1. Unsupervised Learning

In [15]:
```
%matplotlib inline
import scipy
import numpy as np
import itertools
import matplotlib.pyplot as plt
import random
```

1. Generating the data

First, we will generate some data for this problem. Set the number of points $N = 400$, their dimension $D = 2$, and the number of clusters $K = 2$, and generate data from the distribution $p(x|z = k) = \mathcal{N}(\mu_k, \Sigma_k)$. Sample 200 data points for $k = 1$ and 200 for $k = 2$, with

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

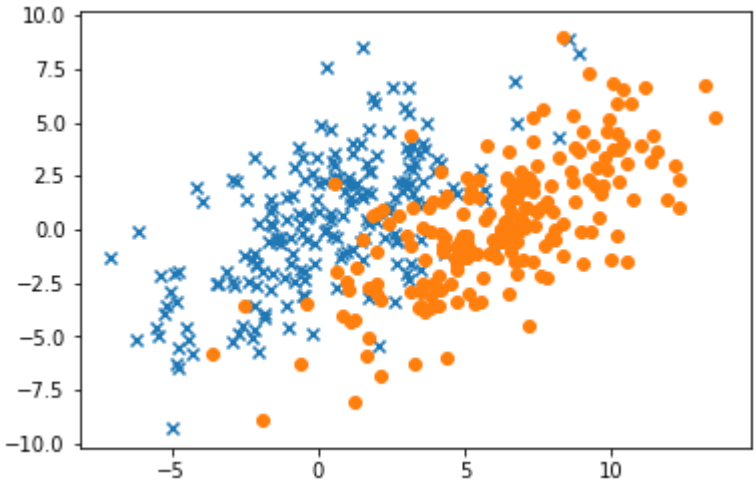
Here, $N = 400$. Since you generated the data, you already know which sample comes from which class. Run the cell in the IPython notebook to generate the data.

In [17]:
```
# TODO: Run this cell to generate the data
num_samples = 400
cov = np.array([[1., .7], [.7, 1.]]) * 10
mean_1 = [.1, .1]
mean_2 = [6., .1]

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

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

In [18]:
```
# TODO: Make a scatterplot for the data points showing the true cluster assign
ments of each point
# plt.plot(...) # first class, x shape
# plt.plot(...) # second class, circle shape

plt.scatter(x_class1[:,0], x_class1[:,1], marker="x")
plt.scatter(x_class2[:,0], x_class2[:,1])
```

Out[18]:
```
<matplotlib.collections.PathCollection at 0x1eb7a3db6c8>
```



2. Implement and Run K-Means algorithm

Now, we assume that the true class labels are not known. Implement the k-means algorithm for this problem. Write two functions: `km_assignment_step` , and `km_refitting_step` as given in the lecture (Here, `km_` means k-means). Identify the correct arguments, and the order to run them. Initialize the algorithm with

$$\hat{\mu}_1 = \begin{bmatrix} 0.0 \\ 0.0 \end{bmatrix}, \hat{\mu}_2 = \begin{bmatrix} 1.0 \\ 1.0 \end{bmatrix}$$

and run it until convergence. Show the resulting cluster assignments on a scatter plot either using different color codes or shape or both. Also plot the cost vs. the number of iterations. Report your misclassification error.

In [19]:
```
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
```

In [20]:
```
# 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] = ...
        r[:, k] = np.linalg.norm(data - np.array([Mu[:, k], ] * N), axis=1)**2

    # arg_min = ... # argmax/argmin along dimension 1
    # axis = 1 -> by rows
    arg_min = np.argmin(r, axis = 1)


    # R_new = ... # Set to zeros/ones with shape (N, K)
    # R_new[..., ...] = 1 # Assign to 1

    R_new = np.zeros((N,K))
    R_new[np.array(range(N)), arg_min] = 1

    return R_new
```

In [21]:
```
# TODO: K-means Refitting Step
def km_refitting_step(data, R, Mu):
    """ Compute K-Means refitting step.

