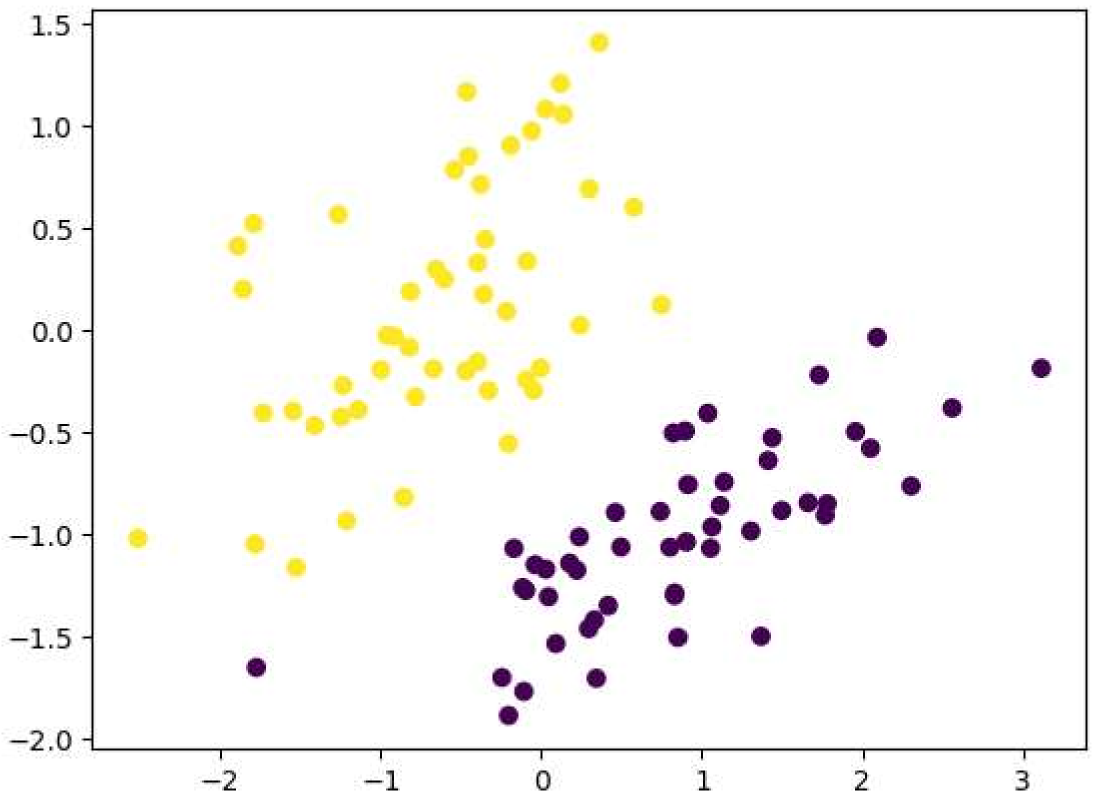
In [1]:
```
import numpy as np
import mltools as ml
import matplotlib.pyplot as plt
```

In [2]:
```
iris = np.genfromtxt("data/iris.txt",delimiter=None)
X, Y = iris[:,0:2], iris[:,-1] # get first two features & target
X,Y = ml.shuffleData(X,Y) # reorder randomly (important later)
X,_ = ml.rescale(X) # works much better on rescaled data
XA, YA = X[Y<2,:], Y[Y<2] # get class 0 vs 1
XB, YB = X[Y>0,:], Y[Y>0] # get class 1 vs 2
```

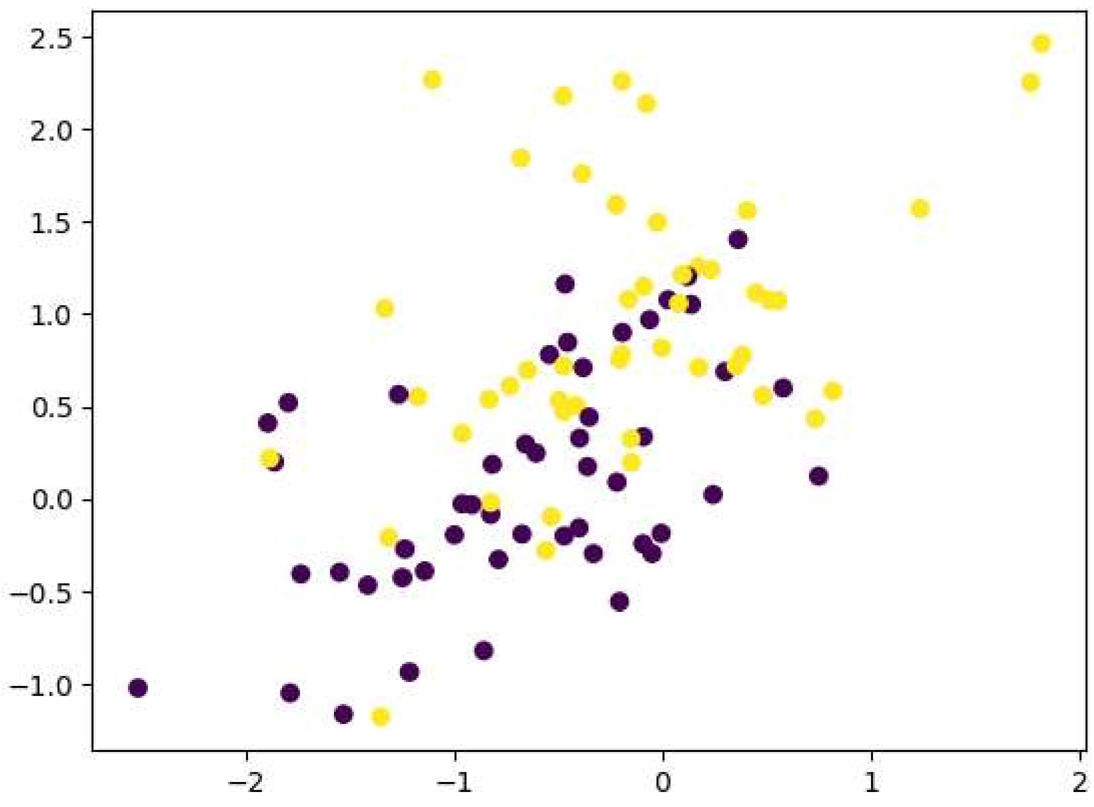
In [3]:
```
plt.scatter(XA[:,0],XA[:,1],c=YA)
```

