I. Data Structure

The input dataset has 150 2-dimensional data points. Once loaded into the program, the dataset will be stored into a 150x2 data frame with two columns: 'x' for x values and 'y' for y value. After the completion of the k-means, the data structure will be reshaped to a 150 x 3 data frame with the third column storing the cluster each data point belongs to:

	x	y	cluster
0	-1.861331	-2.991683	1
1	-2.170092	-3.292318	1
2	-1.014081	0.385795	1
3	-2.912943	-2.579539	1
4	0.035721	-0.799698	1
..	...	...	...
145	7.613987	3.886738	0
146	5.699455	3.433903	0
147	4.268830	2.009613	2
148	2.100505	1.734405	2
149	3.808990	5.119568	0

Meanwhile, return a 3 x2 data frame to show the coordinate of three centroids:

	x	y
0	5.384881	4.747021
1	-1.060286	-0.739493
2	2.561971	1.304000

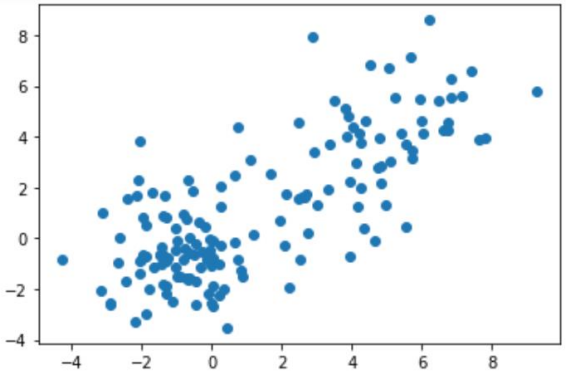
And for the GMM algorithm, the program will return a 150 x 150 covariance matrix, as well as three 2 x 3 data frames to store means, variance and covariance of three Gaussian distributions. The result will be showed on III. Gaussian Mixture Model Programming.

II. K-means Algorithm Programming

1. Firstly, I import the handy packages and the dataset and print out the scattergram of data points.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import itertools
import random
```

In [2]:
```
#Read the data
data_clu = pd.read_table('C:/Users/User/Desktop/USC/Machine Learning/HW2/clusters.txt',sep=',',
                         header = None,names = ['x','y'])

plt.scatter(data_clu['x'],data_clu['y'])
```



2. Next, define the function ***kmeans*** to run the k-means clustering. For function input, 'data' represents the dataset; 'numb_k' is the number of groups we want; 'numb_update' enters how many times of K-measwe want to do, and the remaining four inputs ask the user to enter the upper and lower of data points for assigning the initial centroids. Here, to optimize my coding, I put all calculations under one function as opposed to separating them then call several functions in by ***if __name__ == 'main'*** .

In [3]:
```
def kmeans (data,numb_k,numb_update,min_x,max_x,min_y,max_y):

#Randomly assign the initial centroids
 centroids = random.sample(list(itertools.product(range(min_x, max_x), range(min_y, max_y))), numb_k)

 for _ in range(numb_update):

    #Calculate the distances to centroids from each data entity:
    dist = np.zeros((data.shape[0],numb_k))
    for i in range(len(data)):
        for j in range(len(centroids)):
            dist[i,j] = np.linalg.norm(data.iloc[i]-centroids[j])

    #Find the nearest centroid:
    near_cen = np.zeros((data.shape[0],1))
    near_cen = np.argmin(dist,1)

    #Update centroids
    for cen_i in range(numb_k):
        centroids[cen_i] = np.mean(data[near_cen==cen_i])

 #Add to the original data frame the cluster each entity belongs to
 data_with_cluster = data.join(pd.DataFrame(near_cen,columns=['cluster']),how='outer')

 return centroids,data_with_cluster
```

3. Then, we use the "***keams***" function to iterate the clustering 10 times and print out the results. One optimization here is to assign different colors to different clusters for a more noticeable diagram.

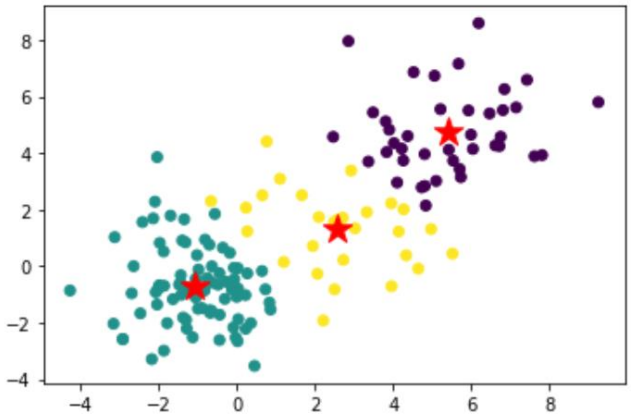
In [5]:
```
#Iterate the k-means cluster 10 times:
centroids,data_with_cluster=kmeans(data_clu,3,10,-5,10,-5,10)