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

    Returns:
        Mu_new: a DxK matrix for the new cluster means locations
    """
    N, D = data.shape # Number of datapoints and dimension of datapoint
    K = Mu.shape[1]  # number of clusters

    # axis = 0 will fix the column
    Mu_new = np.dot(data.T, R)/np.sum(R, axis = 0)
    return Mu_new
```

In [22]:
```
# 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)

# Empty list to store cost values across 100 iterations
cost_list = []
for it in range(max_iter):
    R = km_assignment_step(data, Mu)
    Mu = km_refitting_step(data, R, Mu)
    cost_list.append(cost(data, R, Mu))
    print(it, cost(data, R, Mu))

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

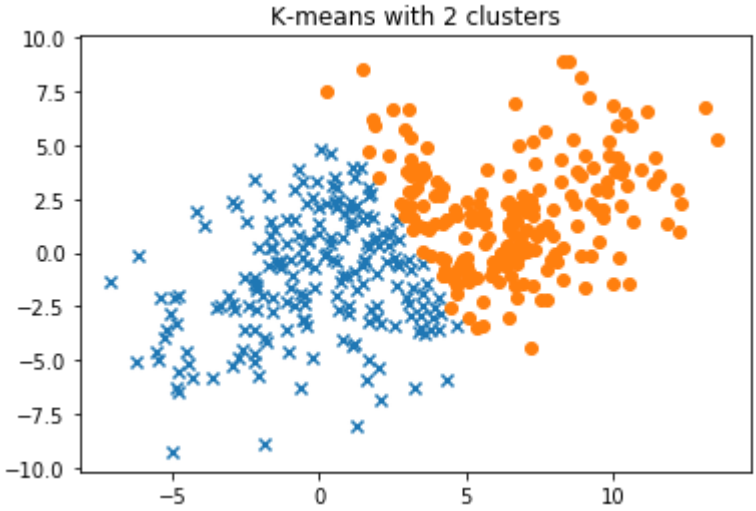
```
0 5965.112882370139
1 5784.405334402856
2 5690.517540407955
3 5613.911631270668
4 5584.230874622903
5 5568.057794787547
6 5563.389663832051
7 5563.181362525232
8 5562.84997138672
9 5562.84997138672
10 5562.84997138672
11 5562.84997138672
12 5562.84997138672
13 5562.84997138672
14 5562.84997138672
15 5562.84997138672
16 5562.84997138672
17 5562.84997138672
18 5562.84997138672
19 5562.84997138672
20 5562.84997138672
21 5562.84997138672
22 5562.84997138672
23 5562.84997138672
24 5562.84997138672
25 5562.84997138672
26 5562.84997138672
27 5562.84997138672
28 5562.84997138672
29 5562.84997138672
30 5562.84997138672
31 5562.84997138672
32 5562.84997138672
33 5562.84997138672
34 5562.84997138672
35 5562.84997138672
36 5562.84997138672
37 5562.84997138672
38 5562.84997138672
39 5562.84997138672
40 5562.84997138672
41 5562.84997138672
42 5562.84997138672
43 5562.84997138672
44 5562.84997138672
45 5562.84997138672
46 5562.84997138672
47 5562.84997138672
48 5562.84997138672
49 5562.84997138672
50 5562.84997138672
51 5562.84997138672
52 5562.84997138672
53 5562.84997138672
54 5562.84997138672
55 5562.84997138672
56 5562.84997138672
```

```
57 5562.84997138672
58 5562.84997138672
59 5562.84997138672
60 5562.84997138672
61 5562.84997138672
62 5562.84997138672
63 5562.84997138672
64 5562.84997138672
65 5562.84997138672
66 5562.84997138672
67 5562.84997138672
68 5562.84997138672
69 5562.84997138672
70 5562.84997138672
71 5562.84997138672
72 5562.84997138672
73 5562.84997138672
74 5562.84997138672
75 5562.84997138672
76 5562.84997138672
77 5562.84997138672
78 5562.84997138672
79 5562.84997138672
80 5562.84997138672
81 5562.84997138672
82 5562.84997138672
83 5562.84997138672
84 5562.84997138672
85 5562.84997138672
86 5562.84997138672
87 5562.84997138672
88 5562.84997138672
89 5562.84997138672
90 5562.84997138672
91 5562.84997138672
92 5562.84997138672
93 5562.84997138672
94 5562.84997138672
95 5562.84997138672
96 5562.84997138672
97 5562.84997138672
98 5562.84997138672
99 5562.84997138672
```