Out[3]:
```
<matplotlib.collections.PathCollection at 0x1ecf653a0e0>
```



In [4]:
```
plt.scatter(XB[:,0],XB[:,1],c=YB)
```

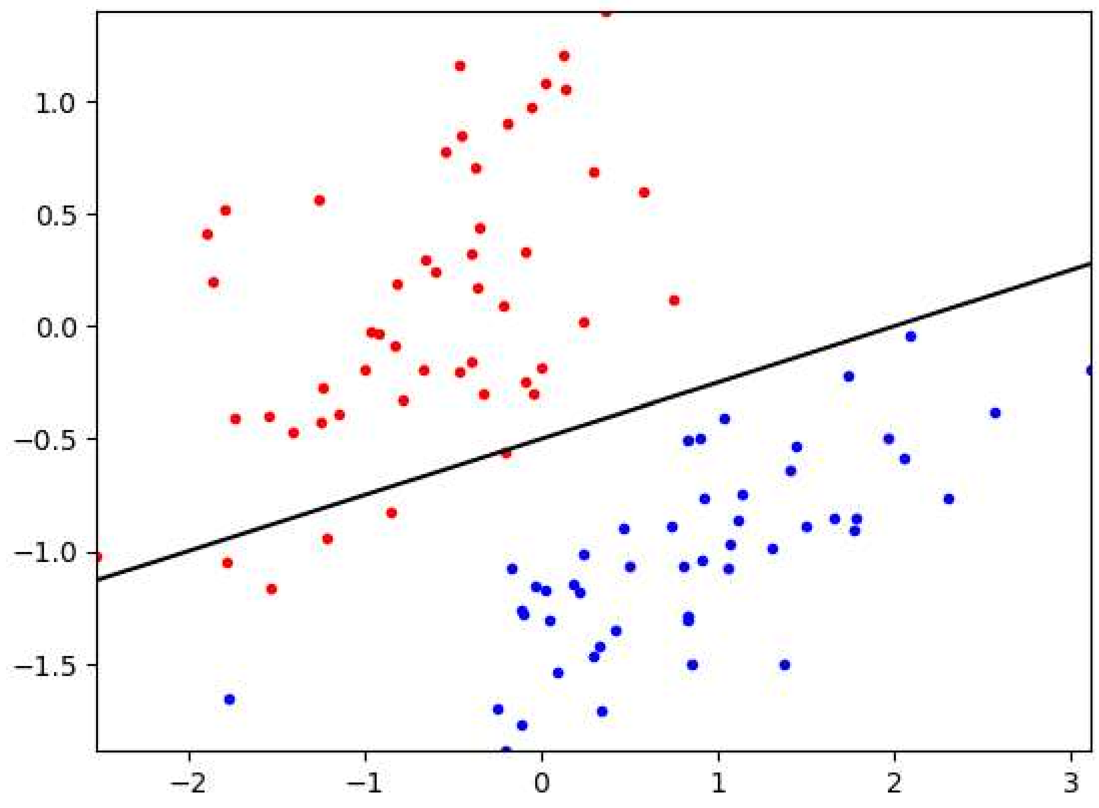
Out[4]:
```
<matplotlib.collections.PathCollection at 0x1ecf875da20>
```



In [5]:
```
from logisticClassify2 import *
learner = logisticClassify2(); # create "blank" learner
learner.classes = np.unique(YA) # define class labels using YA or YB
wts = np.array([0.5,-0.25,1.0]); # TODO: fill in values
learner.theta = wts; # set the learner's parameters
```

In [ ]:
```
    def cal_boundary(self,x1):
        return (-self.theta[0]-self.theta[1]*x1)/self.theta[2]
    def plotBoundary(self,X,Y):
        """ Plot the (linear) decision boundary of the classifier, along with data """
        if len(self.theta) != 3: raise ValueError('Data & model must be 2D');
        ax = X.min(0),X.max(0); ax = (ax[0][0],ax[1][0],ax[0][1],ax[1][1]);
        ## TODO: find points on decision boundary defined by theta0 + theta1 X1 + theta2 X2 == 0
        x1b = np.array([ax[0],ax[1]]) # at X1 = points in x1b

        x2b = np.array([self.cal_boundary(ax[0]),self.cal_boundary(ax[1])])   # TODO find x2 values as a function of x1's values
        A = Y==self.classes[0]; # and plot it:
        plt.plot(X[A,0],X[A,1],'b.',X[~A,0],X[~A,1],'r.',x1b,x2b,'k-'); plt.axis(ax); plt.draw();
```

In [6]:
```
learner.plotBoundary(XA,YA)
```



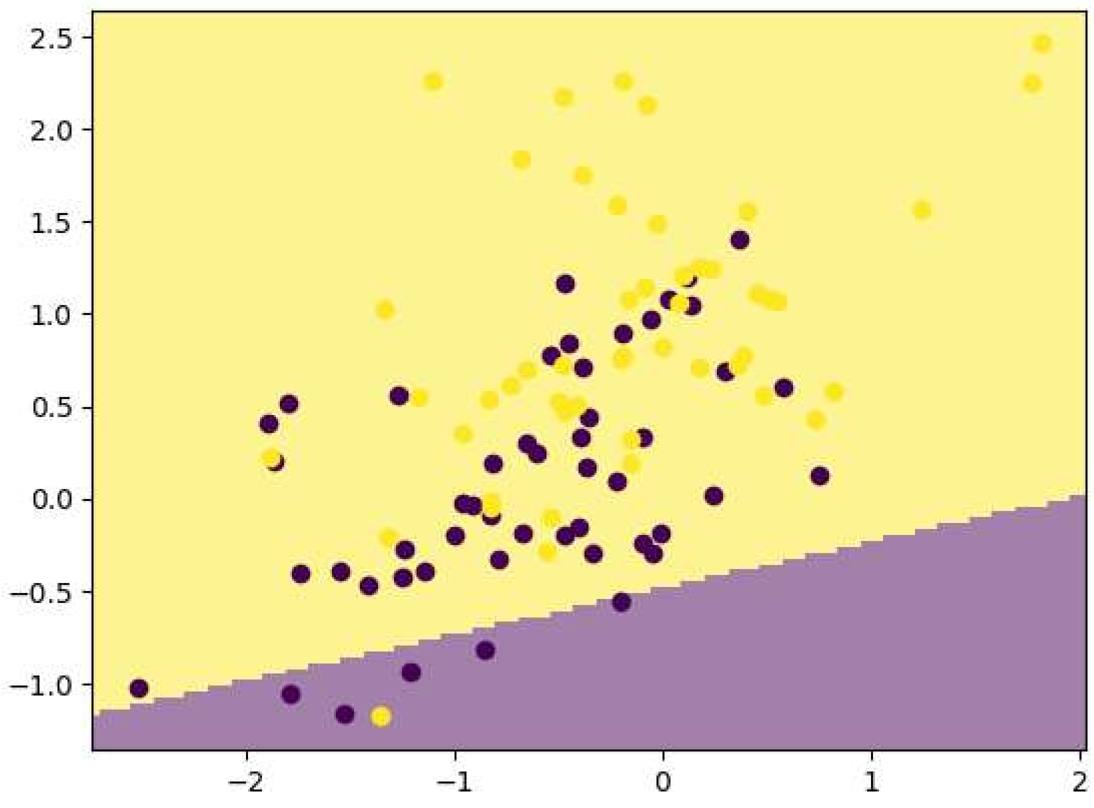
In [ ]:
```
    def predict(self, X):
        """ Return the predictied class of each data point in X"""
        XX = np.hstack((np.ones((X.shape[0], 1)), X))
        R = np.matmul(XX,self.theta)
        Z = np.sign(R)
        Yhat = np.vectorize(self.get_class)(Z)
        return Yhat
    def get_class(self,z):
        if z > 0:
            return self.classes[1]
        else:
            return self.classes[0]
```