#Three centroids
centroids

#Store the centroids to dataframe cs
cs=pd.DataFrame(centroids)

print(cs)

#Print the data with the cluster every entity belongs to
print(data_with_cluster)

#Plot the diagram
plt.scatter(data_clu['x'],data_clu['y'],c=data_with_cluster['cluster'])

plt.scatter(cs['x'],cs['y'],marker='*',c='r',s=300)
```

Three centroids:

	x	y
0	5.384881	4.747021
1	-1.060286	-0.739493
2	2.561971	1.304000

The visualized result with three centroids labeled as red stars:

```
Out[5]: <matplotlib.collections.PathCollection at 0x2378a5174c0>
```



III. Gaussian Mixture Model Programming

1. First of all, we run the K-means algorithm two time to get three groups of data points. I assume the three groups follow the Gaussian distribution and calculate their means, variance and covariance as our starting parameters to run the GMM:

```
In [132]: #Use the three clusters after 2 times k-means algorithm as the initial three Gaussian distributions

          centroids,data_with_cluster = kmeans (data_clu,3,2,-5,10,-5,10)
```

```
In [134]: #Split the dataset into three sub-datasets which respectively belongs to one Gaussian distribution
          df1 = data_with_cluster.loc[data_with_cluster['cluster']==0]

          df2 = data_with_cluster.loc[data_with_cluster['cluster']==1]

          df3 = data_with_cluster.loc[data_with_cluster['cluster']==2]

          data_GMM = pd.read_table('C:/Users/User/Desktop/USC/Machine Learning/HW2/clusters.txt',sep=',',
                                   header = None)
```

mu_init stores the initial means of three Gaussian distributions:

```
In [135]: #Calculate the means for each gaussian distribution, store them into a 3x2 matrix:mu_init
          mu1x = np.mean(df1['x'])

          mu1y = np.mean(df1['y'])

          mu1 = [mu1x,mu1y]


          mu2x = np.mean(df2['x'])

          mu2y = np.mean(df2['y'])

          mu2 = [mu2x,mu2y]


          mu3x = np.mean(df3['x'])

          mu3y = np.mean(df3['y'])

          mu3 = [mu3x,mu3y]


          mu_init = [mu1,mu2,mu3]
```

cov_init stores the initial covariance of three Gaussian distributions:

```
In [140]: #Calculate the covariances for each gaussian distribution, store them into a 3x2 matrix:mu_init
          cov1 = np.cov(df1['x'],df1['y'],bias=True)

          cov2 = np.cov(df2['x'],df2['y'],bias=True)

          cov3 = np.cov(df3['x'],df3['y'],bias=True)

          cov_init=[cov1,cov2,cov3]
```

weights_init calculates the initial weights of each data point on three Gaussian distributions; ***pi0*** records the proportion of data points on each Gaussian distribution:

In [8]:
```
#Calculate each data entity's initial weights on three Gaussian distributions
weights_init = np.ones((data_clu.shape[0], 3)) / 3

#pi:The proportion of all data entities on each Gaussian distribution
pi0 = weights_init.sum(axis=0) / weights_init.sum()
```

2. With the initial values of means, covariance, the weights, and proportion pi, we can initialize the GMM as follows, starting with the expectation step by updating the weights and pi. One noteworthy thing is that as opposed to putting all calculations under one big function, I separate all GMM algorithm steps into four independent functions to optimize my coding. The reason is to avoid the confusion of different scale matrices generated in GMM.

In [141]:
```
#Initialize the E-step:
#Updating the weights:

def update_weight (data,mu,cov,k,pi):
    pdf= np.zeros((data.shape[0],k))
    weights = np.zeros((data.shape[0],k))
    for i in range(k):
        pdf[:,i] = (pi[i])*multivariate_normal.pdf(x=data,mean=mu[i],cov=cov[i])

    weights = pdf/np.sum(pdf,axis=1).reshape(-1,1)

    return(weights)