K-means Plot

In [23]:
```
# TODO: Make a scatterplot for the data points showing the K-Means cluster ass
ignments of each point
# plt.plot(...) # first class, x shape
# plt.plot(...) # second class, circle shape

class_1_val = data[class_1]
class_2_val = data[class_2]

plt.scatter(class_1_val[:,0], class_1_val[:,1], marker="x")
plt.scatter(class_2_val[:,0], class_2_val[:,1])
plt.title("K-means with 2 clusters")
```

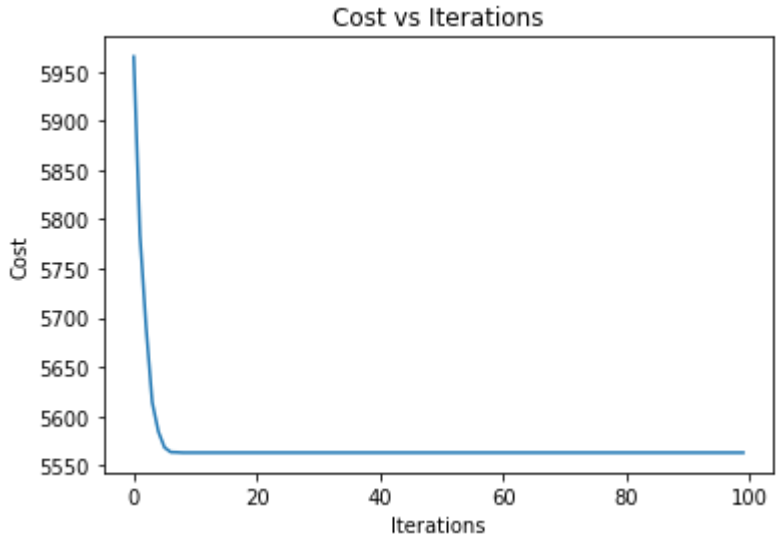
Out[23]:
```
Text(0.5, 1.0, 'K-means with 2 clusters')
```



Cost vs Iteration Plot - K means

In [24]:
```
plt.plot(range(max_iter), cost_list)
plt.xlabel("Iterations")
plt.ylabel("Cost")
plt.title("Cost vs Iterations")
```

Out[24]: Text(0.5, 1.0, 'Cost vs Iterations')



Misclassification Error - K means

In [25]:
```
misclass_1 = sum(data_full[class_1][:,2])
misclass_2 = sum(data_full[class_2][:,2] == 0)

mis_error = (misclass_1 + misclass_2)/400
mis_error
```

Out[25]: 0.2325

3. Implement EM algorithm for Gaussian mixtures

Next, implement the EM algorithm for Gaussian mixtures. Write three functions: `log_likelihood`, `gm_e_step`, and `gm_m_step` as given in the lecture. Identify the correct arguments, and the order to run them. Initialize the algorithm with means as in Qs 2.1 k-means initialization, covariances with $\hat{\Sigma}_1 = \hat{\Sigma}_2 = I$, and $\hat{\pi}_1 = \hat{\pi}_2$.

In addition to the update equations in the lecture, for the M (Maximization) step, you also need to use this following equation to update the covariance Σ_k:

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

Run the algorithm until convergence and show the resulting cluster assignments on a scatter plot either using different color codes or shape or both. Also plot the log-likelihood vs. the number of iterations. Report your misclassification error.