In [7]:
```
e = learner.err(XA,YA)
print(e)
learner2 = logisticClassify2()
learner2.classes = np.unique(YB) # define class labels using YA or YB
learner2.theta = wts
e = learner2.err(XB,YB)
print(e)
```

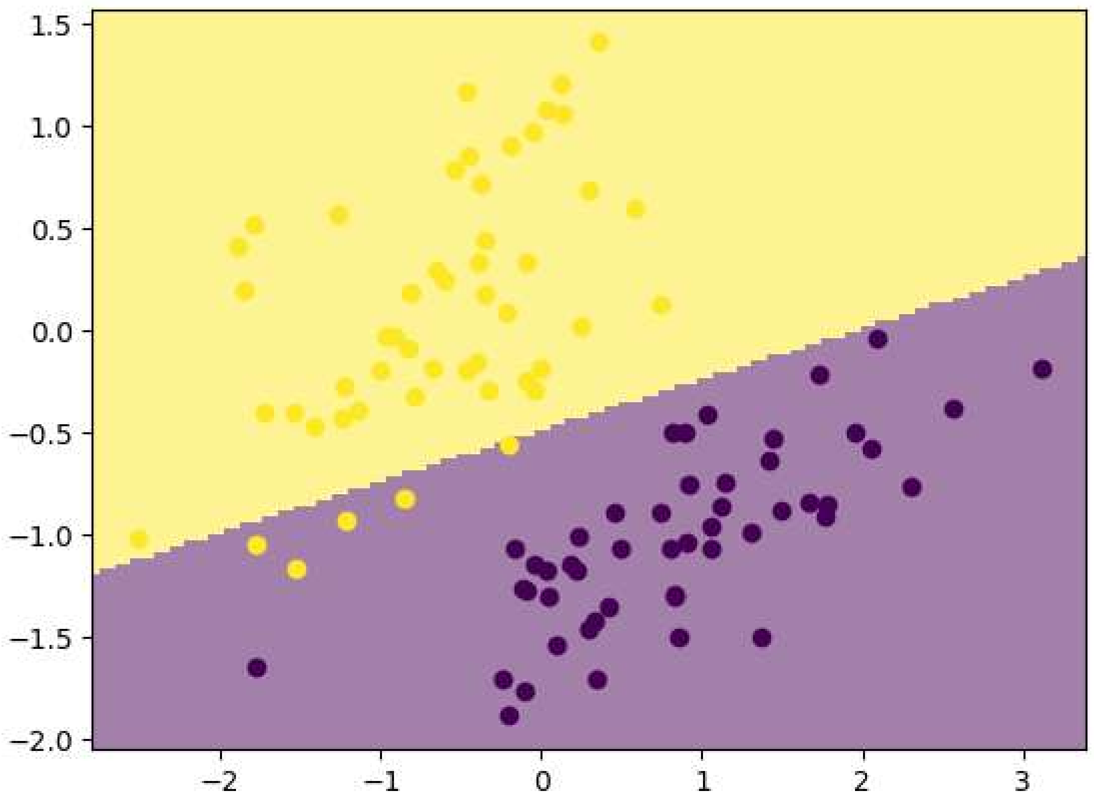
```
0.050505050505050504
0.46464646464646464
```

In [8]:
```
ml.plotClassify2D(learner2, XB, YB)
```

```
c:\mcsuci\273P\homework\Assignment 3\Assignment 3\mltools\plot.py:61: UserWarning: color is redundantly defined by the 'color'
keyword argument and the fmt string "ko" (-> color='k'). The keyword argument will take precedence.
  axis.plot( X[Y==c,0],X[Y==c,1], 'ko', color=cmap(cvals[i]), **kwargs )
```



In [9]:
```
ml.plotClassify2D(learner, XA, YA)
```



The gradient equations:

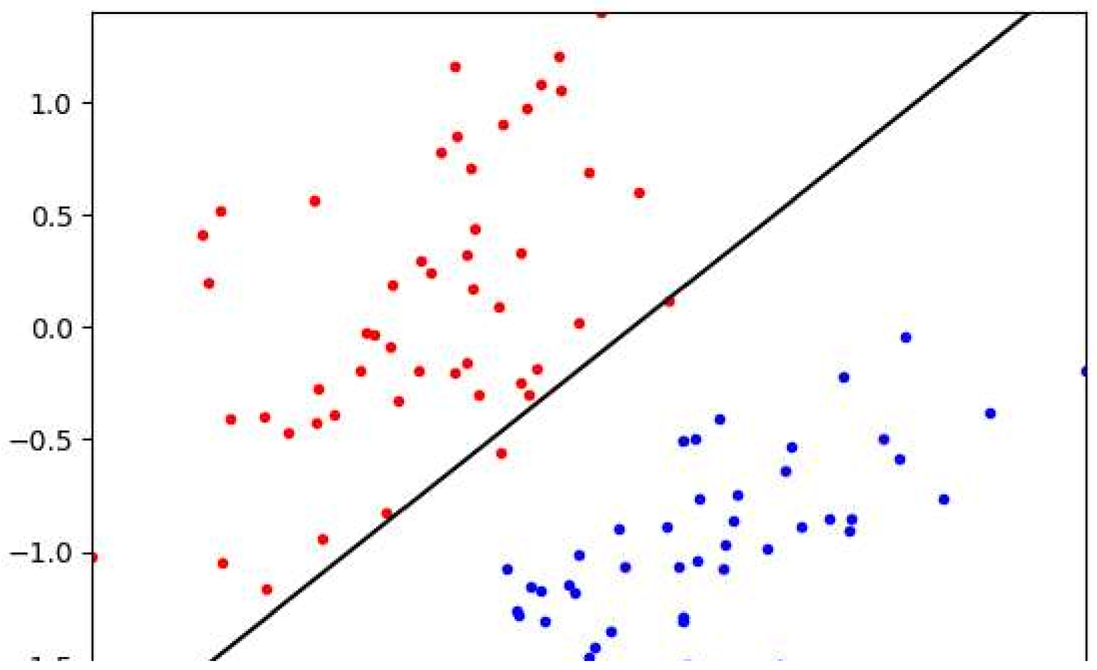
$$\frac{\partial J_j(\theta)}{\partial\theta_0} = -y^{(j)}\left(1-\sigma\left(x^{(j)}\cdot\theta^T\right)\right)x_0^{(j)} - (1-y^{(j)})\left(-\sigma\left(x^{(j)}\cdot\theta^T\right)\right)x_0^{(j)} = \left(\sigma\left(x^{(j)}\cdot\theta^T\right)-y^{(j)}\right)x_0^{(j)}$$

$$\frac{\partial J_j(\theta)}{\partial\theta_1} = \left(\sigma\left(x^{(j)}\cdot\theta^T\right)-y^{(j)}\right)x_1^{(j)}$$

$$\frac{\partial J_j(\theta)}{\partial\theta_2} = \left(\sigma\left(x^{(j)}\cdot\theta^T\right)-y^{(j)}\right)x_2^{(j)}$$

