STA2104_HW4_vFINAL

April 5, 2021

Luis Alvaro Correia - *Student Id:***1006508566**

1 1. Unsupervised Learning

[1]:
```
%matplotlib inline
import scipy
import numpy as np
import pandas as pd
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

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

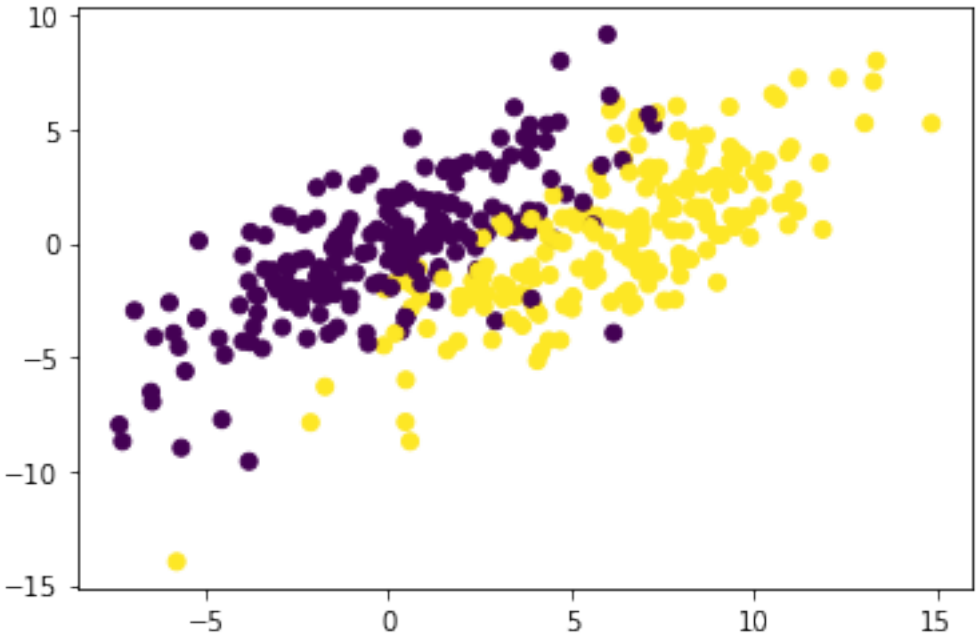
Here, $N = 400$. Since you generated the data, you already know which sample comes from which class. Run the cell in the IPython notebook to generate the data.

[2]:
```
# TODO: Run this cell to generate the data
num_samples = 400
cov = np.array([[1., .7], [.7, 1.]]) * 10
mean_1 = [.1, .1]
mean_2 = [6., .1]

np.random.seed(963)  # Included to enable reproducibility

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):

[3]:
```
# TODO: Make a scatterplot for the data points showing the true cluster␣
 ↪assignments of each point
plt.scatter(data[:,0],data[:,1],c=labels)
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.

[4]:
```
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
```

[5]:
```
# TODO: K-Means Assignment Step
def km_assignment_step(data, Mu):
    """ Compute K-Means assignment step

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

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

    # Fill this in:
    N, D = data.shape # Number of datapoints and dimension of datapoint
    K = Mu.shape[1] # number of clusters
    r = np.zeros([N,K])

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

    arg_min = r.argmin(axis=1) # argmax/argmin along dimension 1
    R_new = np.eye(K)[arg_min] # Get Cluster Assignment

    return R_new
```

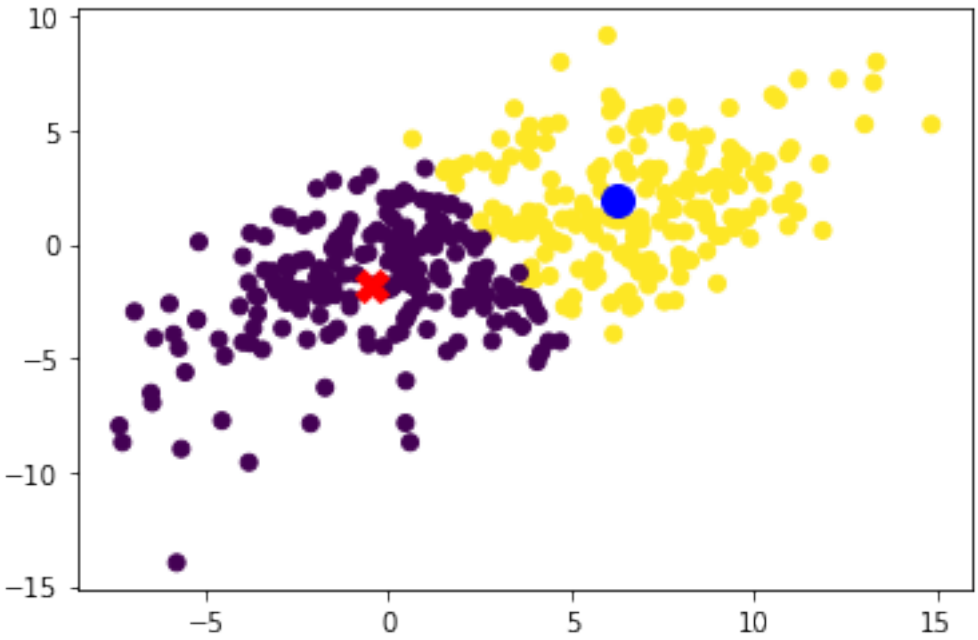
[6]:
```
# TODO: K-means Refitting Step
def km_refitting_step(data, R, Mu, plotstatus):  # included parameter for␣
↪debugging
    """ Compute K-Means refitting step.

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

    Returns:
        Mu_new: a DxK matrix for the new cluster means locations
    """
    N, D = data.shape # Number of datapoints and dimension of datapoint
    K = Mu.shape[1]  # number of clusters
    Mu_new = (R.T.dot(data)/np.sum(R,axis=0)).T

    if (plotstatus):
        plt.scatter(data[:,0],data[:,1],c=R.argmax(axis=1))
```

```
        plt.plot(Mu_new[0,0],Mu_new[1,0],marker='X',color='r', markersize=12)
        plt.plot(Mu_new[0,1],Mu_new[1,1],marker='o', color='b', markersize=12)
        plt.show()
        print(Mu_new)
        print(np.sum(R,axis=0))
    return Mu_new
```

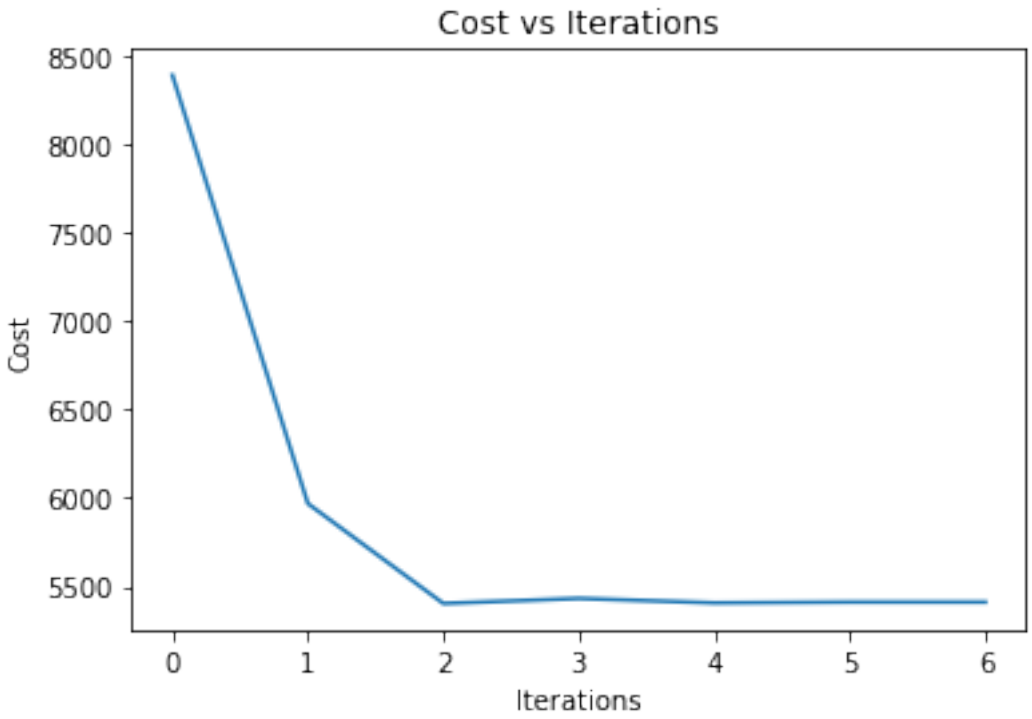
[7]:
```
# TODO: Run this cell to call the K-means algorithm
N, D = data.shape
K = 2
max_iter = 100
class_init = np.random.binomial(1., .5, size=N)
R = np.vstack([class_init, 1 - class_init]).T

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

# Changed to stop after convergence
it = 0

tolerance = 1e-5                  # Tolerance to check convergence
Converged = False                 # Set Control Variable
cost_history = np.zeros(max_iter)  # Cost History of Convergence

while (not Converged and (it < max_iter)):
    R = km_assignment_step(data, Mu)
    Mu = km_refitting_step(data, R, Mu, False) # DEBUG -> (it % 10 == 0))

    cost_history[it] = cost(data, R, Mu)
    if (it > 0):
        Converged = (abs(cost_history[it]-cost_history[it-1])<=tolerance)

    # print(it, cost_history[it])

    # Increment iteration
    it += 1

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

print("\n>>> %s after %d iterations with Cost=%.5f" %
        (("Convergence" if Converged else "Not converged"), it, cost_history[it␣
 ↪if Converged else (it-1)]))

print("\n\nK-Means classified %d points as Class 1 and %d points as Class 2\n"␣
 ↪% (len(class_1[0]), len(class_2[0])))
```

```
>>> Convergence after 7 iterations with Cost=0.00000

K-Means classified 207 points as Class 1 and 193 points as Class 2
```

[8]:
```
# TODO: Make a scatterplot for the data points showing the K-Means cluster␣
 ↪assignments of each point
plt.scatter(data[:,0],data[:,1],c=R.argmax(axis=1))
plt.plot(Mu[0,0],Mu[1,0],marker='X',color='r', markersize=12)
plt.plot(Mu[0,1],Mu[1,1],marker='o', color='b', markersize=12)
plt.show()
```



[9]:
```
# Also plot the Cost vs. iterations
plt.figure()
plt.title("Cost vs Iterations")
plt.ylabel("Cost")
plt.xlabel("Iterations")
plt.plot(range(it), cost_history[:it])
plt.show()
```



[10]:
```
# CLASSIFICATION ERROR - Included as requested by Q1(b)
accuracy = (1-np.count_nonzero(R.argmax(axis=1)-labels)/num_samples)*100.0
print("\nK-Means accuracy is %5.2f%%" % accuracy)

print("\n\n------ Listing of Classification Errors ------\n")

Err = np.nonzero(R.argmax(axis=1)-labels)[0]
print("\n>>> Total No. of Errors: %6d\n\n" % Err.shape[0])
dErr = {'Item No.':Err,
        'Label':labels[Err].astype(int),
        'Classif.':R.argmax(axis=1)[Err]}
dfErr = pd.DataFrame(dErr)

print(dfErr.to_string(header=True, index=False))
```

```
K-Means accuracy is 75.75%


------ Listing of Classification Errors ------


>>> Total No. of Errors:     97
```

Item No.	Label	Classif.
1	0	1
6	0	1
8	0	1
10	0	1
17	1	0
18	0	1
22	1	0
23	0	1
26	1	0
34	1	0
37	0	1
43	0	1
44	0	1
50	1	0
53	0	1
62	1	0
70	1	0
80	0	1
88	1	0
92	1	0
100	1	0
110	0	1
112	1	0
115	1	0
117	0	1
120	1	0
122	0	1
125	1	0
129	0	1
130	1	0
132	0	1
142	1	0
147	0	1
148	0	1
154	1	0
155	1	0
158	1	0
159	0	1
163	1	0
169	1	0
176	1	0
178	0	1
181	0	1
183	1	0
184	0	1
187	0	1

209	1	0
215	1	0
216	1	0
217	0	1
218	0	1
219	1	0
224	0	1
229	0	1
232	1	0
238	1	0
239	0	1
245	0	1
246	1	0
249	1	0
254	1	0
265	1	0
271	0	1
272	1	0
273	0	1
275	0	1
277	0	1
278	1	0
289	0	1
290	1	0
310	0	1
312	0	1
315	1	0
320	0	1
325	1	0
327	1	0
333	1	0
334	1	0
339	0	1
340	0	1
343	1	0
344	1	0
351	0	1
356	1	0
357	0	1
358	1	0
366	1	0
381	1	0
385	0	1
389	1	0
392	1	0
393	1	0
394	0	1
395	0	1

```
396       1          0
397       0          1
399       1          0
```

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 the same initialization as in Q2.1 for the means, and with $\hat{\Sigma}_1 = \hat{\Sigma}_2 = I$, and $\hat{\pi}_1 = \hat{\pi}_2$ for the covariances.

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.

[11]:
```
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))
```

[12]:
```
def log_likelihood(data, Mu, Sigma, Pi):
    """ Compute log likelihood on the data given the Gaussian Mixture␣
↪Parameters.