In [26]:
```
def normal_density(x, mu, Sigma):
    return np.exp(-.5 * np.dot(x - mu, np.linalg.solve(Sigma, x - mu))) \
        / np.sqrt(np.linalg.det(2 * np.pi * Sigma))
```

In [27]:
```
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 += ... # Compute the likelihood from the k-th Gaussian weighted by the mixing coefficients
            T += Pi[k]*normal_density(data[n,], Mu[:,k], Sigma[k])
        L += np.log(T)
    return L
```

In [28]:
```
# TODO: Gaussian Mixture Expectation Step
def gm_e_step(data, Mu, Sigma, Pi):
    """ Gaussian Mixture Expectation Step.

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

    Returns:
        Gamma: a NxK matrix of responsibilities
    """
    # Fill this in:
    N, D = data.shape # Number of datapoints and dimension of datapoint
    K = Mu.shape[1] # number of mixtures
    Gamma = np.zeros((N,K)) # zeros of shape (N,K), matrix of responsibilities
    for n in range(N):
        for k in range(K):
            # Gamma[n, k] = ....
            Gamma[n, k] = Pi[k]*normal_density(data[n,], Mu[:,k], Sigma[k])
        # Gamma[n, :] /= ... # Normalize by sum across second dimension (mixtu
res)
        Gamma[n, :] /= np.sum(Gamma[n, :], axis = 0)
    return Gamma
```

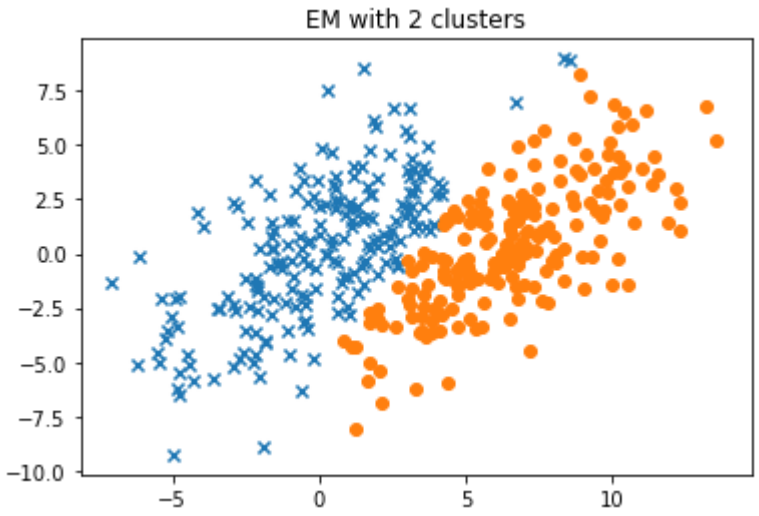
In [29]:
```
# TODO: Gaussian Mixture Maximization Step
def gm_m_step(data, Gamma):
    """ Gaussian Mixture Maximization Step.

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

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

        xmu = (data - Mk).T*np.sqrt(Gamma[:,k])
        Sigma[k] = np.dot(xmu, xmu.T)/Nk[k]
        # Sigma[k] = ...
    # Pi = ...
    Pi = Nk/N
    return Mu, Sigma, Pi
```

In [30]:
```
# 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

log_val = []
for it in range(max_iter):
    Gamma = gm_e_step(data, Mu, Sigma, Pi)
    Mu, Sigma, Pi = gm_m_step(data, Gamma)
    log_val.append(log_likelihood(data, Mu, Sigma, Pi))
    # print(it, log_likelihood(data, Mu, Sigma, Pi)) # This function makes the
computation longer, but good for debugging

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

EM algorithm Plot

In [31]:
```
# TODO: Make a scatterplot for the data points showing the Gaussian Mixture cl
uster assignments of each point
# plt.plot(...) # first class, x shape
# plt.plot(...) # second class, circle shape

class_1_val_EM = data[class_1]
class_2_val_EM = data[class_2]