In [ ]:
```
    def train(self, X, Y, initStep=1.0, stopTol=1e-4, stopEpochs=5000, plot=None):
        """ Train the logistic regression using stochastic gradient descent """
        M,N = X.shape;                     # initialize the model if necessary:
        self.classes = np.unique(Y);       # Y may have two classes, any values
        XX = np.hstack((np.ones((M,1)),X)) # XX is X, but with an extra column of ones
        YY = ml.toIndex(Y,self.classes);   # YY is Y, but with canonical values 0 or 1
        if len(self.theta)!=N+1: self.theta=np.random.rand(N+1);
        # init loop variables:
        epoch=0; done=False; Jnll=[]; J01=[];

        while not done:
            stepsize, epoch = initStep*2.0/(2.0+epoch), epoch+1; # update stepsize
            loss = 0
            # Do an SGD pass through the entire data set:
            for i in np.random.permutation(M):
                ri  = np.matmul(XX[i,:],self.theta)     # TODO: compute linear response r(x)
                sigmoid = 1/(1+np.exp(-ri))
                gradi = (sigmoid-YY[i])*(XX[i,:])     # TODO: compute gradient of NLL loss
                self.theta -= stepsize * gradi;  # take a gradient step
                jsur = (-YY[i])*np.log(sigmoid)-(1-YY[i])*np.log(1-sigmoid)
                loss = loss+jsur

            J01.append( self.err(X,Y) )  # evaluate the current error rate

            ## TODO: compute surrogate loss (logistic negative log-likelihood)
            ##  Jsur = sum_i [ (log si) if yi==1 else (log(1-si)) ]
            Jnll.append( loss ) # TODO evaluate the current NLL loss

            plt.figure(1); plt.plot(Jnll,'b-',J01,'r-'); plt.draw();    # plot losses
            if N==2: plt.figure(2); self.plotBoundary(X,Y); plt.draw(); # & predictor if 2D
            plt.pause(.01);                    # let OS draw the plot

            print(self.theta,"=>",Jnll[-1],'/',J01[-1])
            # TODO check stopping criteria: exit if exceeded # of epochs ( > stopEpochs)
            done = epoch > stopEpochs
            #input("continue:")
            if epoch >3:
                done = done or abs(Jnll[-2]-Jnll[-1]) <stopTol
        return Jnll,J01
```

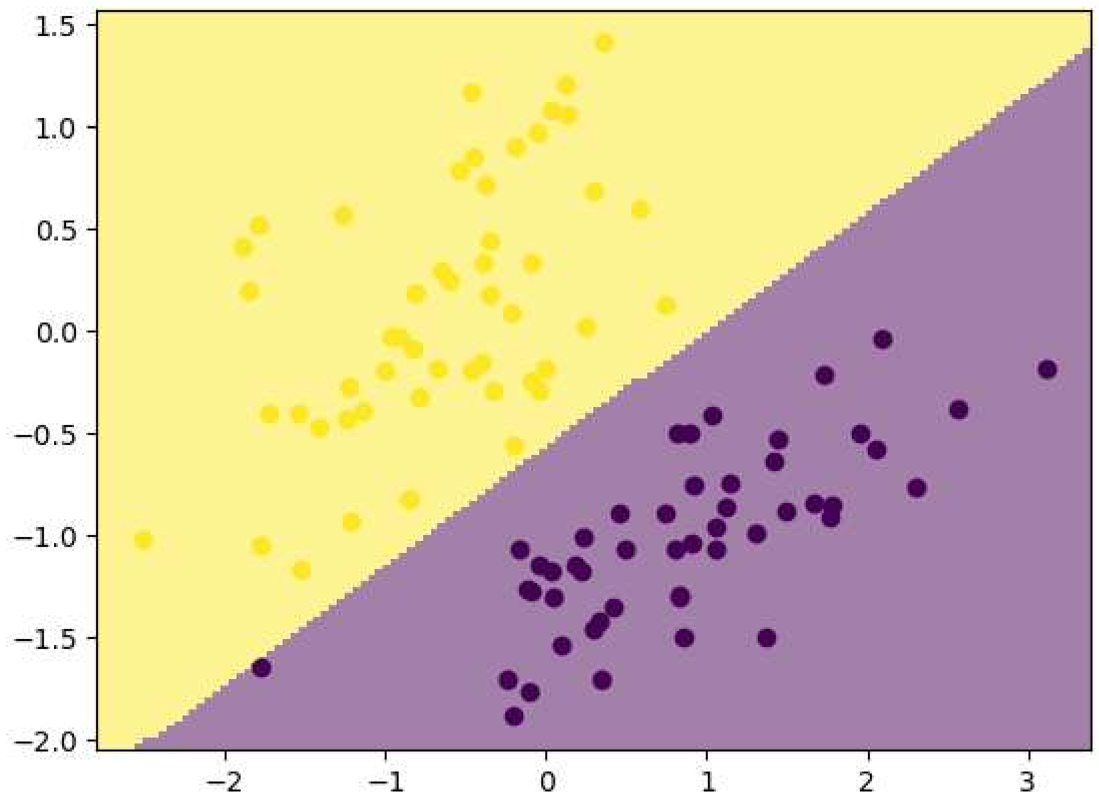
In [20]:
```
learner = logisticClassify2(); # create "blank" learner
learner.classes = np.unique(YA) # define class labels using YA or YB
wts = np.array([0.,-0.,0]); # TODO: fill in values
learner.theta = wts; # set the learner's parameters
(J1,J2) = learner.train(XA,YA,0.5,0.01,50)
plt.show()
```