    Args:
        data: a NxD matrix for the data points
        Mu: a DxK matrix for the means of the K Gaussian Mixtures
        Sigma: a list of size K with each element being DxD covariance matrix
        Pi: a vector of size K for the mixing coefficients

    Returns:
        L: a scalar denoting the log likelihood of the data given the Gaussian␣
↪Mixture
    """
    # Fill this in:
    N, D = data.shape  # Number of datapoints and dimension of datapoint
    K = Mu.shape[1]    # number of mixtures
    L, T = 0., 0.
    for n in range(N):
        for k in range(K):
            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
```

[13]:
```
# TODO: Gaussian Mixture Expectation Step
def gm_e_step(data, Mu, Sigma, Pi):
    """ Gaussian Mixture Expectation Step.

    Args:
        data: a NxD matrix for the data points
```

```
        Mu: a DxK matrix for the means of the K Gaussian Mixtures
        Sigma: a list of size K with each element being DxD covariance matrix
        Pi: a vector of size K for the mixing coefficients

    Returns:
        Gamma: a NxK matrix of responsibilities
    """
    # Fill this in:
    N, D = data.shape  # Number of datapoints and dimension of datapoint
    K = Mu.shape[1]   # number of mixtures
    Gamma = np.zeros([N,K]) # zeros of shape (N,K), matrix of responsibilities
    for n in range(N):
        for k in range(K):
            Dens = normal_density(data[n,:],Mu[:,k], Sigma[k])
            Gamma[n, k] = Pi[k]*Dens
        Gamma[n, :] /= np.sum(Gamma[n,:]) # Normalize by sum across second␣
↪dimension (mixtures)
    return Gamma
```

[14]:
```
# TODO: Gaussian Mixture Maximization Step
def gm_m_step(data, Gamma, plotstatus):   # Included for debugging
    """ Gaussian Mixture Maximization Step.

    Args:
        data: a NxD matrix for the data points
        Gamma: a NxK matrix of responsibilities

    Returns:
        Mu: a DxK matrix for the means of the K Gaussian Mixtures
        Sigma: a list of size K with each element being DxD covariance matrix
        Pi: a vector of size K for the mixing coefficients
    """
    # Fill this in:
    N, D = data.shape  # Number of datapoints and dimension of datapoint
    K = Gamma.shape[1]   # number of mixtures
    Nk = np.sum(Gamma, axis=0) # Sum along first axis
    Mu = (1/Nk)*(Gamma.T.dot(data)).T
    Sigma = [np.eye(2), np.eye(2)]

    for k in range(K):
        weightedSum = np.zeros([D,D])
        diff = (data-np.array([Mu[:, k], ]* N)).T
        weightedSum = (Gamma[:,k]*diff).dot(diff.T)
        Sigma[k] = weightedSum/Nk[k]

    Pi = Nk/N
```

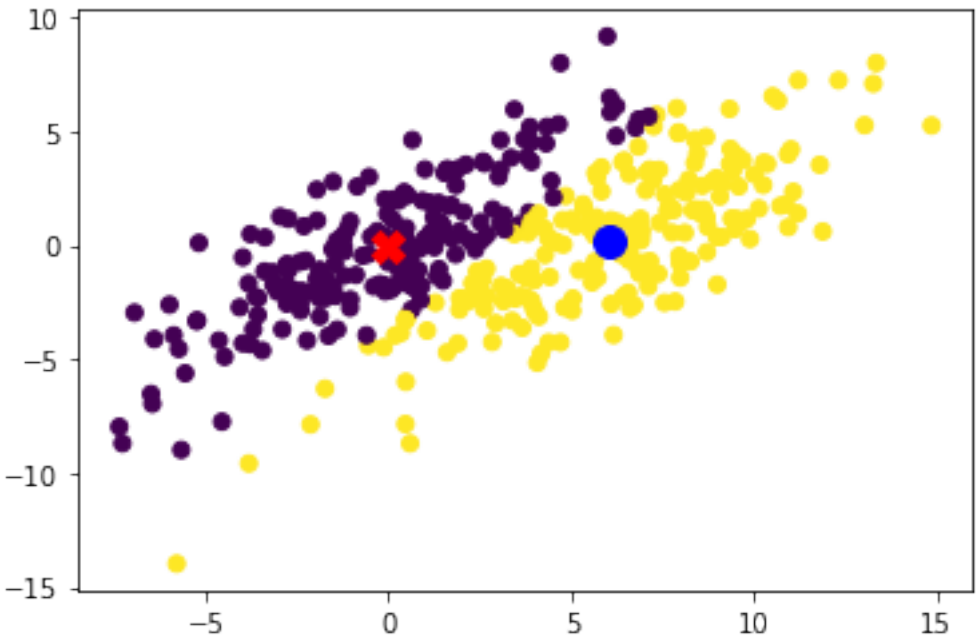
```
    if (plotstatus):
        plt.scatter(data[:,0],data[:,1],c=Gamma.argmax(axis=1))
        plt.plot(Mu[0,0],Mu[1,0],marker='X',color='r', markersize=12)
        plt.plot(Mu[0,1],Mu[1,1],marker='o', color='b', markersize=12)
        plt.show()
        print(Mu)
        print(np.sum(Gamma,axis=0))

    return Mu, Sigma, Pi
```

[15]:
```
# TODO: Run this cell to call the Gaussian Mixture EM algorithm
N, D = data.shape
K = 2
Mu = np.zeros([D, K])
Mu[:, 1] = 1.
Sigma = [np.eye(2), np.eye(2)]
Pi = np.ones(K) / K
Gamma = np.zeros([N, K]) # Gamma is the matrix of responsibilities

max_iter  = 200

# Changed to stop after convergence
it = 0

tolerance = 1e-5                      # Tolerance to check convergence
Converged = False                     # Set Control Variable
loglik_history = np.zeros(max_iter)   # Log-Likelihood History of Convergence

while (not Converged and (it < max_iter)):
    Gamma = gm_e_step(data, Mu, Sigma, Pi)
    Mu, Sigma, Pi = gm_m_step(data, Gamma, False) # DEBUG -> (it % 10 == 0))

    loglik_history[it] = log_likelihood(data, Mu, Sigma, Pi)
    if (it > 0):
        Converged = (abs(loglik_history[it]-loglik_history[it-1])<=tolerance)

    # print(it, loglik_history[it])

    # Increment iteration
    it += 1

print("\n>>> %s after %d iterations with LogLike=%.5f" %
        (("Convergence" if Converged else "Not converged"), it,␣
 ↪loglik_history[it if Converged else (it-1)]))

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

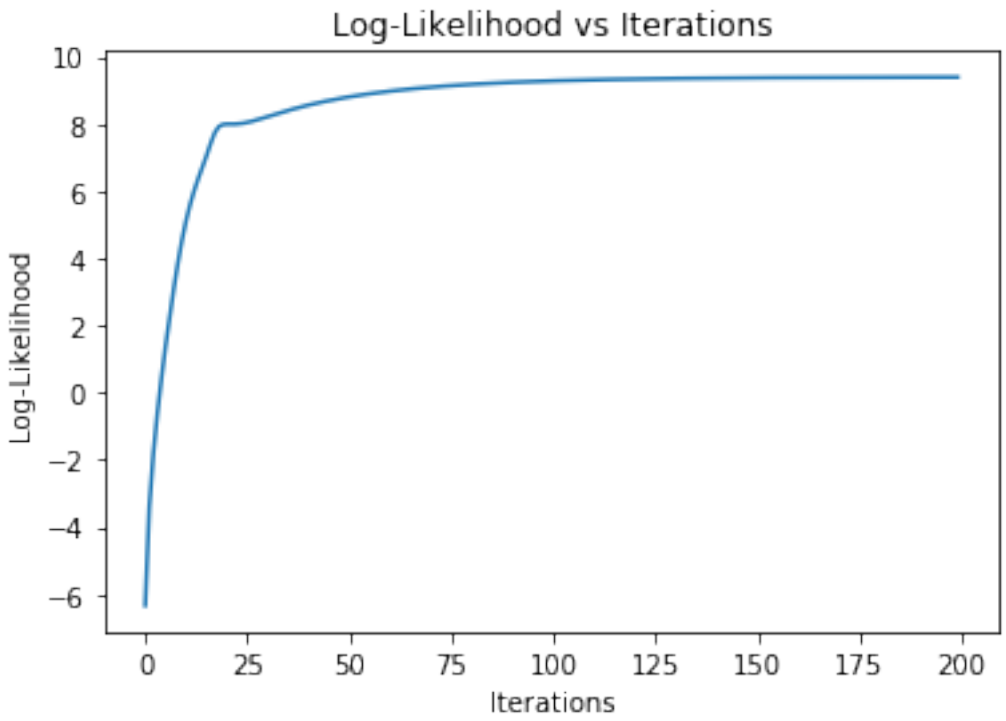
```
print("\n\nE.M. classified %d points as Class 1 and %d points as Class 2\n" %␣
 ↪(len(class_1[0]), len(class_2[0])))
```

```
>>> Not converged after 200 iterations with LogLike=9.38685


E.M. classified 199 points as Class 1 and 201 points as Class 2
```

[16]:
```
# TODO: Make a scatterplot for the data points showing the Gaussian Mixture␣
 ↪cluster assignments of each point
plt.scatter(data[:,0],data[:,1],c=Gamma.argmax(axis=1))
plt.plot(Mu[0,0],Mu[1,0],marker='X',color='r', markersize=12)
plt.plot(Mu[0,1],Mu[1,1],marker='o', color='b', markersize=12)
plt.show()
```



[17]:
```
# Also plot the log-likelihood vs. iterations
plt.figure()
plt.title("Log-Likelihood vs Iterations")
plt.ylabel("Log-Likelihood")
plt.xlabel("Iterations")
plt.plot(range(it), loglik_history[:it])
plt.show()
```



[18]:
```
accuracy = (1-np.count_nonzero(Gamma.argmax(axis=1)-labels)/num_samples)*100.0
print("\nE.M. accuracy is %5.2f%%" % accuracy)

print("\n\n------ Listing of Classification Errors ------\n")

Err = np.nonzero(Gamma.argmax(axis=1)-labels)[0]
print("\n>>> Total No. of Errors: %6d\n\n" % Err.shape[0])
dErr = {'Item No.':Err,
        'Label':labels[Err].astype(int),
        'Classif.':Gamma.argmax(axis=1)[Err]}
dfErr = pd.DataFrame(dErr)

print(dfErr.to_string(header=True, index=False))
```

```
E.M. accuracy is 89.75%


------ Listing of Classification Errors ------


>>> Total No. of Errors:     41
```

Item No.	Label	Classif.
10	0	1
17	1	0
26	1	0
34	1	0
39	1	0
44	0	1
57	0	1
62	1	0
69	1	0
70	1	0
89	0	1
91	0	1
96	0	1
100	1	0
113	1	0
130	1	0
132	0	1
169	1	0
170	1	0
183	1	0
204	1	0
207	1	0
215	1	0
218	0	1
224	0	1
229	0	1
231	0	1
234	0	1
237	0	1
271	0	1
274	0	1
277	0	1
278	1	0
288	1	0
294	1	0
312	0	1
320	0	1
339	0	1
373	0	1
390	0	1
393	1	0

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

Item (a) - Comparing the Performance of K-Means and E.M. Algorithm in terms of cluster allocation (default configuration)

K-Means Algorithm classified data-points in (C1/C2) at a proportion of $(207/193)$. The classification did not follow the real allocation because K-Mean's two-step *assignment/refitting* starts from the estimated initial means, classifies the points aligned with the distance from points to that means and iterates until stability. Sometimes this stability doesn't occurs according with the original labels and this was we observed in present run. This generated an accuracy of 75.75% with 97 misclassifications.