plt.scatter(class_1_val_EM[:,0], class_1_val_EM[:,1], marker="x")
plt.scatter(class_2_val_EM[:,0], class_2_val_EM[:,1])
plt.title("EM with 2 clusters")
```

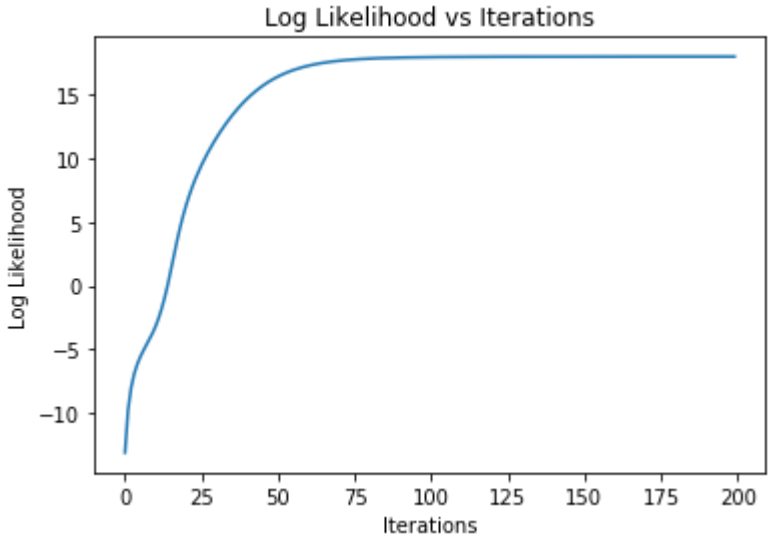
Out[31]:
```
Text(0.5, 1.0, 'EM with 2 clusters')
```



Log likelihood vs Iterations - EM

In [32]:
```
plt.plot(range(max_iter), log_val)
plt.xlabel("Iterations")
plt.ylabel("Log Likelihood")
plt.title("Log Likelihood vs Iterations")
```

Out[32]: Text(0.5, 1.0, 'Log Likelihood vs Iterations')



Misclassification Error - EM

In [33]:
```
misclass_1_EM = sum(data_full[class_1][:,2])
misclass_2_EM = sum(data_full[class_2][:,2] == 0)

mis_error_EM = (misclass_1_EM + misclass_2_EM)/400
mis_error_EM
```

Out[33]: 0.11

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?

Comparing the resulting cluster assignments to the labelled cluster assignment (from part 1), we can see that EM algorithm's cluster comes to very close proximity of what we want. K-Means algorithm clusters the points diagonally from left to right, but EM clusters the points diagonally from right to left.

Looking at the cost vs iterations for K-means and log likelihood vs iterations for EM, it looks like K-means algorithm converges around after ~5 iterations. This is very fast and probable given the size of data. But EM algorithm converges around after ~25 iterations. There is quite a noticeable difference in convergence rate between K-means and EM

The bottleneck between the two methods is the covariance matrix. In K-means we kept the covariance matrix constant, whereas EM constantly updates the covariance matrix. We have more parameters to converge for the algorithm to converge in the case of EM.

In [34]:
```
# Generate 5 New data here
x_class1_1 = np.random.multivariate_normal(mean_1, cov, num_samples // 2)
x_class2_1 = np.random.multivariate_normal(mean_2, cov, num_samples // 2)
xy_class1_1 = np.column_stack((x_class1_1, np.zeros(num_samples // 2)))
xy_class2_1 = np.column_stack((x_class2_1, np.ones(num_samples // 2)))
data_full_1 = np.row_stack([xy_class1_1, xy_class2_1])
np.random.shuffle(data_full_1)
data_1 = data_full_1[:, :2]
labels = data_full_1[:, 2]

x_class1_2 = np.random.multivariate_normal(mean_1, cov, num_samples // 2)
x_class2_2 = np.random.multivariate_normal(mean_2, cov, num_samples // 2)
xy_class1_2 = np.column_stack((x_class1_2, np.zeros(num_samples // 2)))
xy_class2_2 = np.column_stack((x_class2_2, np.ones(num_samples // 2)))
data_full_2 = np.row_stack([xy_class1_2, xy_class2_2])
np.random.shuffle(data_full_2)
data_2 = data_full_2[:, :2]
labels = data_full_2[:, 2]