#Get the new weights(frequency) of each data entity on different Gaussian distributions
```

In [143]:
```
#Update pi:the proportion of all data entities in each Gaussian distribution

def update_pi(weight):
    new_pi = weight.sum(axis=0)/weight.sum()
    return new_pi
```

3. Then, run the Maximization step by updating the Gaussian distribution parameters and return the covariance matrix:

In [144]:
```
#Initialize the M-step:
#Updating the Gaussian parameters:average,variance,covariance and the covariance matrix

def update_gaussian (data,weight,k):
    new_mu = np.zeros((k,2))
    new_var = np.zeros((k,2))
    new_cov_matrix = []
    new_cov = []

    for i in range(k):
        new_mu[i] = (np.average(data,axis=0,weights=weight[:,i]))

        new_var[i] = np.average((data - new_mu[i]) ** 2, axis=0, weights=weight[:, i])

        new_cov_matrix.append(weight[:,i]*(data-new_mu[i]).dot((data-new_mu[i]).T))

        new_cov.append(np.dot(((weight.T)[i].reshape(len(data),1) *
                              (data - new_mu[i])).T, (data - new_mu[i])) / (np.sum((weight.T)[i])))

    return new_mu,new_var,new_cov_matrix,new_cov
```

4. Next, define a function to visualize the GMM on 2-d projection.

In [46]:
```
#Plot the Gaussian 2-d projection on the data scattergram
def plot(data,mu,var,k):
    plt.figure()
    plt.axis([-5,10,-5,10])
    plt.scatter(data[0],data[1])
    color=['red','blue','lime']
    for i in range(k):
        for j in range(4):
            gaussian = Ellipse(mu[i],width= j * math.sqrt(var[i][0]),
                               height=j * math.sqrt(var[i][1]),edgecolor=color[i],
                               **{'fc': 'None','ls': '--'})
            (plt.gca()).add_patch(gaussian)
```

5. Finally, the program will call the main functions and do the GMM EM algorithm 20 times and prints out the means, covariance matrix and the visualized results:

In [48]:
```
#Call the main functions, doing the EM algorithm 20 times

if __name__ == '__main__':
    data = data_GMM
    mu = mu_init
    cov = cov_init
    k = 3
    pi = pi0
    weight = weights_init
    for i in range(20):
        weight = update_weight(data,mu,cov,k,pi)
        pi = update_pi(weight)
        mu,var,cov_matrix,cov=update_gaussian (data,weight,k)
        plot(data,mu,var,k)

    print(mu)
    print(var)
    print(cov)
    print(cov_matrix)
```

Three Gaussian means:

```
[[ 4.24991441  3.29982988]
 [-0.90756622 -0.07103146]
 [-1.13775638 -1.32294287]]
```

Three Gaussian variance:

```
[[4.06408712 5.05720279]
 [0.82687279 2.26675992]
 [1.50015373 0.79570872]]
```

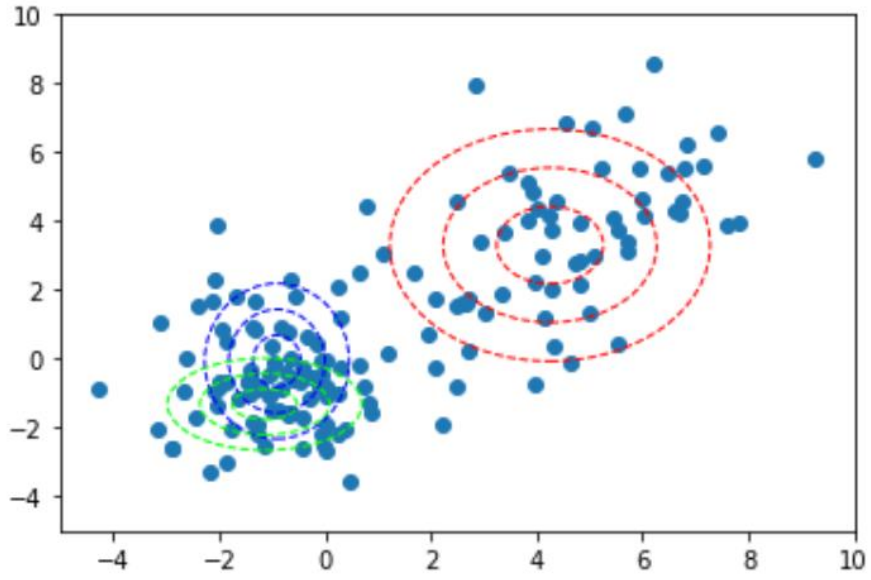
Three Gaussian covariance between x and y:

```
[array([[4.06408712, 2.42856463],
        [2.42856463, 5.05720279]]), array([[ 0.82687279, -0.6418151 ],
        [-0.6418151 ,  2.26675992]]), array([[1.50015373, 0.22436344],
        [0.22436344, 0.79570872]])]
```

Covariance matrix (150 x 150):