On the other hand, **E.M. Algorithm** started from the initial estimates of $\hat{\mu_1}$, $\hat{\mu_2}$ and $\hat{\Sigma}$ and assumes a Gaussian distribution of each class around its means. By iterating a cycle of estimation of means, variances, gaussian mixtures given responsibilities (and vice-versa) EM adjusts better to the original distributions of data when they are indeed gaussians. By doing this, the algorithm obtained a better cluster allocation, with an almost perfectly paired proportion of $(199/201)$, accuracy of 89.75% and 41 misclassifications.

Item (b) - Comparing the Performance of K-Means and E.M. Algorithm in terms of convergence (default configuration)

K-Means Algorithm has converged very fast, after 07 iterations while **E.M. Algorithm** reached 200 maximum iterations limit with no convergence achieved. For this experiment we have set a tolerance level $\epsilon = 10^{-5}$ to evaluate the variation for *cost* (for K-means) and *log-likelihood* (for EM) convergence.

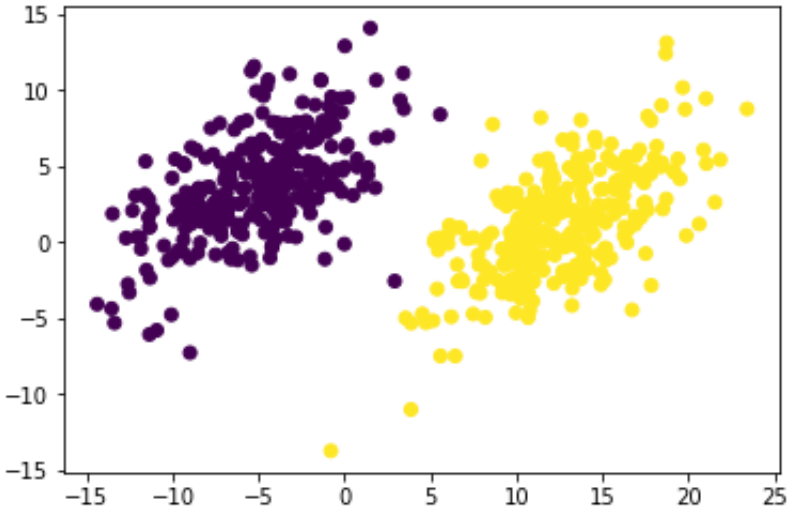
Item (c) - Experiment 5 different data realizations and summarize findings.

For this question, we generated data from 05 different models with different sample sizes, as follows:

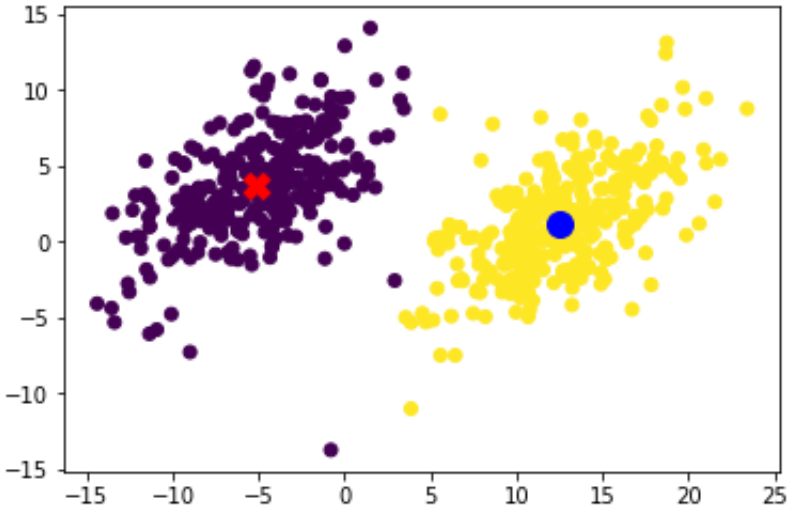
Configuration No.1 - `num_samples`$= 600$

$$\Sigma = \begin{bmatrix} 14.0 & 7.0 \\ 7.0 & 12.3 \end{bmatrix} \mu_1 = \begin{bmatrix} -5.0 \\ 4.0 \end{bmatrix} \mu_2 = \begin{bmatrix} 12.6 \\ 1.2 \end{bmatrix} \tag{1}$$

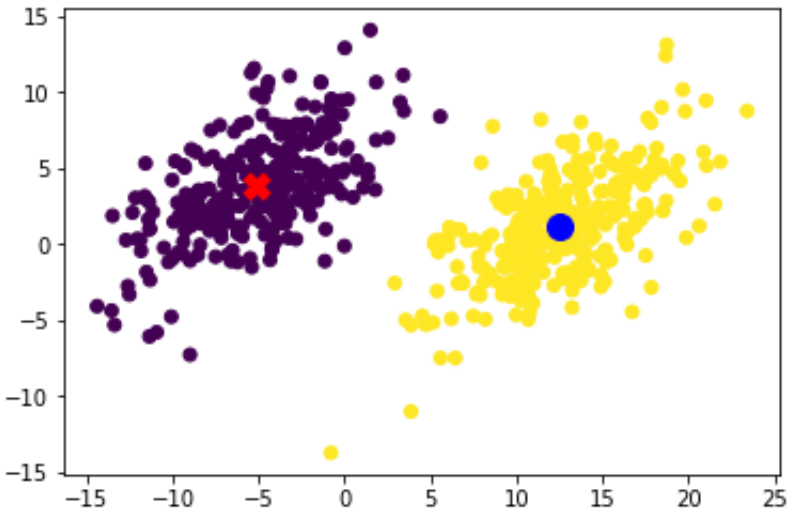
Comment:- In this sample, the populations were linearly separable which leaded to *high accuracy* for both algorithms. We achieved accuracy of 99.67% in 5 iterations for K-Means, and 99.83% in 11 iterations for EM. Very fast convergence with high accuracy.



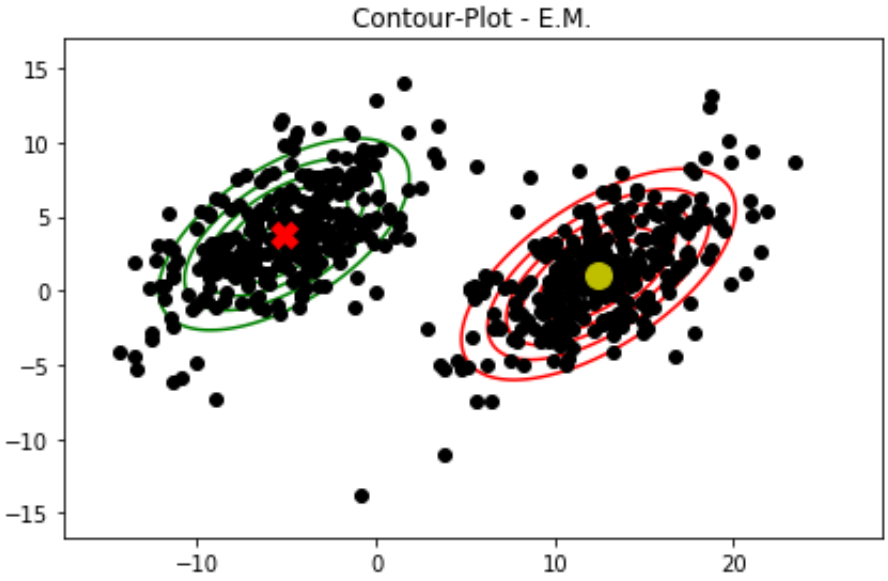
New Samples - Config 1



K-Means Algorithm - Config 1



EM Algorithm - Config 1



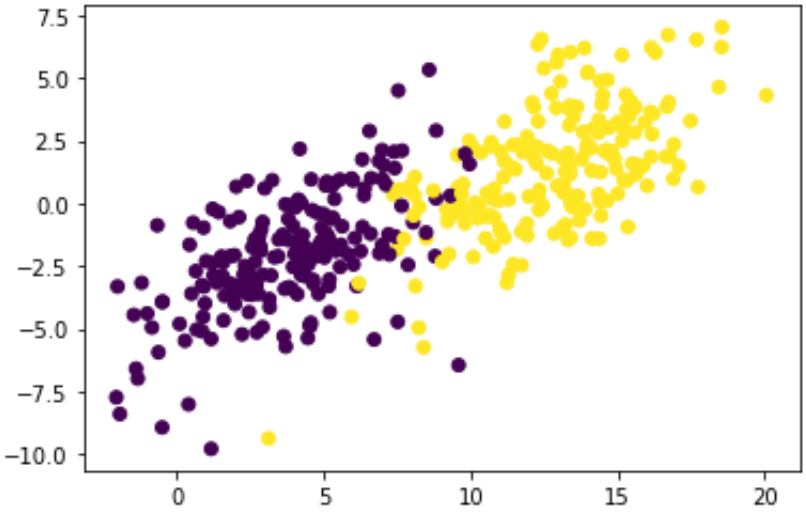
EM Contour-Plot - Config 1

Configuration No.2 - `num_samples`$= 400$

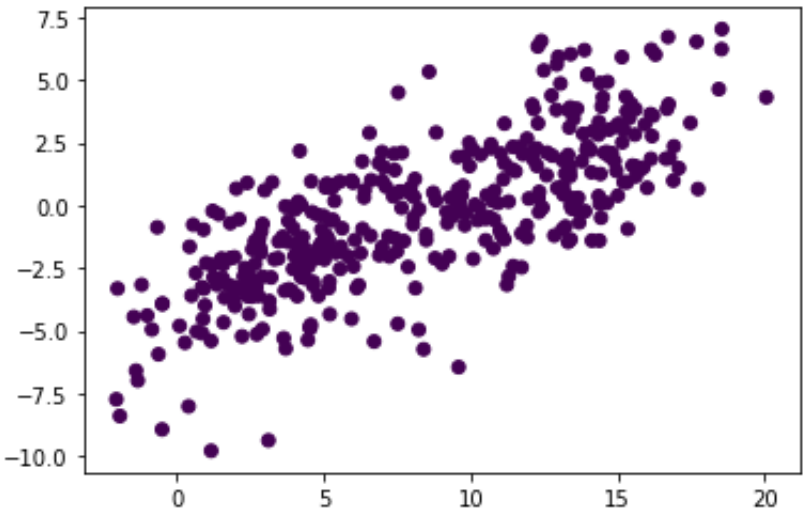
$$\Sigma = \begin{bmatrix} 7.0 & 3.5 \\ 3.5 & 6.15 \end{bmatrix} \mu_1 = \begin{bmatrix} 4.0 \\ -1.8 \end{bmatrix} \mu_2 = \begin{bmatrix} 12.6 \\ 1.2 \end{bmatrix} \tag{2}$$

Comment:- In this configuration, K-Means failed to converge, while EM algorithm converged very well in 106 iterations and 94.0% of accuracy. The reason is due to the populations which are right-shifted from the origin, then initial points estimates $\hat{\mu_1}$ and $\hat{\mu_2}$ are bad for K-Means iteration,

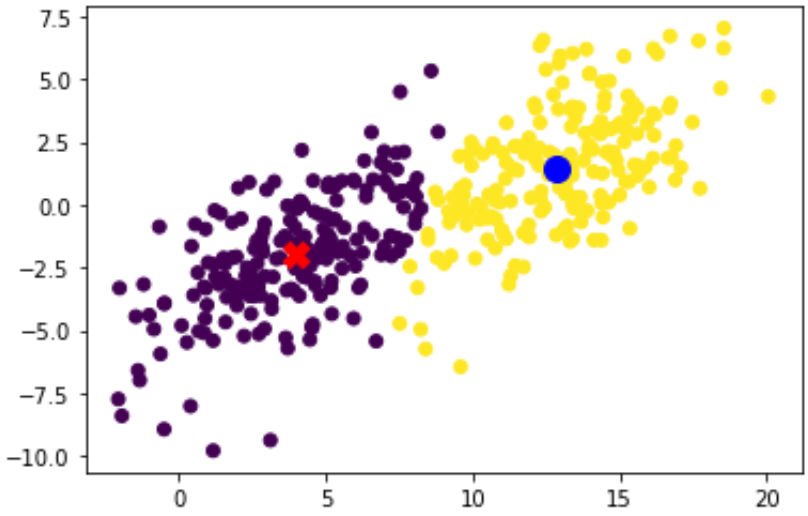
but it doesn't affects EM which continuously generates new estimates for mean and covariance for each class and can adjust itself even with bad initial choices.