x_class1_3 = np.random.multivariate_normal(mean_1, cov, num_samples // 2)
x_class2_3 = np.random.multivariate_normal(mean_2, cov, num_samples // 2)
xy_class1_3 = np.column_stack((x_class1_3, np.zeros(num_samples // 2)))
xy_class2_3 = np.column_stack((x_class2_3, np.ones(num_samples // 2)))
data_full_3 = np.row_stack([xy_class1_3, xy_class2_3])
np.random.shuffle(data_full)
data_3 = data_full_3[:, :2]
labels = data_full_3[:, 2]

x_class1_4 = np.random.multivariate_normal(mean_1, cov, num_samples // 2)
x_class2_4 = np.random.multivariate_normal(mean_2, cov, num_samples // 2)
xy_class1_4 = np.column_stack((x_class1_4, np.zeros(num_samples // 2)))
xy_class2_4 = np.column_stack((x_class2_4, np.ones(num_samples // 2)))
data_full_4 = np.row_stack([xy_class1_4, xy_class2_4])
np.random.shuffle(data_full)
data_4 = data_full_4[:, :2]
labels = data_full_4[:, 2]

x_class1_5 = np.random.multivariate_normal(mean_1, cov, num_samples // 2)
x_class2_5 = np.random.multivariate_normal(mean_2, cov, num_samples // 2)
xy_class1_5 = np.column_stack((x_class1_5, np.zeros(num_samples // 2)))
xy_class2_5 = np.column_stack((x_class2_5, np.ones(num_samples // 2)))
data_full_5 = np.row_stack([xy_class1_5, xy_class2_5])
np.random.shuffle(data_full)
data_5 = data_full_5[:, :2]
labels = data_full_5[:, 2]
```

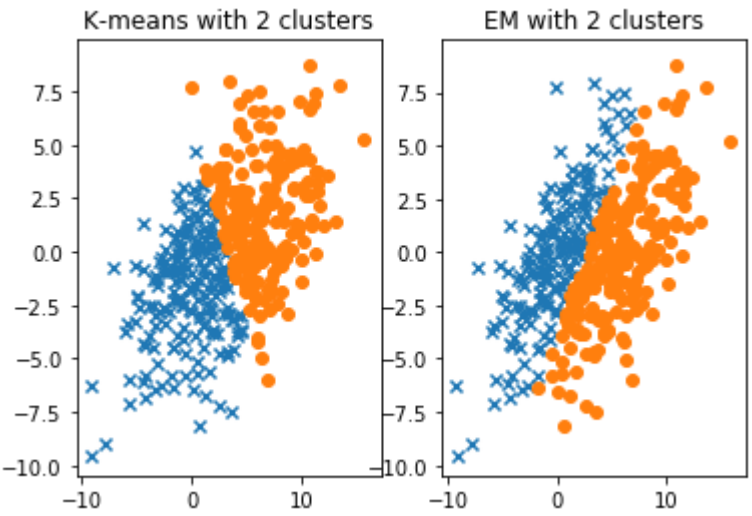
In [35]:
```
def get_misclassification(data, data_full):
    """helper function
    Returns misclassification rate using Kmeans and EM.
    Returns 2 classification scatter plots.
    """
    # K-means
    N, D = data.shape
    K = 2
    max_iter = 100
    class_init = np.random.binomial(1., .5, size=N)
    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)

    # Empty list to store cost values across 100 iterations
    cost_list = []
    for it in range(max_iter):
        R = km_assignment_step(data, Mu)
        Mu = km_refitting_step(data, R, Mu)
        cost_list.append(cost(data, R, Mu))
        #print(it, cost(data, R, Mu))

    class_1_KM = np.where(R[:, 0])
    class_2_KM = np.where(R[:, 1])