In [21]:
```
learner.theta
```

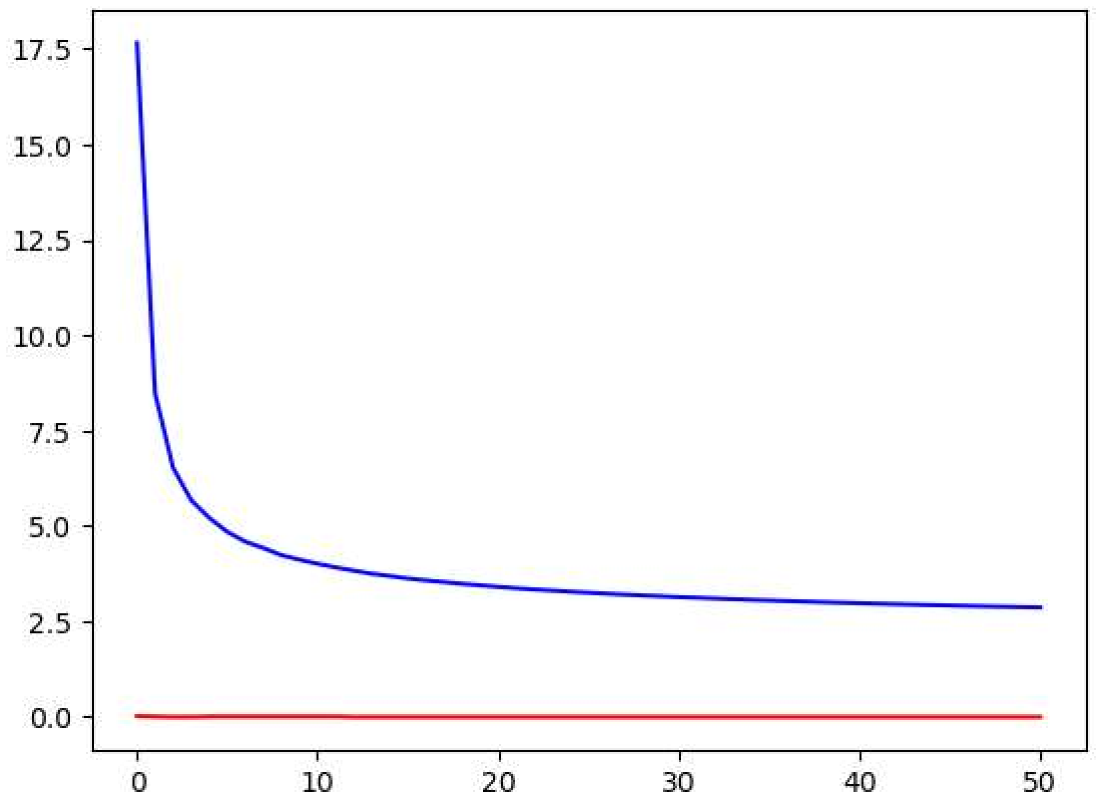
Out[21]:
```
array([ 3.49561704, -3.44760687,  5.9903019 ])
```

In [22]:
```
ml.plotClassify2D(learner, XA, YA)
```



In [11]:
```
plt.plot(J1,"b")
plt.plot(J2,"r")
```

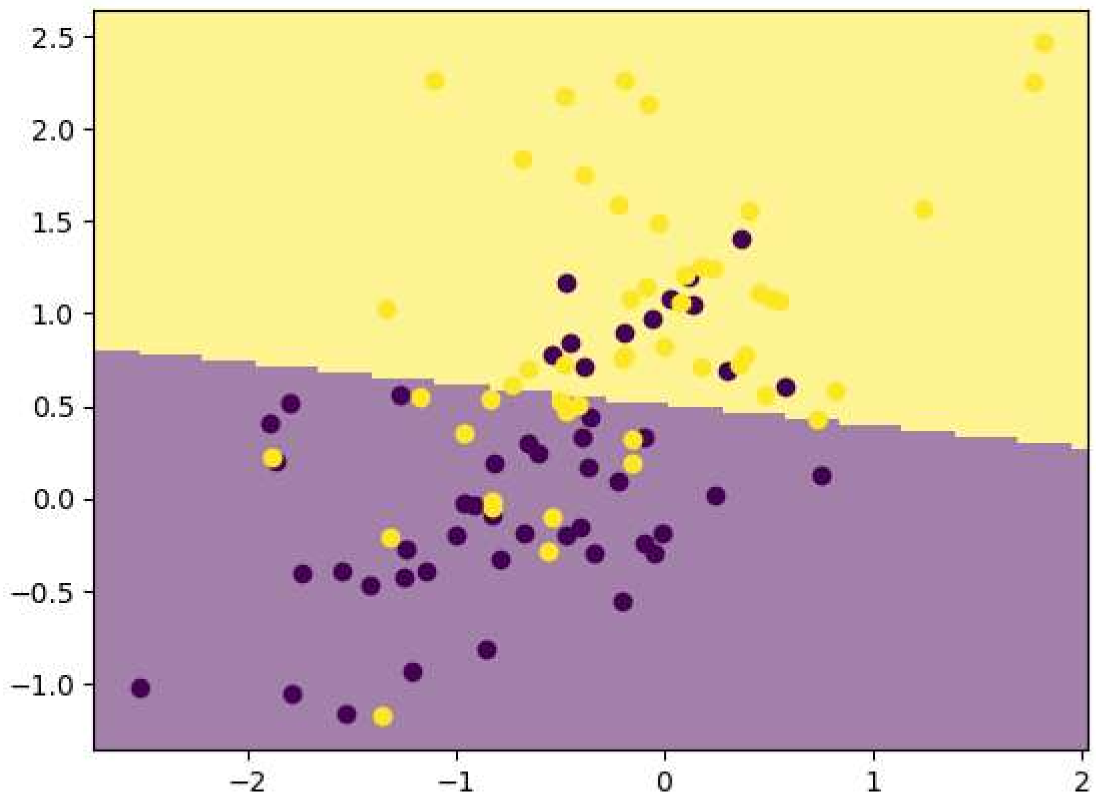
Out[11]:
```
[<matplotlib.lines.Line2D at 0x1ecf89d3c10>]
```



In [11]:
```
learner2 = logisticClassify2(); # create "blank" learner
learner2.classes = np.unique(YB) # define class labels using YA or YB
wts = np.array([0.,-0.,0]); # TODO: fill in values
learner2.theta = wts; # set the learner's parameters
(J21,J22) = learner2.train(XB,YB,0.5,0.01,100)
plt.show()
```

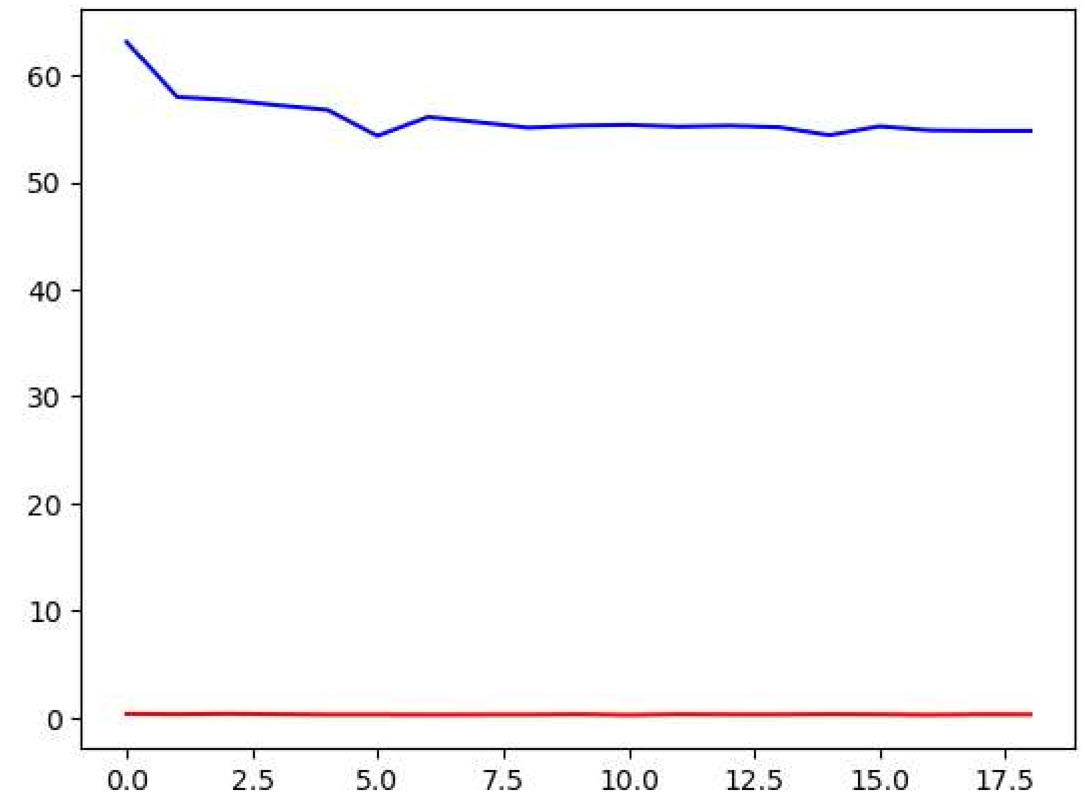
...