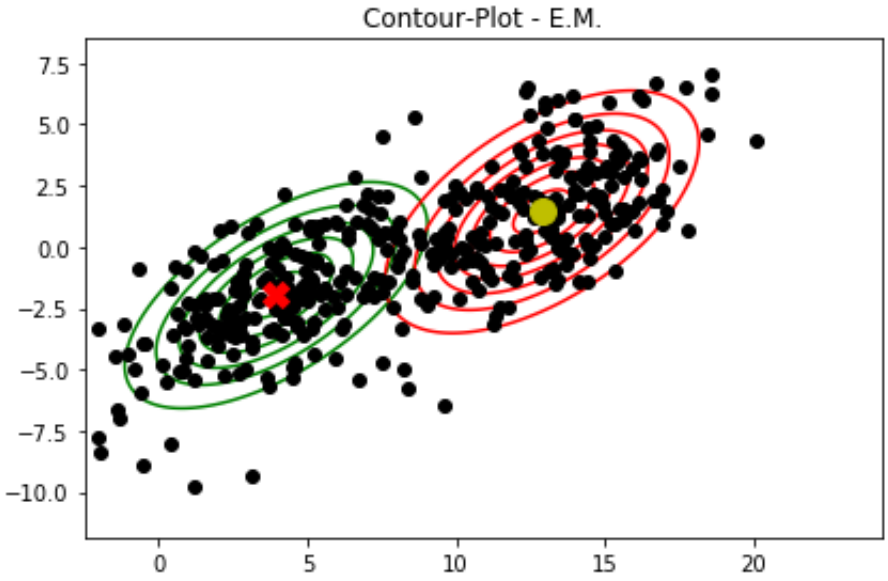
New Samples - Config 2



K-Means Algorithm - Config 2



EM Algorithm - Config 2



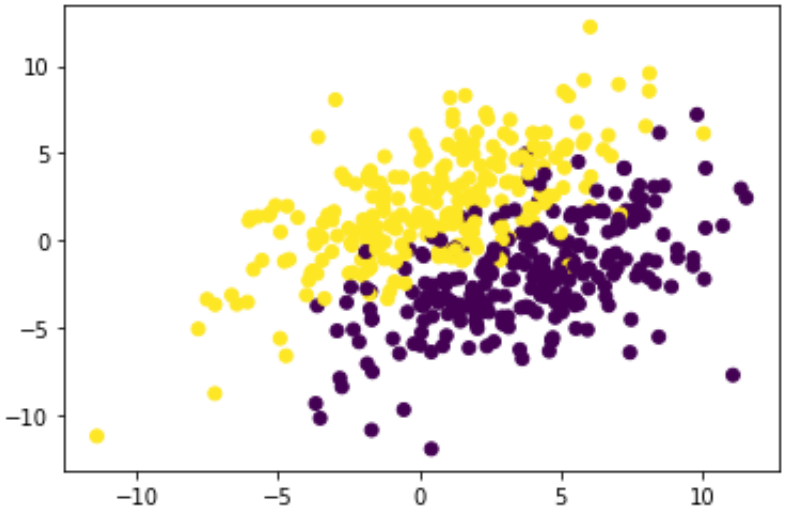
EM Contour-Plot - Config 2

Configuration No.3 - `num_samples`$= 500$

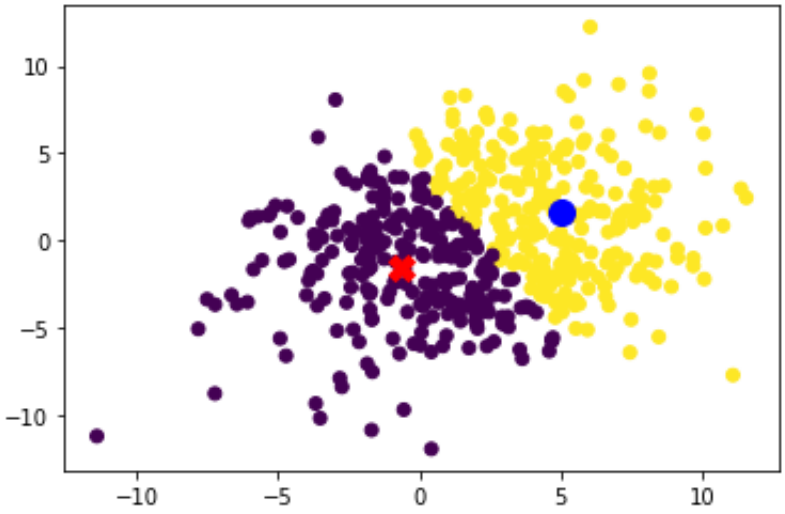
$$\Sigma = \begin{bmatrix} 11.2 & 5.6 \\ 5.6 & 9.84 \end{bmatrix} \mu_1 = \begin{bmatrix} 4.0 \\ -1.8 \end{bmatrix} \mu_2 = \begin{bmatrix} 0.6 \\ 2.2 \end{bmatrix} \tag{3}$$

Comment:- Using this configuration we get a curious behaviour from both algorithms. K-Means tried to separate the populations in a different way they were generated. As the algorithm looks for differences between points and given means at the start, it doesn't care about which axis

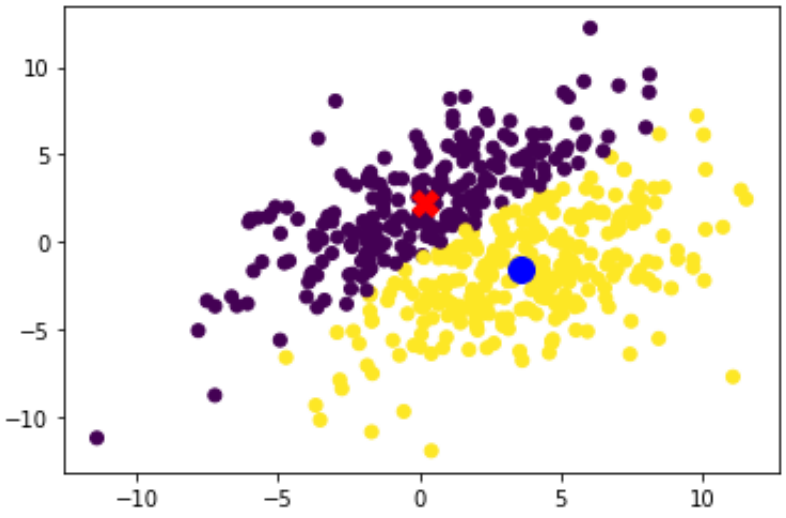
were selected as starting point and try using them to separate the data. That's why the populations were indeed separated with a low accuracy of 43.6% and consumed all 100 iterations available. On the other hand, EM algorithm has separated correctly but classified the populations with inverted labels. The initial points estimates $\hat{\mu_1}$ and $\hat{\mu_2}$ had the reverse effect on how the algorithm estimated class means and covariances. This fact made classes wrongly classified and the process consumed 200 iterations to get an accuracy of only 12.8%. If we just invert the means of each class, K-Means increases accuracy to 53.2% and EM's up to 90.4% which confirms how bad we set our initial estimates.



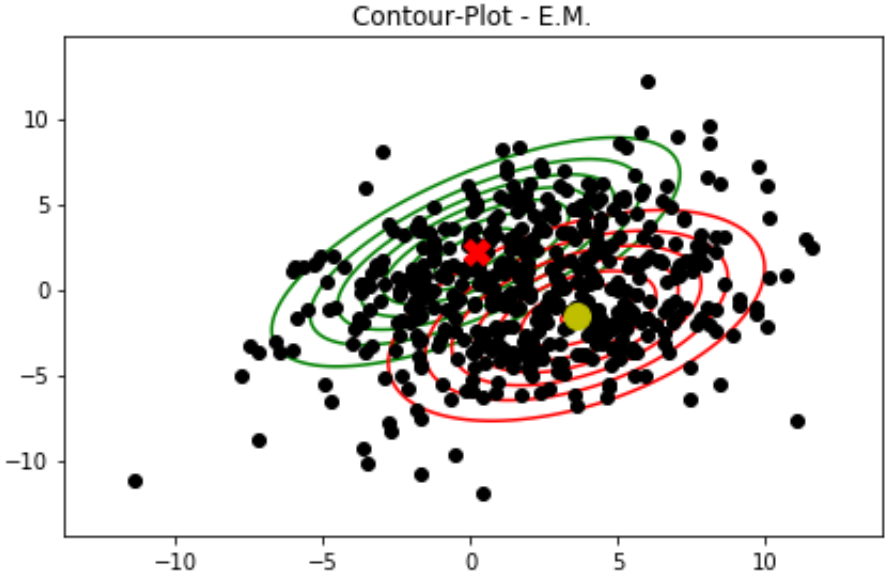
New Samples - Config 3



K-Means Algorithm - Config 3



EM Algorithm - Config 3



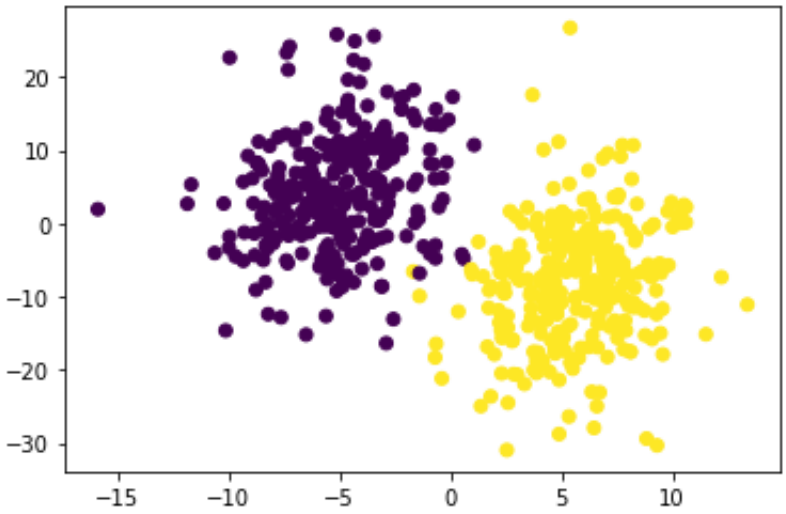
EM Contour-Plot - Config 3

Configuration No.4 - `num_samples`$= 600$

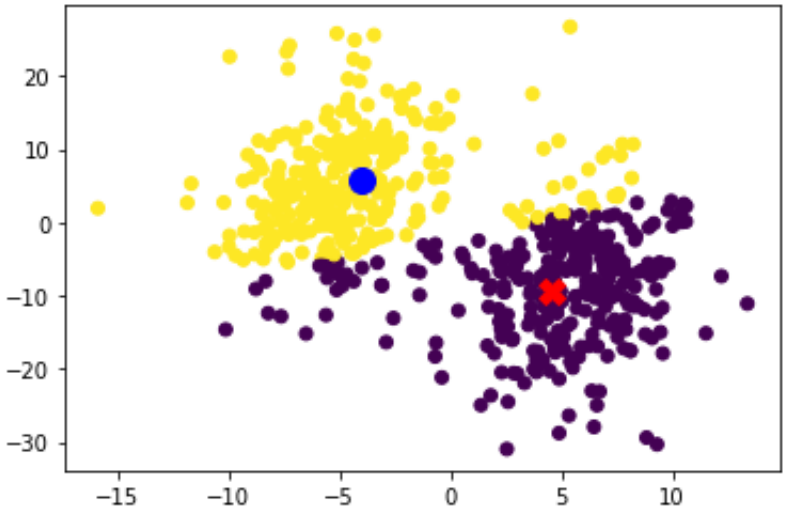
$$\Sigma = \begin{bmatrix} 7.7 & 3.5 \\ 3.5 & 61.5 \end{bmatrix} \mu_1 = \begin{bmatrix} -5.0 \\ 4.0 \end{bmatrix} \mu_2 = \begin{bmatrix} 5.6 \\ -8.2 \end{bmatrix} \tag{4}$$