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

    log_val = []
    for it in range(max_iter):
        Gamma = gm_e_step(data, Mu, Sigma, Pi)
        Mu, Sigma, Pi = gm_m_step(data, Gamma)
        log_val.append(log_likelihood(data, Mu, Sigma, Pi))
        # print(it, log_likelihood(data, Mu, Sigma, Pi)) # This function makes
the computation longer, but good for debugging

    class_1_EM = np.where(Gamma[:, 0] >= .5)
    class_2_EM = np.where(Gamma[:, 1] >= .5)


    misclass_1_KM = sum(data_full[class_1_KM][:,2])
    misclass_2_KM = sum(data_full[class_2_KM][:,2] == 0)

    mis_error_KM = (misclass_1_KM + misclass_2_KM)/400

    misclass_1_EM = sum(data_full[class_1_EM][:,2])
    misclass_2_EM = sum(data_full[class_2_EM][:,2] == 0)

    mis_error_EM = (misclass_1_EM + misclass_2_EM)/400
```

```
    class_1_val = data[class_1_KM]
    class_2_val = data[class_2_KM]

    plt.subplot(1,2,1)
    plt.scatter(class_1_val[:,0], class_1_val[:,1], marker="x")
    plt.scatter(class_2_val[:,0], class_2_val[:,1])
    plt.title("K-means with 2 clusters")

    class_1_val_EM = data[class_1_EM]
    class_2_val_EM = data[class_2_EM]

    plt.subplot(1,2,2)
    plt.scatter(class_1_val_EM[:,0], class_1_val_EM[:,1], marker="x")
    plt.scatter(class_2_val_EM[:,0], class_2_val_EM[:,1])
    plt.title("EM with 2 clusters")

    print("Misclassification rate for K-means: {} and EM: {}".format(mis_error
_KM, mis_error_EM))
```

5 New data realization plots and misclassification rate

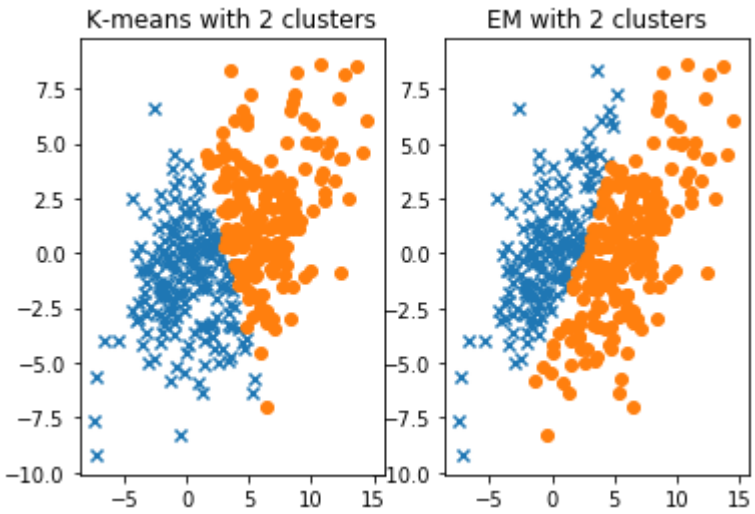
In [36]:
```
print(get_misclassification(data_1, data_full_1))
```

```
Misclassification rate for K-means: 0.27 and EM: 0.1025
None
```



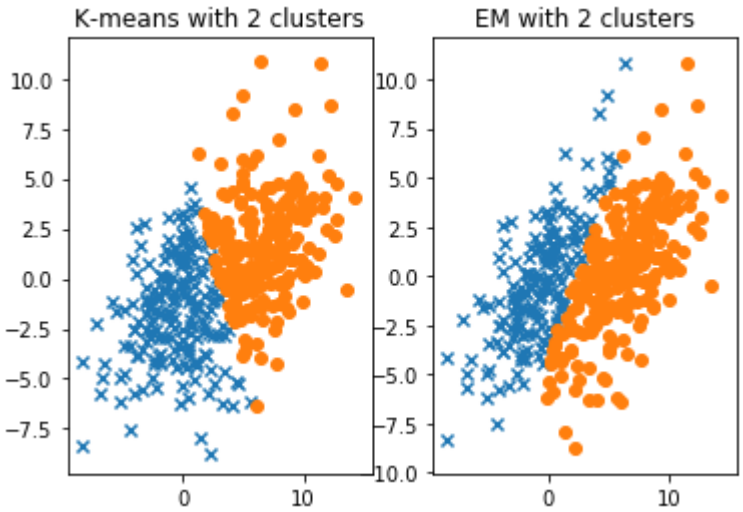
In [37]:
```
print(get_misclassification(data_2, data_full_2))
```