In [12]:
```
ml.plotClassify2D(learner2, XB, YB)
```



In [13]:
```
plt.plot(J21,"b")
plt.plot(J22,"r")
print(J22[-1])
```

```
0.2727272727272727
```



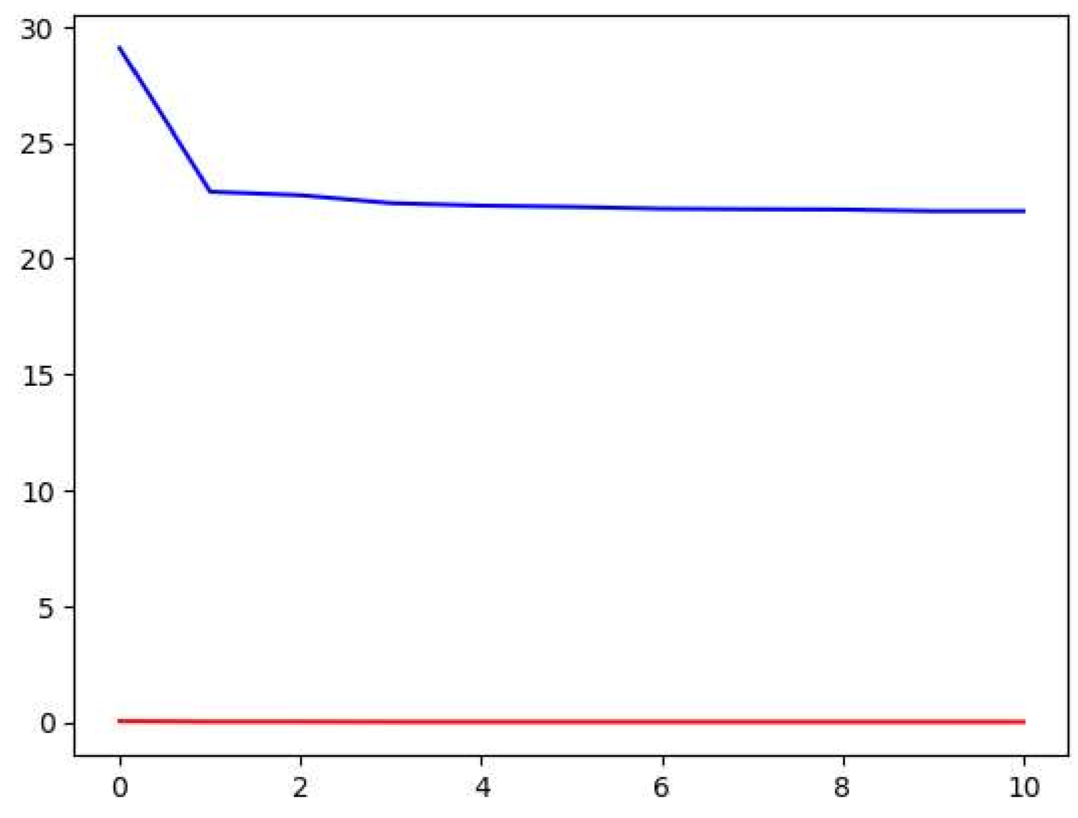
In [ ]:
```
    def train_with_l1(self, X, Y, alpha=2.0,initStep=1.0, stopTol=1e-4, stopEpochs=5000, plot=None):
        """ Train the logistic regression using stochastic gradient descent """
        M,N = X.shape;                     # initialize the model if necessary:
        self.classes = np.unique(Y);       # Y may have two classes, any values
        XX = np.hstack((np.ones((M,1)),X)) # XX is X, but with an extra column of ones
        YY = ml.toIndex(Y,self.classes);   # YY is Y, but with canonical values 0 or 1
        if len(self.theta)!=N+1: self.theta=np.random.rand(N+1);
        # init loop variables:
        epoch=0; done=False; Jnll=[]; J01=[];

        while not done:
            stepsize, epoch = initStep*2.0/(2.0+epoch), epoch+1; # update stepsize
            loss = 0
            # Do an SGD pass through the entire data set:
            for i in np.random.permutation(M):
                ri  = np.matmul(XX[i,:],self.theta)     # TODO: compute linear response r(x)
                sigmoid = 1/(1+np.exp(-ri))
                gradi = (sigmoid-YY[i])*(XX[i,:])+alpha*np.sign(self.theta)/M   # TODO: compute gradient of NLL loss
                self.theta -= stepsize * gradi;  # take a gradient step
                jsur = (-YY[i])*np.log(sigmoid)-(1-YY[i])*np.log(1-sigmoid)+alpha*np.sum(np.abs(self.theta))/M
                loss = loss+jsur

            J01.append( self.err(X,Y) )  # evaluate the current error rate

            ## TODO: compute surrogate loss (logistic negative log-likelihood)
            ##  Jsur = sum_i [ (log si) if yi==1 else (log(1-si)) ]
            Jnll.append( loss ) # TODO evaluate the current NLL loss

            plt.figure(1); plt.plot(Jnll,'b-',J01,'r-'); plt.draw();    # plot losses
            if N==2: plt.figure(2); self.plotBoundary(X,Y); plt.draw(); # & predictor if 2D
            plt.pause(.01);                    # let OS draw the plot

            print(self.theta,"=>",Jnll[-1],'/',J01[-1])
            # TODO check stopping criteria: exit if exceeded # of epochs ( > stopEpochs)
            done = epoch > stopEpochs
            #input("continue:")
            if epoch >3:
                done = done or abs(Jnll[-2]-Jnll[-1]) <stopTol
        return Jnll,J01
```

In [12]:
```
learner = logisticClassify2();
learner.classes = np.unique(YA)
wts = np.array([0.,-0.,0]);
learner.theta = wts;
(J1,J2) = learner.train_with_l1(XA,YA,2,0.5,0.01,50)
plt.show()
```

...

In [15]:
```
plt.plot(J1,"b")
plt.plot(J2,"r")
```

Out[15]:
```
[<matplotlib.lines.Line2D at 0x1ecf66920b0>]
```



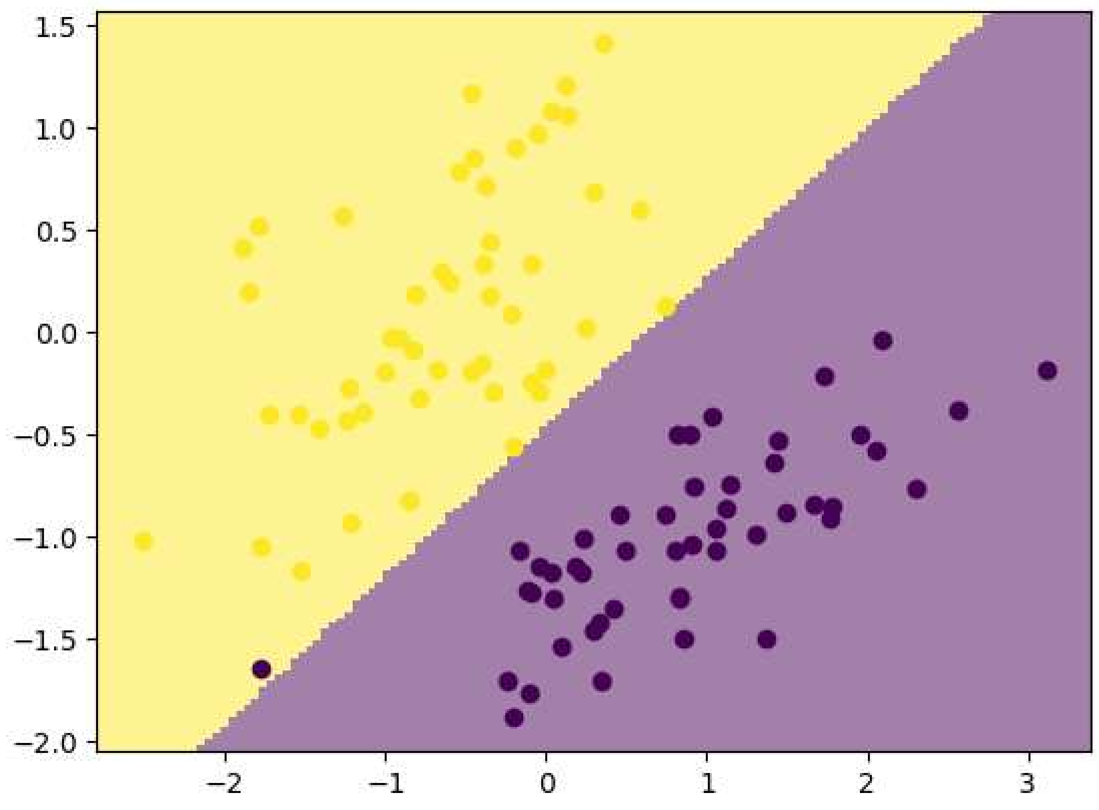
In [13]:
```
learner.theta
```

Out[13]:
```
array([ 1.36119842, -2.10605746,  2.88380247])
```

Compared with the theta trained without regulation components: array([3.49561704, -3.44760687, 5.9903019]) The absolute value of trained theta with L1 regulation is less And it takes less iterations to converge. But the training error is higher due to the model gets less "complicated" because it was "punished" by regulation components.

In [14]:
```
ml.plotClassify2D(learner, XA, YA)
```

```
c:\mcsuci\273P\homework\Assignment 3\Assignment 3\mltools\plot.py:61: UserWarning: color is redundantly defined by the 'color'
keyword argument and the fmt string "ko" (-> color='k'). The keyword argument will take precedence.
  axis.plot( X[Y==c,0],X[Y==c,1], 'ko', color=cmap(cvals[i]), **kwargs )
```



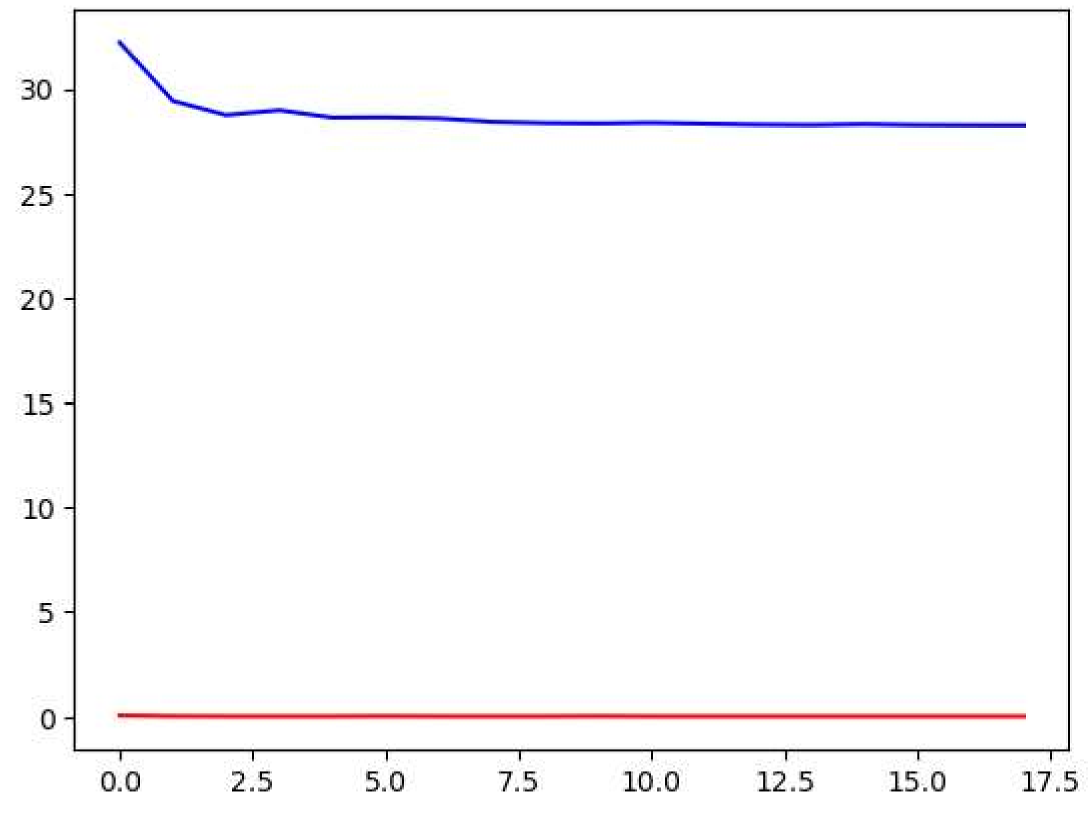
In [ ]:
```
    def train_with_l2(self, X, Y, alpha=2.0,initStep=1.0, stopTol=1e-4, stopEpochs=5000, plot=None):
        """ Train the logistic regression using stochastic gradient descent """
        M,N = X.shape;                     # initialize the model if necessary:
        self.classes = np.unique(Y);       # Y may have two classes, any values
        XX = np.hstack((np.ones((M,1)),X)) # XX is X, but with an extra column of ones
        YY = ml.toIndex(Y,self.classes);   # YY is Y, but with canonical values 0 or 1
        if len(self.theta)!=N+1: self.theta=np.random.rand(N+1);
        # init loop variables:
        epoch=0; done=False; Jnll=[]; J01=[];

        while not done:
            stepsize, epoch = initStep*2.0/(2.0+epoch), epoch+1; # update stepsize
            loss = 0
            # Do an SGD pass through the entire data set:
            for i in np.random.permutation(M):
                ri  = np.matmul(XX[i,:],self.theta)      # TODO: compute linear response r(x)
                sigmoid = 1/(1+np.exp(-ri))
                gradi = (sigmoid-YY[i])*(XX[i,:])+2*alpha*self.theta/M   # TODO: compute gradient of NLL loss
                self.theta -= stepsize * gradi;  # take a gradient step
                jsur = (-YY[i])*np.log(sigmoid)-(1-YY[i])*np.log(1-sigmoid)+alpha*np.sum(np.square(self.theta))/M
                loss = loss+jsur

            J01.append( self.err(X,Y) )  # evaluate the current error rate

            ## TODO: compute surrogate loss (logistic negative log-likelihood)
            ##  Jsur = sum_i [ (log si) if yi==1 else (log(1-si)) ]
            Jnll.append( loss ) # TODO evaluate the current NLL loss

            plt.figure(1); plt.plot(Jnll,'b-',J01,'r-'); plt.draw();    # plot losses
            if N==2: plt.figure(2); self.plotBoundary(X,Y); plt.draw(); # & predictor if 2D
            plt.pause(.01);                    # let OS draw the plot

            print(self.theta,"=>",Jnll[-1],'/',J01[-1])
            # TODO check stopping criteria: exit if exceeded # of epochs ( > stopEpochs)
            done = epoch > stopEpochs
            #input("continue:")
            if epoch >3:
                done = done or abs(Jnll[-2]-Jnll[-1]) <stopTol
        return Jnll,J01
```

In [23]:
```
learner = logisticClassify2();
learner.classes = np.unique(YA)
wts = np.array([0.,-0.,0]);
learner.theta = wts;
(J1,J2) = learner.train_with_l2(XA,YA,2,0.5,0.01,50)
plt.show()
```

...

In [27]:
```
plt.plot(J1,"b")
plt.plot(J2,"r")
```

Out[27]:
```
[<matplotlib.lines.Line2D at 0x1ecf9ee6410>]
```

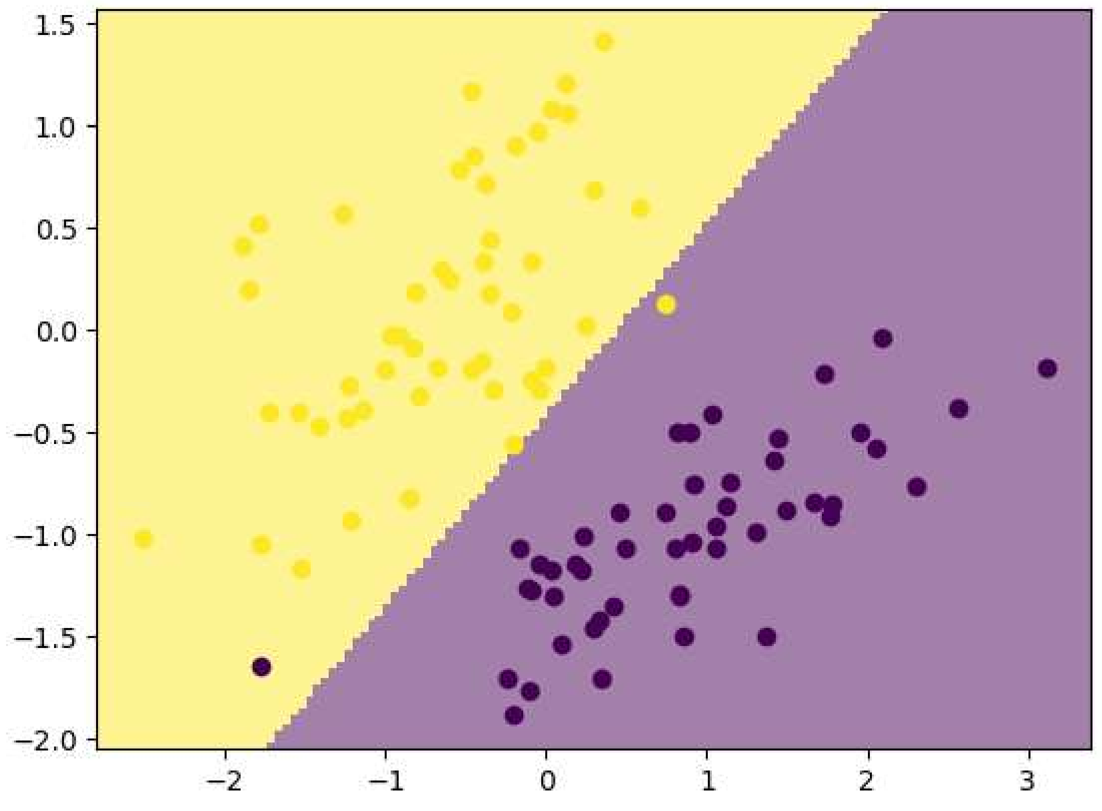


In [28]:
```
learner.theta
```

Out[28]:
```
array([ 0.66986963, -1.46509372,  1.56546168])
```

Compared with the theta trained without regulation components: array([3.49561704, -3.44760687, 5.9903019]) The theta trained with L1 regulation array([1.36119842, -2.10605746, 2.88380247]) The absolute value of trained theta with L2 regulation is less And it takes more iterations than trainer with L1 regulation to converge, but much less iterations than that without regulation components. But the training error is much higher due to the model gets less "complicated" because it was "punished" by regulation components.

In [29]:
```
ml.plotClassify2D(learner, XA, YA)
```



The influence on the gradient is different, the derivative of L1 regulation components equals to const value and reduces to 0 when theta equals to 0, while the derivative of L2 regulation gets smaller when theta reduces. L1 regulation can lead to a sparser weights vector while L2 regulation tends to produce smaller weights. L1 is useful when we want to select unrelated features , as it can force the weights of irrelevant features to become zero. However, since the two features in our dataset are equally important, L1 regularization is not suitable for this model.

Statement of Collaboration：
I write my homework by myself.
Ran Ran

```
In [ ]:
```