```
[          0         1         2         3         4          5         6    \
0    0.754120  0.984618  0.728978  0.260328  1.649106  21.772799  3.195326
1    0.791158  1.032978  0.765083  0.273154  1.730164  22.842730  3.354685
2    0.495064  0.646637  0.522539  0.176760  1.091902  14.377802  2.449989
3    0.791700  1.033837  0.791546  0.276797  1.736833  22.908159  3.564775
4    0.505288  0.659754  0.492637  0.174988  1.105848  14.596619  2.174593
..        ...       ...       ...       ...       ...        ...       ...
145 -0.237725 -0.310680 -0.280295 -0.088793 -0.530533  -6.960475 -1.411802
146 -0.095105 -0.124313 -0.115778 -0.036008 -0.213017  -2.791622 -0.593990
147  0.078439  0.102280  0.052832  0.024014  0.166669   2.220542  0.148170
148  0.225308  0.294240  0.229161  0.079292  0.495105   6.526856  1.045710
149 -0.085815 -0.111813 -0.043039 -0.024306 -0.179221  -2.401031 -0.043856

           7         8         9    ...        140        141       142  \
0    20.491071  2.133031  19.385546  ... -12.301385 -40.076502  3.708375
1    21.522724  2.237382  20.337642  ... -12.908832 -42.044625  3.875788
2    17.123091  1.340597  12.725257  ...  -8.551879 -26.280973  0.294714
3    23.702611  2.203709  20.351010  ... -13.198569 -42.056662  2.616464
4    14.079952  1.423514  12.988930  ...  -8.287808 -26.850003  2.280635
..         ...       ...        ...  ...        ...        ...       ...
145 -10.674533 -0.603870  -6.109919  ...   4.424670  12.600906  1.287760
146  -4.574534 -0.236642  -2.444276  ...   1.809593   5.038806  0.692402
147   0.212156  0.253071   2.016861  ...  -1.030526  -4.183365  1.504348
148   7.070771  0.621859   5.791559  ...  -3.798352 -11.966283  0.555006
149   1.000010 -0.296904  -2.206845  ...   0.967587   4.586305 -2.363969
```

Visualized GMM after 20 time iterations: (each color represents a Gaussian distribution, concentic ellipse indicates the distance from mean to one, two and three units of standard deviation)



IV. Compare the Results of K-means and GMM

1. The center of three clusters:

K-means

```
          x         y
0  5.384881  4.747021
1 -1.060286 -0.739493
2  2.561971  1.304000
```

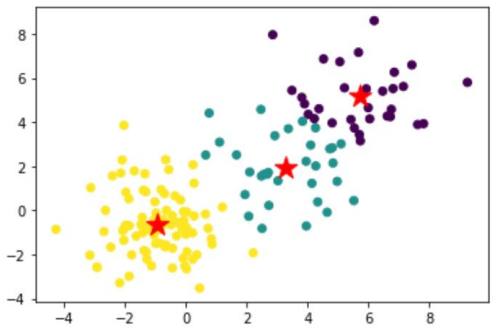
GMM

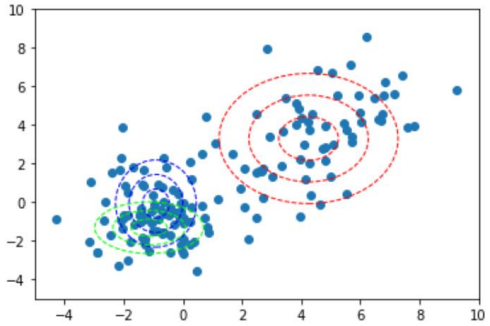
```
[[ 4.24991441  3.29982988]
 [-0.90756622 -0.07103146]
 [-1.13775638 -1.32294287]]
```

2. Visualized reuslts:

K-means GMM

```
Out[6]: <matplotlib.collections.PathCollection at 0x25dcc0c8220>
```





V. Challenges:

As the comparison result indicates above, the clustering outcome from K-means and from GMM is not necessarily the same one. Observing this, I summarize the challenge of my programming in one sentence: the setting of initial values.

In complementing both algorithms, the very first step we do is to randomly choose the initial centers of datasets waiting for being updated. For K-means, the programming without specific orders select three coordinate among the ranges of datasets; and for GMM, I use the rough grouping result from a time or two K-means iteration as the starting point.

Accordingly, we face two risks here. First, the initial centers might be either already divided the dataset into proper groups, or deviate from majority data points that cost us extra time to reach convergence. Second, because I assume the K-means result follows Gaussian distribution and afterward run the GMM based on this assumption, the other caveat we might hit is that the means and variances of each K-means group actually do not follow the Gaussian distribution. This concern is of significance given I don't run the point estimation for population parameters first.