```
Misclassification rate for K-means: 0.245 and EM: 0.11
None
```



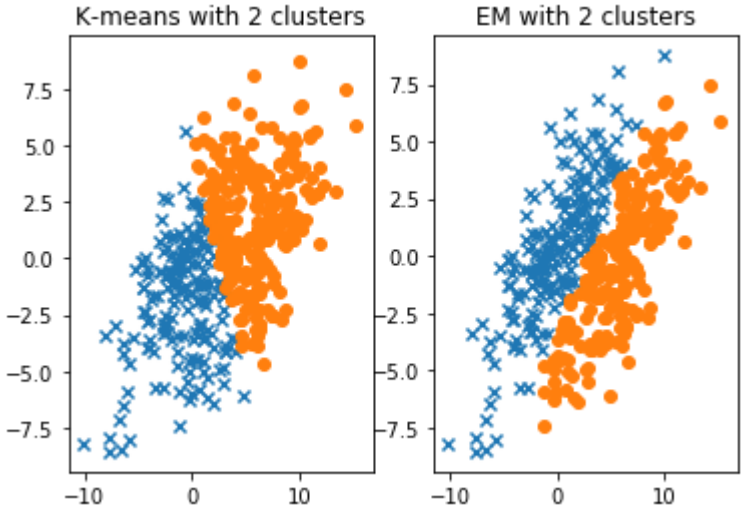
In [38]:
```
print(get_misclassification(data_3, data_full_3))
```

```
Misclassification rate for K-means: 0.195 and EM: 0.105
None
```



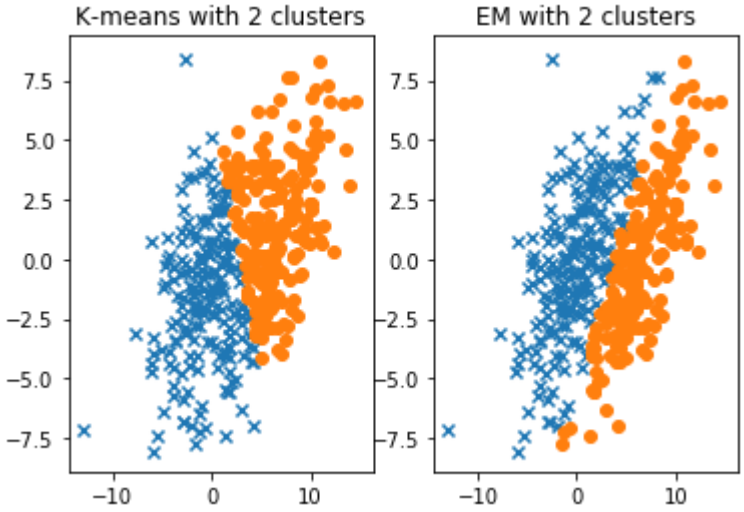
In [39]:
```
print(get_misclassification(data_4, data_full_4))
```

```
Misclassification rate for K-means: 0.27 and EM: 0.1275
None
```



In [40]:
```
print(get_misclassification(data_5, data_full_5))
```

```
Misclassification rate for K-means: 0.2225 and EM: 0.095
None
```



Looking at the above 5 plots and the corresponding misclassification rate, we can see that even with new data we come to the same conclusion as before.

EM algorithm has siginificantly better classification rate than K-means algorithm all the time. Scatter plot also shows that EM algorithm is better at classifying points than K-means.

In [ ]:

In [ ]: