3 Write Up:

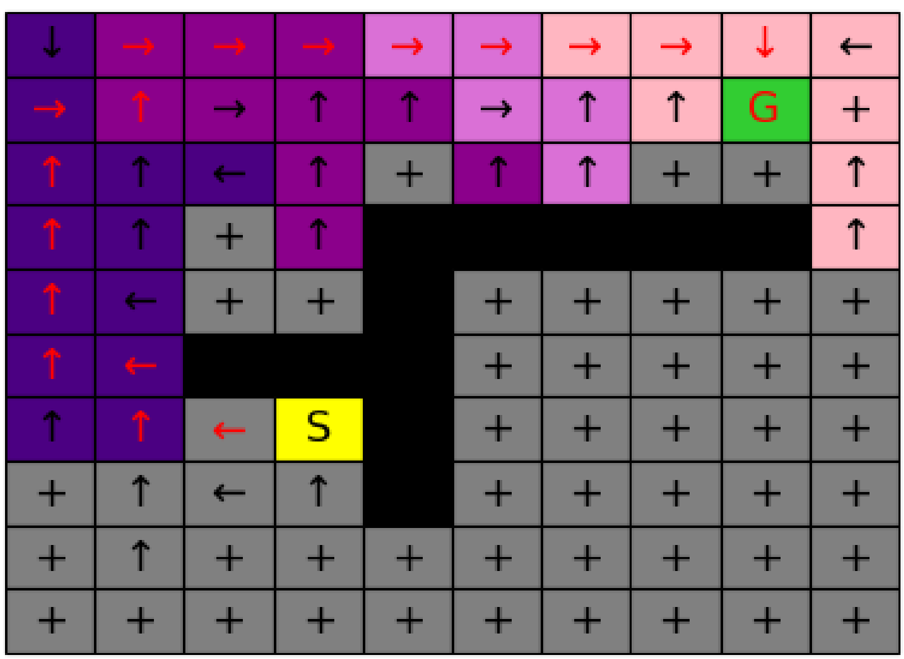
For the section 2.1, the question let us to implement a Q learning algorithm, which is a reinforcement learning algorithm that gives rewards for every decision an agent makes.

For the section 2.2.1, we basically set everything to default and test the correctness of the algorithm. We tried diffrent start location and goal location and get steps vs iterations plots and policy tables like below: (a) -> (b)

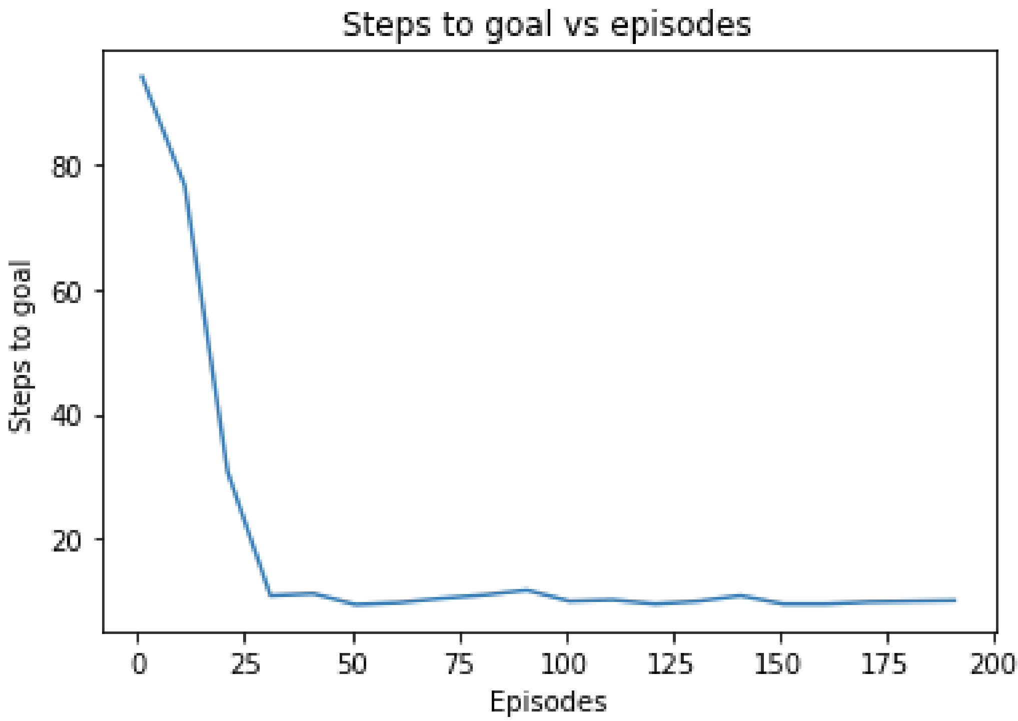
[891]:

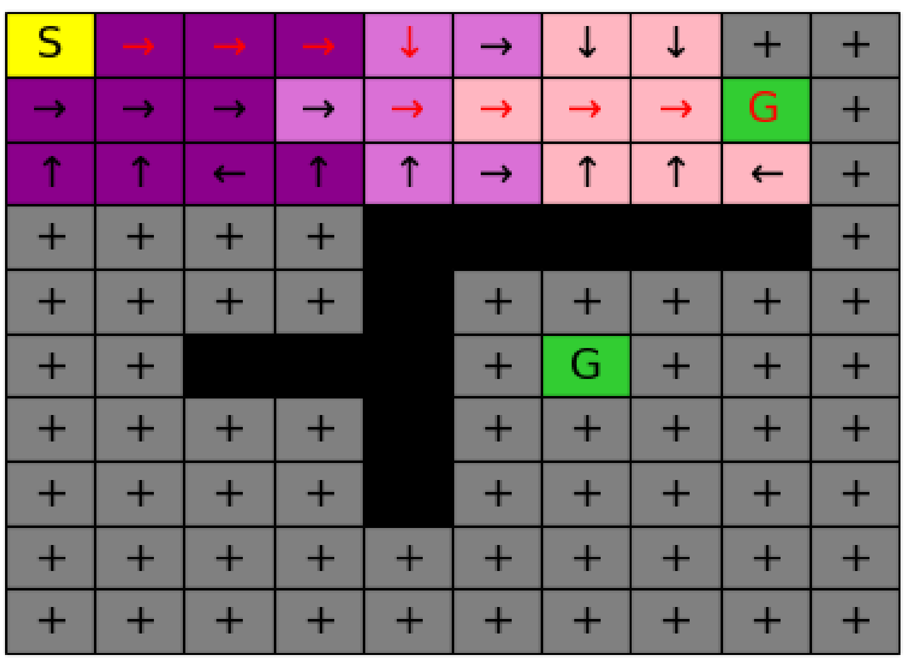
```

```





<Figure size 720x720 with 0 Axes>



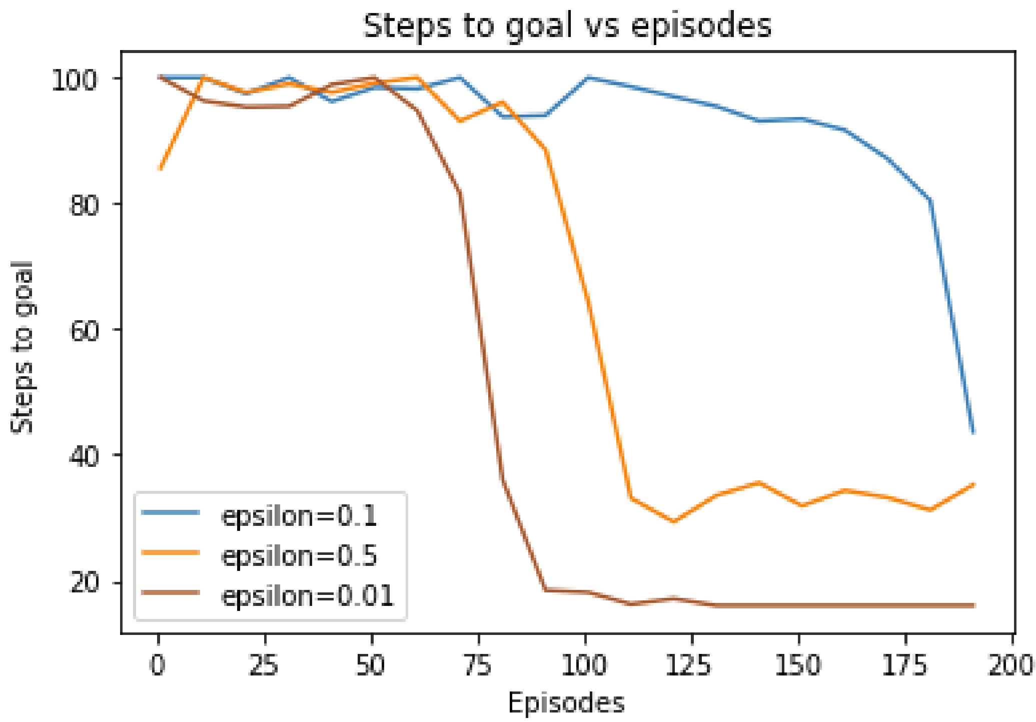


`<Figure size 720x720 with 0 Axes>`

We implemented the qlearn and epsilon_greedy altorithm for this question. Our algorithm find a path from start location to the goal location, and for each step, the policy is to choose the move with maximum reward.

For the section 2.2.2, we have three subproblems. Problem a requires us to change the value of epsilon and to see the effect of different epsilon value to the convergency rate. Below graph shows the relation:

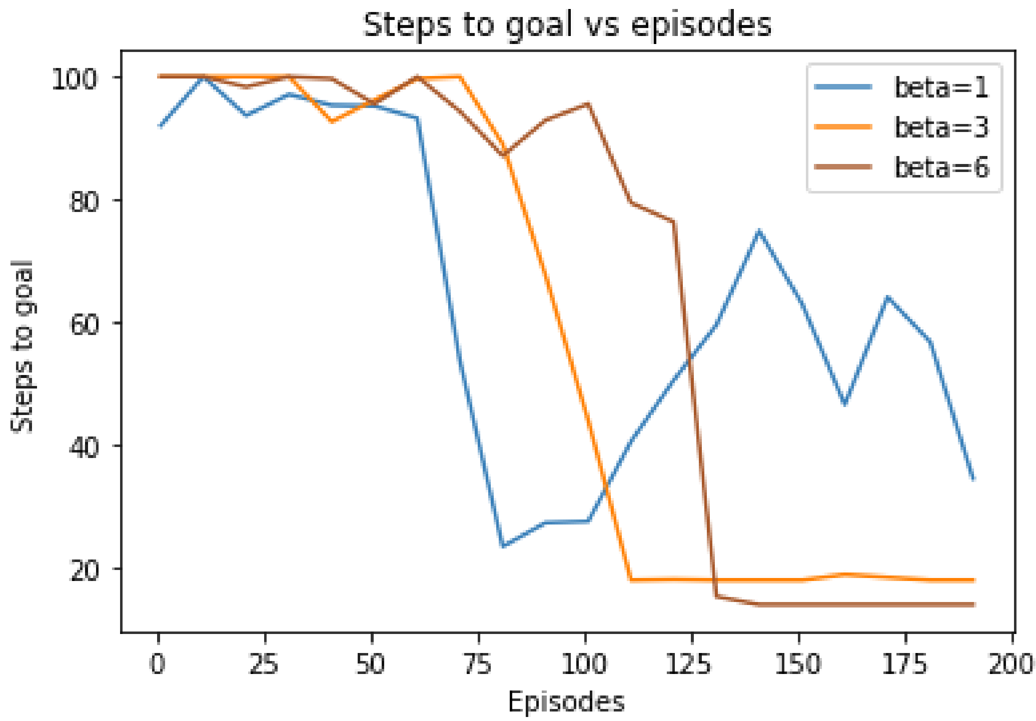
[893]:



The graph shows generally, it take shorter to get to the goal location with a smaller epsilon vlaue, since it is more determinstic (the chance of step a random move is less than others) However, it is also possible that somehow the agent get a better move with random choice so that it converges faster.

For the problem b, we were required to use softmax policy instead of epsilon_greedy algorithm. We examined how different fixed beta value can affect the steps to goal vs episoides curves shape.

[897]:



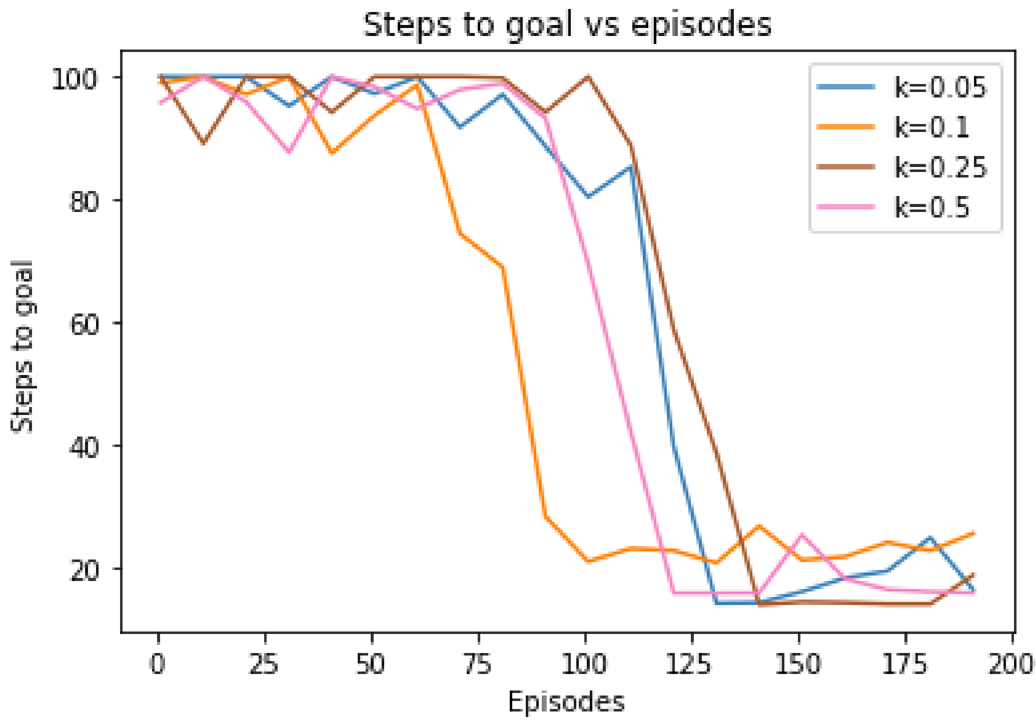
From above image, we found that it is always faster to get closer to the goal location with smaller beta value. However, if the beta value is over small, the steps to goal curve may rebound upward at the end.

For problem c, instead of fixing beta value to a constant c, we used variable beta value with the equation

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

Below graph shows the relationship between k value (how fast does beta changes) and steps to goals vs episodes curve.

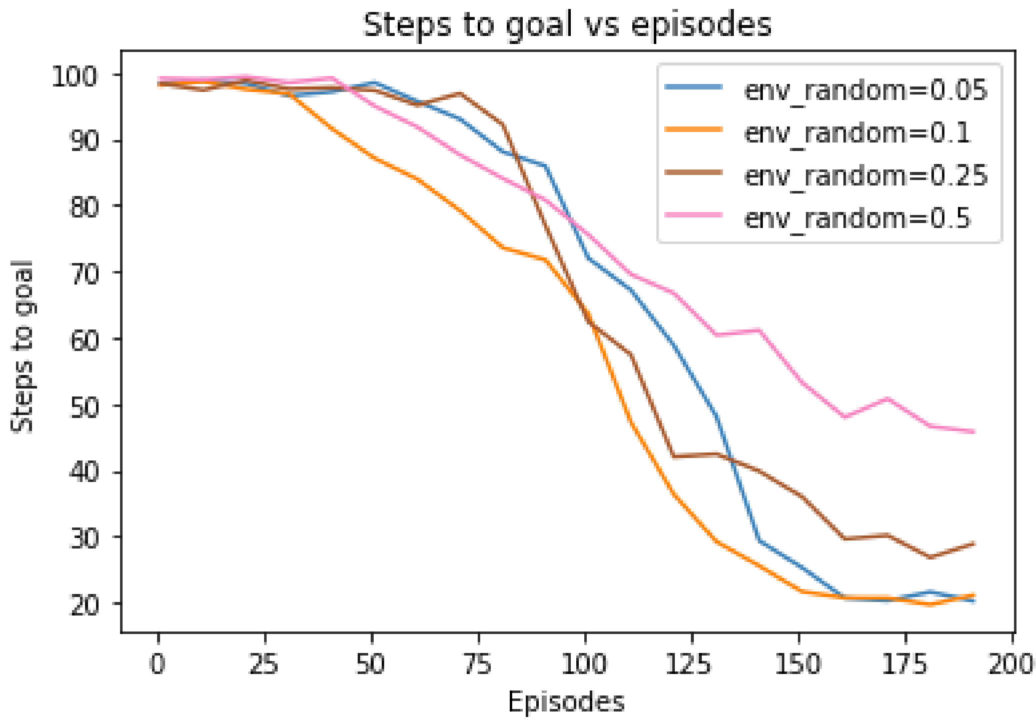
[900]:



The graph suggests that with variable beta value, every curve can eventuall get closer to the goal location, regardless the fact that it takes longer for them to converge. This is due to that the initial beta value was set to 1.0, which is small.

For the section 3, we made the environment more stochastic by sometimes force the agent to move towards random direction.

[901]:



By the graph above, we observed that with higher Prand value, the more unstable the curve is, which correspond to the more randomness the agent's moves are. Therefore, it is faster for lower Prand value to get to the goal location, since in those situations, larger percent of time the agent is moving towards correct computed location, rather than just move randomly.

4 3. Did you complete the course evaluation?

[864]:
```
# Answer: yes / no
```

[865]:
```
# yes
```