Comment:- In this configuration both algorithms converged fast: 8 iterations for K-Means and 62 for EM. The big difference was on accuracy: K-Means got 10.17% while EM obtained 99.33%. When analysing the original data, we can see it is nearly linearly separable, the difference on this

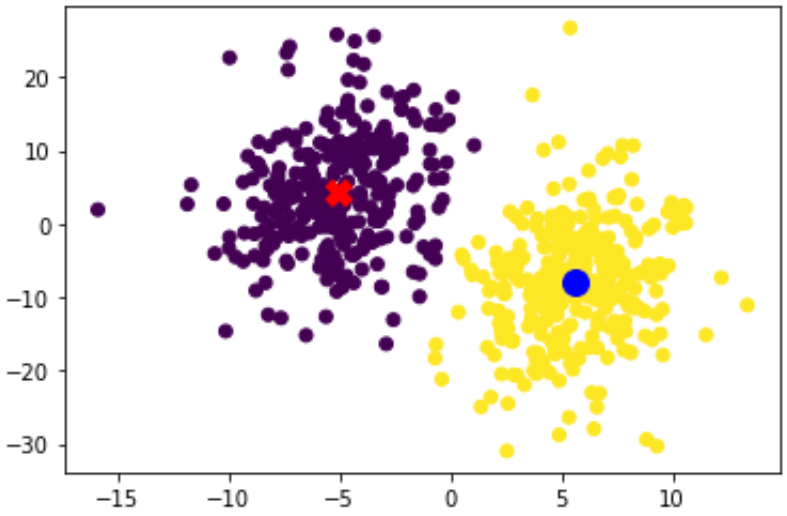
performance was due to small estimates $\hat{\mu_1}$ and $\hat{\mu_2}$ when compared with the real class means, besides the inverted situation.



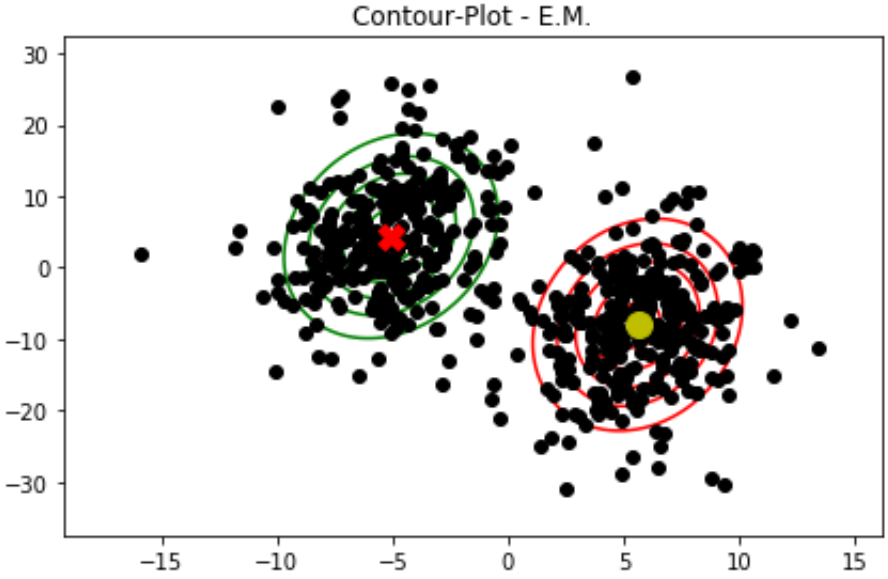
New Samples - Config 4



K-Means Algorithm - Config 4



EM Algorithm - Config 4



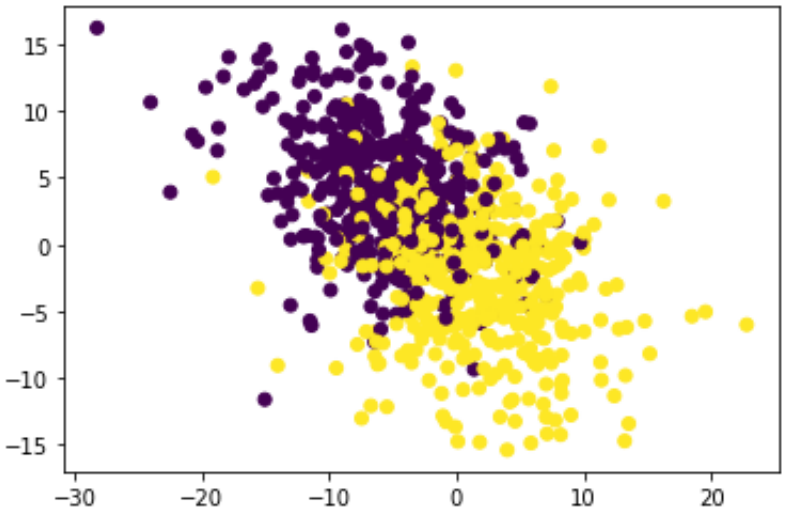
EM Contour-Plot - Config 4

Configuration No.5 - `num_samples`$= 900$

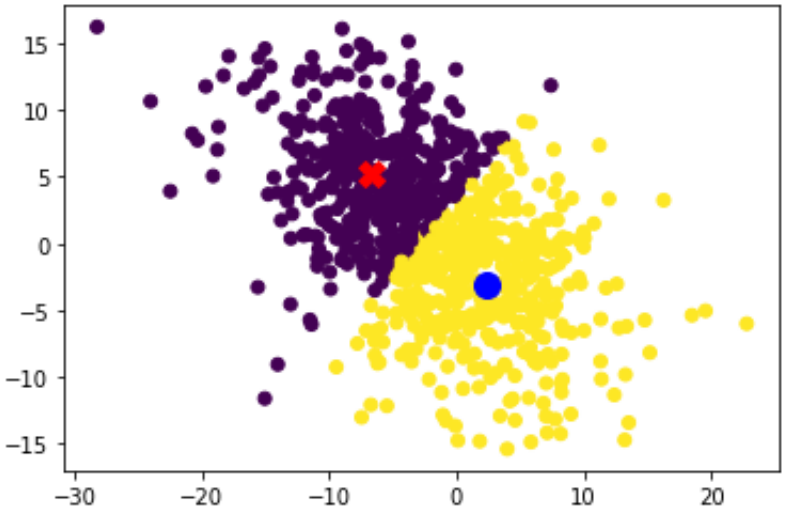
$$\Sigma = \begin{bmatrix} 28.0 & -6.0 \\ -6.0 & 24.6 \end{bmatrix} \mu_1 = \begin{bmatrix} -5.5 \\ 4.3 \end{bmatrix} \mu_2 = \begin{bmatrix} 1.6 \\ -2.5 \end{bmatrix} \tag{5}$$

Comment:- In this final configuration we could see a large population of mixed class observations. K-Means performed quite well, despite of the initial settings for $\hat{\mu_1}$ and $\hat{\mu_2}$, with a convergence in 11 iterations. We can see the *imaginary line* defined by the algorithm to separate the

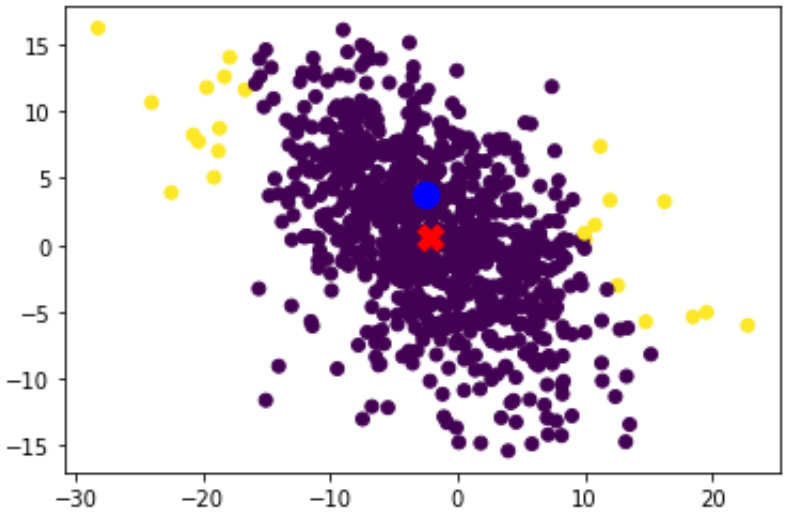
mixed samples, assuming their distance from the centroids of each cluster. Final accuracy was 80.33% after 8 iterations. EM algorithm had a different behavior: it tried to separate the populations considering the initial $\hat{\mu_1}$ and $\hat{\mu_2}$, very close of each other, and the highly disperse and mixed observations made the algorithm almost collapse both means in a central location. The result was quite unusual because it was obtained a not linear separation but a concentric class clustering of the sample of data. The accuracy was also poor: 50.11% with 200 iterations limits reached.



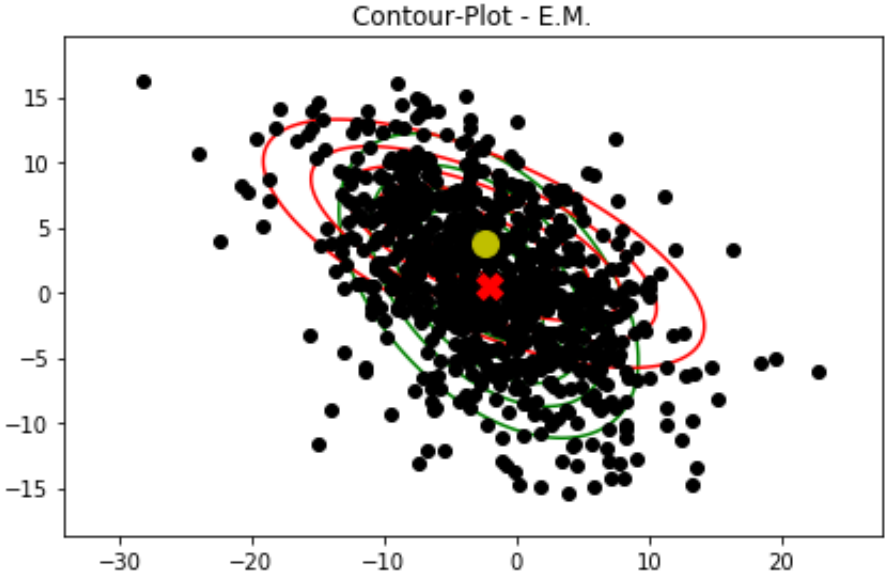
New Samples - Config 5



K-Means Algorithm - Config 5



EM Algorithm - Config 5



EM Contour-Plot - Config 5

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`

[19]:
```
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
```

2.1 1. Basic Q Learning experiments

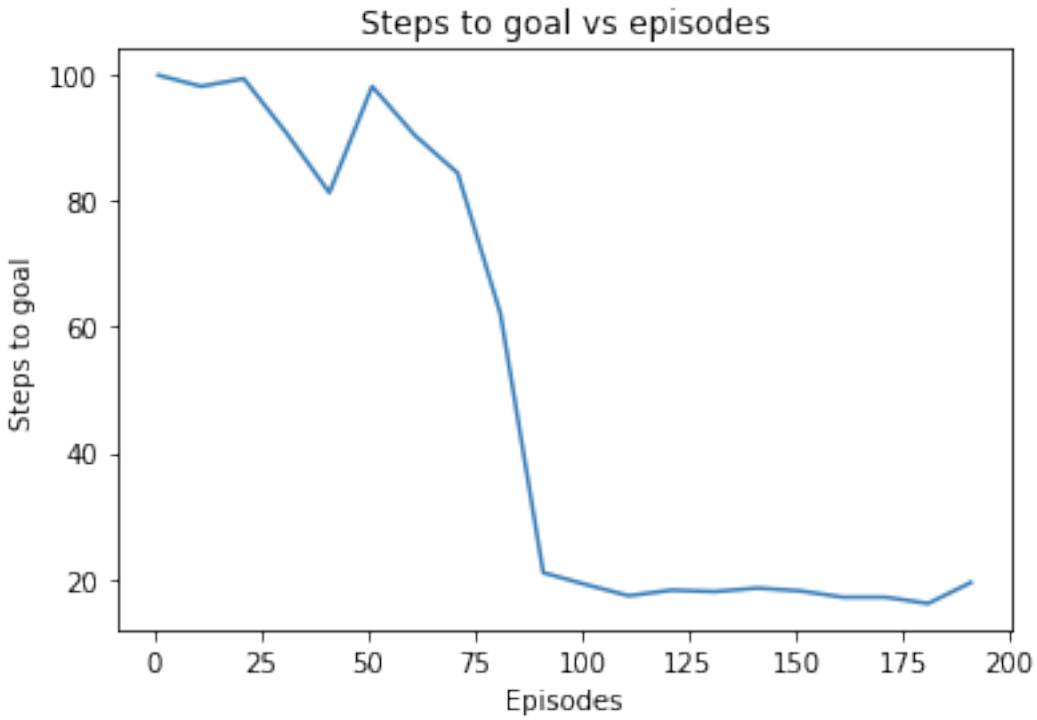
(a) Run your algorithm several times on the given environment. Use the following hyperparameters:

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.

[20]:
```
# 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, init_beta=None,␣
 ↪k_exp_sched=None, SSeed = 12)
```

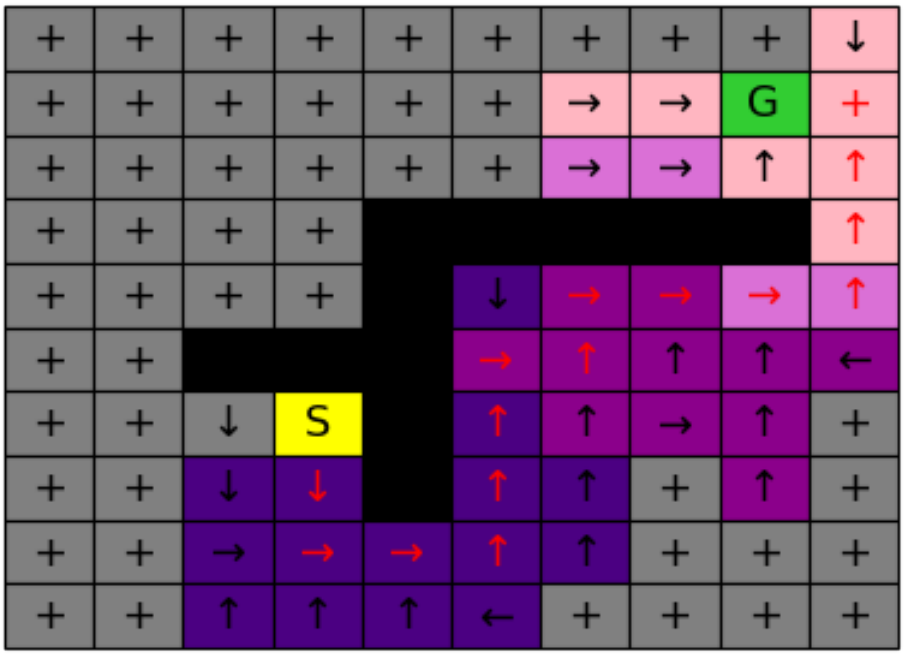
Plot the steps to goal vs training iterations (episodes):

[21]:
```
# TODO: Plot the steps vs iterations
plot_steps_vs_iters(steps_vs_iters, block_size=10)
```



Visualize the learned greedy policy from the Q values:

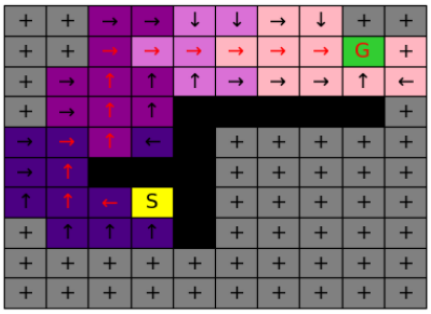
[22]:
```
# TODO: plot the policy from the Q value
plot_policy_from_q(q_hat, env)
```

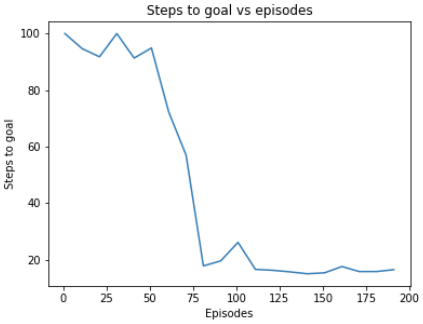


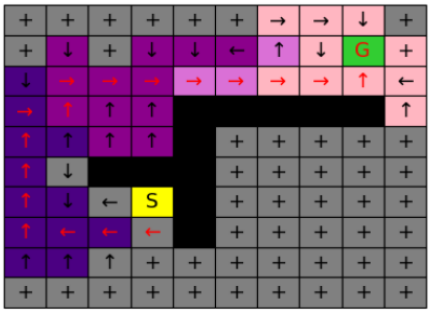
```
<Figure size 720x720 with 0 Axes>
```

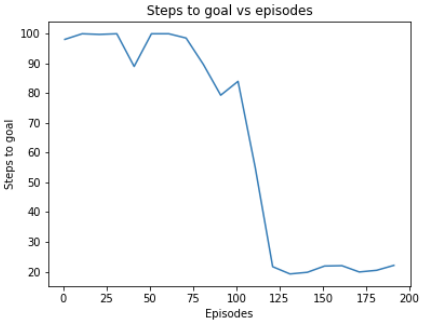
Additional Runs for Default Configuration

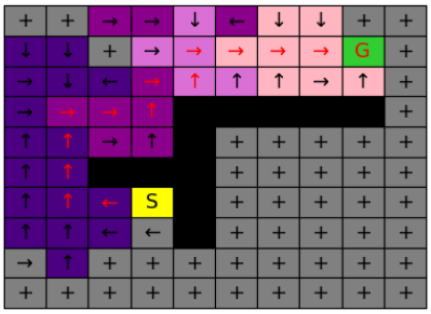
We have run the default configuration with ϵ-greedy policy 04 additional times and the results obtained are summarized below:

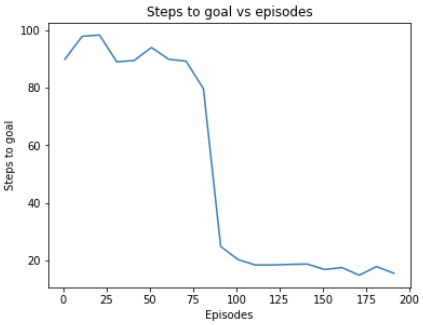

S
G

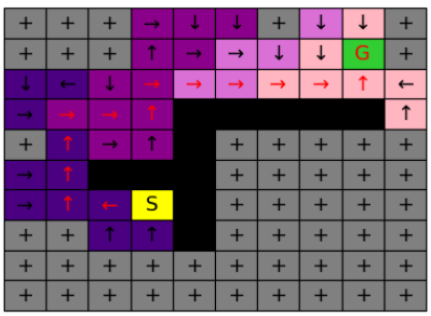

Steps to goal vs episodes
Steps to goal
Episodes
100
80
60
40
20
0
25
50
75
100
125
150
175
200

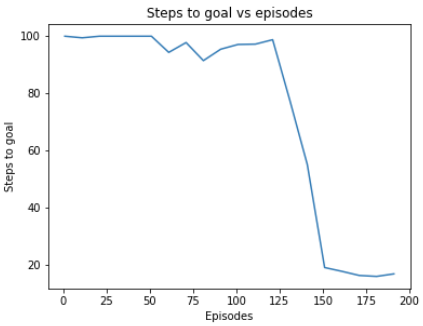

S
G


Steps to goal vs episodes
Steps to goal
Episodes
100
90
80
70
60
50
40
30
20
0
25
50
75
100
125
150
175
200


S
G


Steps to goal vs episodes
Steps to goal
Episodes
100
80
60
40
20
0
25
50
75
100
125
150
175
200


S
G

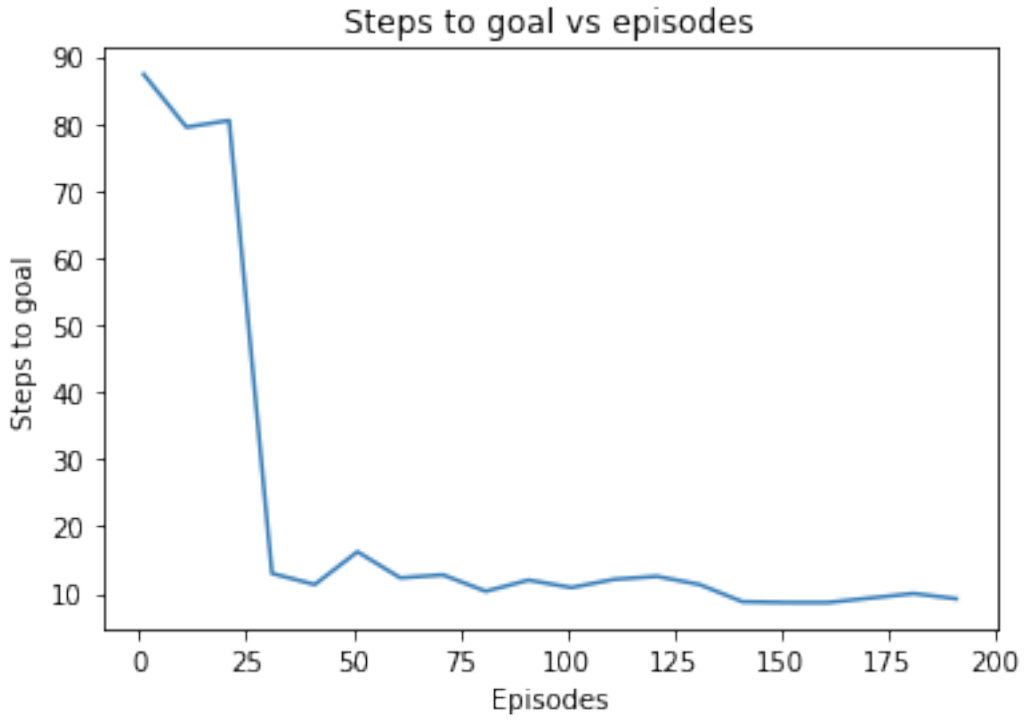

Steps to goal vs episodes
Steps to goal
Episodes
100
80
60
40
20
0
25
50
75
100
125
150
175
200

(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.

[23]:
```
# 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 = goals=[[1, 8],[5, 6]]
env = MazeEnv(goals=goal_locs)

# TODO: Run Q-learning:
q_hat, steps_vs_iters = qlearn(env, num_iters, alpha, gamma, epsilon, max_steps,
                               use_softmax_policy, init_beta=None,␣
↪k_exp_sched=None, SSeed = 12)
```

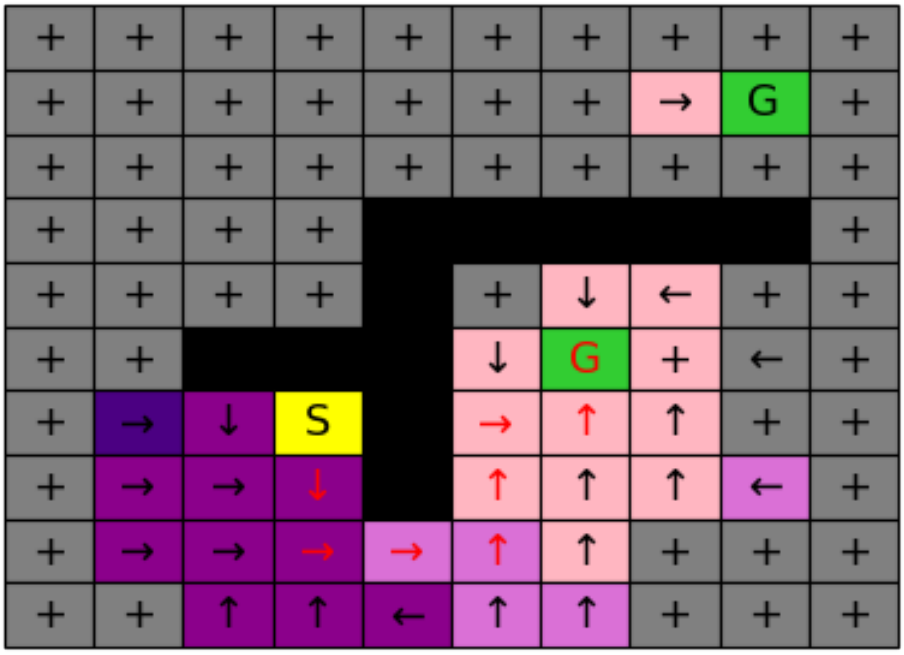
Plot the steps to goal vs training iterations (episodes):

[24]:
```
# TODO: Plot the steps vs iterations
plot_steps_vs_iters(steps_vs_iters, block_size=10)
```



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

[25]:
```
# TODO: plot the policy from the Q values
plot_policy_from_q(q_hat, env)
```




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

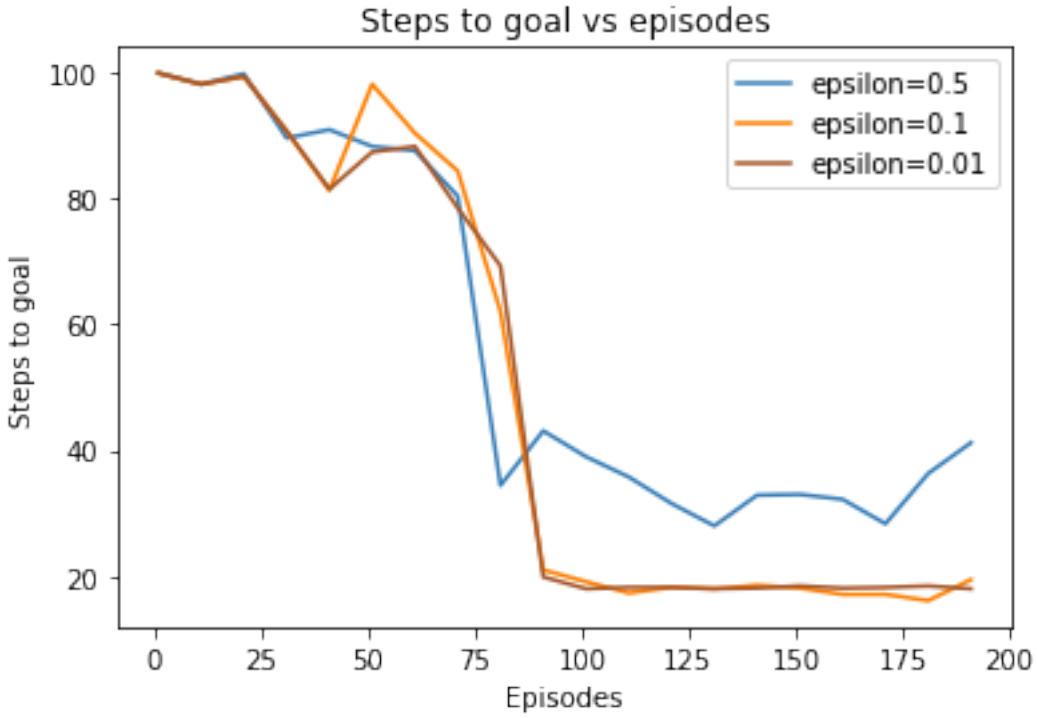
2.2 2. Experiment with the exploration strategy, in the original environment

(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.

[26]:
```
# 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 = [0.5, 0.1, 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, SSeed = 12)
   steps_vs_iters_list.append(steps_vs_iters)
```

[27]:
```
# TODO: Plot the results
label_list = ["epsilon={}".format(eps) for eps in epsilon_list]
plot_several_steps_vs_iters(steps_vs_iters_list, label_list, block_size=10)
```



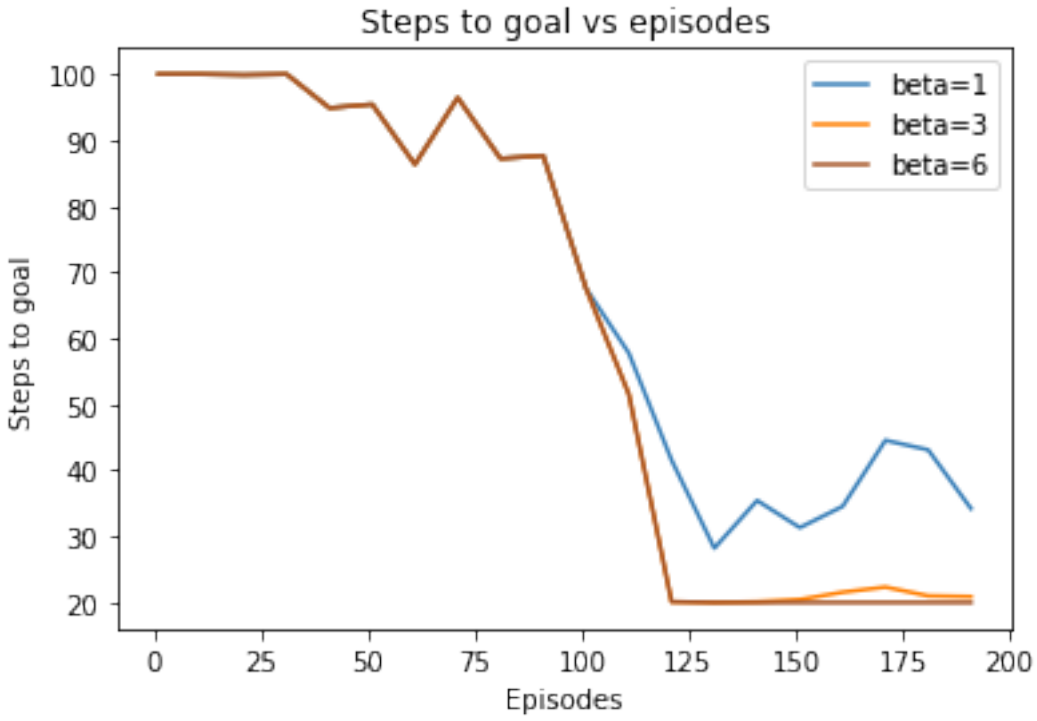
Discuss the costs and benefits of higher and lower exploration rates - We can see through the simulations that higher values for ϵ influences the number of steps to the goal. Higher values means the agent will follow exploration-exploitation in a $50\% - 50\%$ basis meaning the optimal path will be followed only half of the time. This obviously affects performance.

(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.

[28]:
```
# 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, init_beta=beta,␣
 ↪k_exp_sched=k_exp_schedule, SSeed=98)
    steps_vs_iters_list.append(steps_vs_iters)
```

[29]:
```
label_list = ["beta={}".format(beta) for beta in beta_list]
# TODO:
plot_several_steps_vs_iters(steps_vs_iters_list, label_list, block_size=10)
```



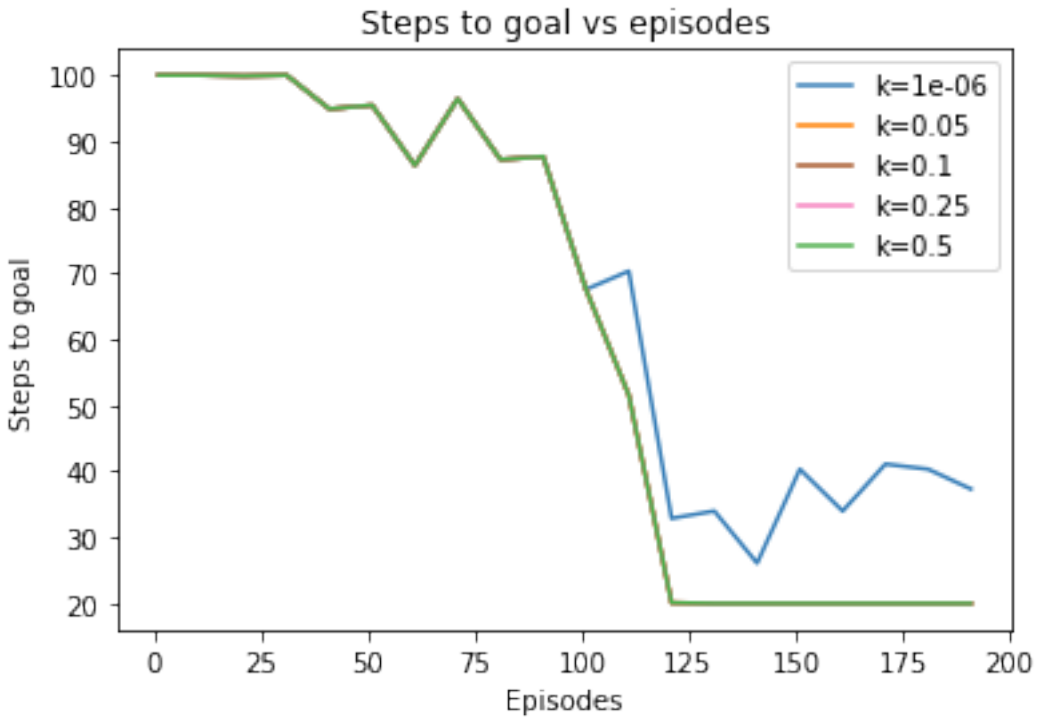
(c) Instead of fixing the $\beta = \beta_0$ to the initial value, we will increase the value of β as the number of episodes t increase:

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

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

[30]:
```
# 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 = [1e-06, 0.05, 0.1, 0.25, 0.5] # Added one additional k to␣
 ↪contrast

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, init_beta=beta,␣
 ↪k_exp_sched=k_exp_schedule, SSeed = 98)
    steps_vs_iters_list.append(steps_vs_iters)
```

[31]:
```
# TODO: Plot the steps vs iterations
label_list = ["k={}".format(k_exp_schedule) for k_exp_schedule in␣
 ↪k_exp_schedule_list]
plot_several_steps_vs_iters(steps_vs_iters_list, label_list, block_size=10)
```



Compare the results obtained with this approach to those obtained with a static β value. - As we could see from graph, all values proposed to k have generated high values for β as the number of episodes increase. In the limit, the interval $[0.05, 0.1, 0.25, 0.5]$ has no difference on their curves as the assiciated values for β are explosive causing the agent's policy behaves like *greedy* policy. On the other side, the small k added to the series makes the policy become less greedy after episode 120 and that's why we observe a divergence of the curve for $k = 1e - 06$ which confirms what we've seen in lecture 10. With static β we have seen a similar behaviour: for small values of β, agent's policy becomes more random because action picking is over a more balanced distribution of probabilities over possible actions. In the limit, for $\beta = 0$, we will have a uniform Boltzman softmax policy, where all actions have the same probability.

2.3 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.

[32]:
```
# TODO: Implement ProbabilisticMazeEnv in maze.py
import numpy as np
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
```

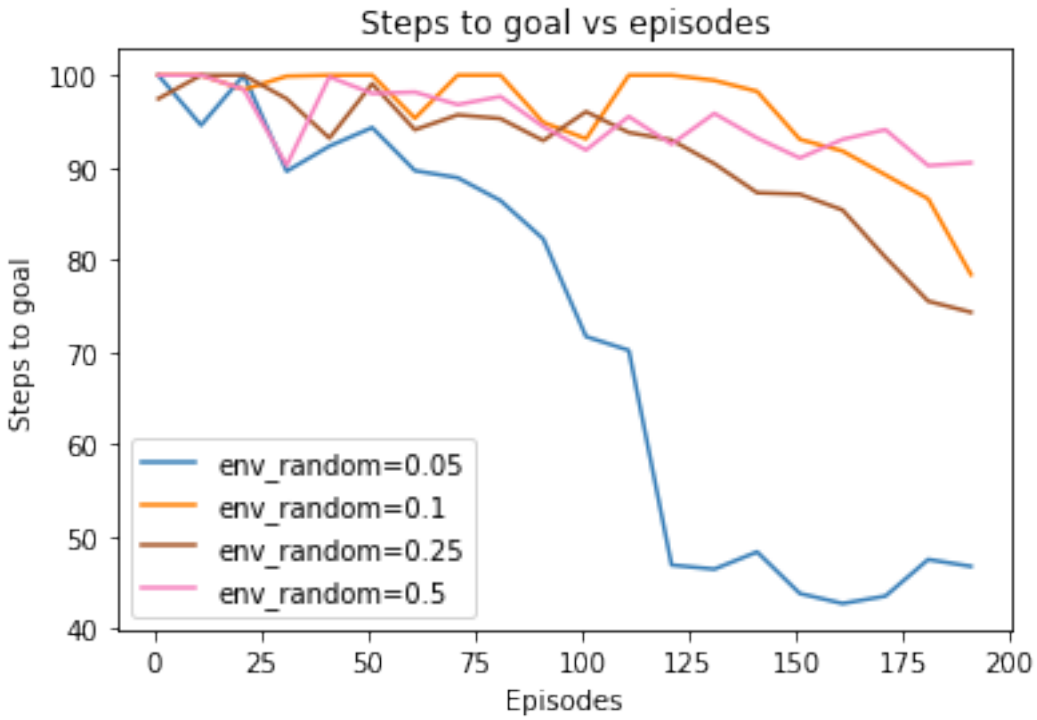
(b) Change the learning rule to handle the non-determinism, and experiment with different

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.

[33]:
```
# 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 = True

# 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()
    env.p_rand = 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, init_beta=1.0, k_exp_sched=0.
 ↪0, SSeed=12)
        avg_steps_vs_iters += steps_vs_iters
    avg_steps_vs_iters /= 10
    steps_vs_iters_list.append(avg_steps_vs_iters)
```

[34]:
```
label_list = ["env_random={}".format(env_p_rand) for env_p_rand in␣
 ↪env_p_rand_list]
plot_several_steps_vs_iters(steps_vs_iters_list, label_list, block_size=10)
```



How does performance vary as environment becomes stochastic? - The performance suffers when environment interfere on agent's actions. The number of steps to goal becomes random, specially in case of higher probabilities. We can see that for lower environment probability $p_{rand} = 0.05$ the policy converges to lower steps but in different level if compared with *greedy* policy (here we have over 40 steps after episode 100). For higher values of $p_{rand} > 0.05$ the policy becomes unconsistent with hight number of steps to reach the goal.

Summary of Experiments

Q-Learning Experiments

In this experiment we were required to implement a reinforcement learning agent that discovers the optimal (shortest) path to a goal. The environment consisted in a squared lattice, with *no border conditions*, i.e., no wraparound. Each cell of this lattice represent a possible move in one of four directions (Up, Right, Down or Left) from the agent in the environment. There is a known *start*, labeled by **S** where the agent starts moving and a *goal*, labeled by **G**, unknown by the agent. The lattice is encoded in such a way that cels with value 1 are possible cels the agent can move to, cels with value 0 that represents *barriers* and the goal, represented by a cel with -1. If the agent tries to move to barrier or outside the borders of the lattice it will remain in the same cel.

2.3.1 Overview

The experiments we conducted on this implementation we tested in 02 types of environment:

- a *Deterministic* environment, where the actions follows the *exploitation-exploration* criteria determined by the agent, where a percentage of its actions are randomly chosen or it follows a strict (*greedy*) behaviour trying to maximize the expected cumulative rewards;
- a *Stochastic* or *Probabilistic* environment where a percentage the actions are randomly chosen by the environment and imposed to the agent at a pre-defined probability level.

We conducted experiments on both environments as follows:

- Deterministic-I:- Runs over the basic environment under ϵ-greedy policy with the default parameters:
  - *Run-01* - Conducted 05 runs of the environment with the default parameters, i.e., `num_iters`= 200, `alpha`= 1.0, `gamma`= 0.9, `epsilon`= 0.1, `max_steps`= 100;
  - *Run-02* - Conducted 01 run of the environment with one additional goal;
  - *Run-03* - Conducted 01 run of the environment varying the exploration-exploitation rate ϵ with additional values, $\{0.5, 0.01\}$;
- Deterministic-II:- Runs over the basic environment under **softmax of Q-Values** policy with the default parameters
  - *Run-04* - Conducted 03 runs of the environment with the default parameters, but varying the parameter $\beta \in \{1, 3, 6\}$ keeping it *fixed* during the training;
  - *Run-05* - Conducted 03 runs of the environment with the default parameters, but keeping the initial parameter $\beta_0 = 1.0$ and varying as the number of episodes increases as as a function of k;
- Stochastic:- Runs over the *Probabilistic* environment under the default parameters with a chance the agent move in a random direction (as an external override of agent's choice):
  - *Run-06* - Conducted 01 run of the environment with the default parameters, allowing the agent move on desired direction in 95% of times and with 5% chance of moving random direction;
  - *Run-07* - Conducted 04 runs of the environment with the default parameters, except $\alpha = 0.5$ and allowing the agent move on desired direction in $\{95\%, 90\%, 75\%, 50\%\}$ of times and with the complement probability of moving in a random direction;

2.3.2 Results Of Experiments

For each run we will summarize the results of the experiments (detailed results can be found in previous section).

- Run-01: The optimal path was found in early episodes which means the agent followed the ϵ-greedy policy approximately 50% of the time.
- Run-02: When we added a second goal, the convergence was much faster, interesting to note that, when the agent discovered a nearer objective this one becomes its preferred and the second objective was abandoned, which makes perfect sense as it searches the shortest path to objective;

- Run-03: Changing the balance *exploration-exploitation* of our agent affected its performance. When the rate of exploitation becomes higher, the agent took more time to find the objective and, consequently, the convergence rate was harmed. For lower rates, agent pursues the optimal path almost all time which have trade-off: the less exploration of new paths the agent does, the more restrictive it becomes to already known paths;
- Run-04: Using $\beta > 1$ generated similar agent behaviour as it was under a *greedy* policy, and we also observed that for $\beta = 1$ we have increasing randomness of softmax policy;
- Run-05: For all values of k in the interval $[0.05 - 0.5]$ the behaviour is quite the same because it forces a huge increase of β as the episodes increases. We have tested for $k = 1e - 06$ the curve of iterations vs. episodes diverge from the others, as Boltzman softmax probabilities becomes more balanced;
- Run-06: The Stochastic Environment has affected agent's performance since it inflicts an external interference on the decision of which action will the agent take next. The agent will increase its exploitation of the environment with a random probability on top of the *softmax of Q-values* policy. This increases the level of instability of agent's movements with impacts on its performance.
- Run-07: In Stochastic Environment with higher rates of randomness turns our agent's search for the optimal path in almost a *random-walk*, where the performance suffers and optimal path not always is found.

3 3. Did you complete the course evaluation?

Answer: YES

4 Appendix - Code-listing

4.1 qlearning.py

[ ]:
```
import numpy as np
import math
import copy
from random import choices   # Softmax: Sample values with defined␣
↪probabilities

def qlearn(env, num_iters, alpha, gamma, epsilon, max_steps,␣
↪use_softmax_policy, init_beta=None, k_exp_sched=None, SSeed = 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)
```

```
    # Set seed to allow reproducibility
    if (SSeed != None):
        np.random.seed(SSeed )

    for i in range(num_iters):
        # TODO: Initialize current state by resetting the environment
        # curr_state = env.start
        curr_state = env.start[0]*env.m_size+env.start[1]  # Get position on␣
↪grid
        env.obs = env.start                                # Resets environment␣
↪start
        num_steps = 0
        done = False

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

            # Choose an action using policy derived from either softmax Q-value
            # or epsilon greedy
            if use_softmax_policy:
                assert(init_beta is not None)
                assert(k_exp_sched is not None)
                # TODO: Boltzmann stochastic policy (softmax policy)
                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,␣
↪action_space_size)
            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 = q_hat[curr_state, action] + \
                        alpha*(reward+gamma*max(q_hat[next_state]) - \
                               q_hat[curr_state, action])

                #print("\nStep %d | Next-State-%d | NewValue-%2.5f" %␣
↪(num_steps, next_state, new_value))
```

```
                # 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] = new_value

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

        #print("\nStep %d | %s " % (num_steps, "Done!" if done else "Fail..."))
        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_states, 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])==0.0):     # If never visited then randomize␣
↪action
        action = int(np.random.rand(1)*action_space_size)
    else:
        P = np.random.rand(1)
        if (P<=epsilon):
            action = int(np.random.rand(1)*action_space_size)
```

```
        else:
            action = q_hat[state].argmax()
    return(action)


def softmax_policy(q_hat, beta, state, action_space_size):
    """ 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_states, 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])==0.0):     # If never visited then randomize␣
 ↪action
        action = int(np.random.rand(1)*action_space_size)
    else:
        probs = stable_softmax(beta * q_hat[state], 0)
        action = choices(np.arange(action_space_size), weights=probs, k=1)[0]

    return(action)

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

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

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

    Returns:
        output: softmax(x) along the specified dimension
    """
```

```
    max_x = np.max(x, axis, keepdims=True)
    z = np.exp(x - max_x)
    output = z / np.sum(z, axis, keepdims=True)

    return output
```

4.2 maze.py

[ ]:
```
import numpy as np
import copy
import math

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

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


class MazeEnv:

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

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

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

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

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

```
        self.obs = self.start

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

            Args:
                a (int): action integer

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

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

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

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

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

        return state, reward, done

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

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

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

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

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

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

        return True

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

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

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

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

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

    def get_state_from_coords(self, row, col):
```

```
        state = row * self.m_size + col
        return state

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


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

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

            Args:
                goals (list): list of goals coordinates
                p_random (float): random action rate
        """
        MazeEnv.__init__(self, goals=[[2, 8]])
        self.p_rand = p_random

    def step(self, a):

        done, state, reward = False, self.obs, 0.0

        if (np.random.rand(1)<=self.p_rand):  # Move in a random direction
            state, reward, done = MazeEnv.step(self, int(np.random.rand(1)*self.
↪num_actions))
        else:
            state, reward, done = MazeEnv.step(self, a)

        return state, reward, done
```