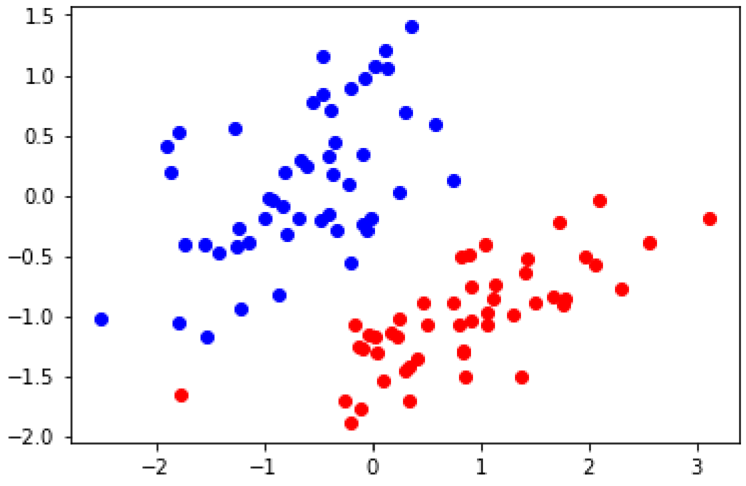
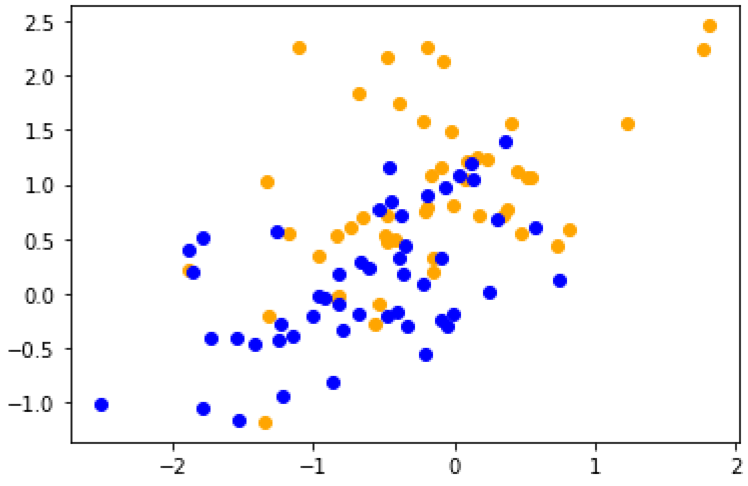
In [1]:
```
import numpy as np
import mltools as ml
from logisticClassify2 import *
import matplotlib.pyplot as plt
```

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
```

Q1. scatter plot

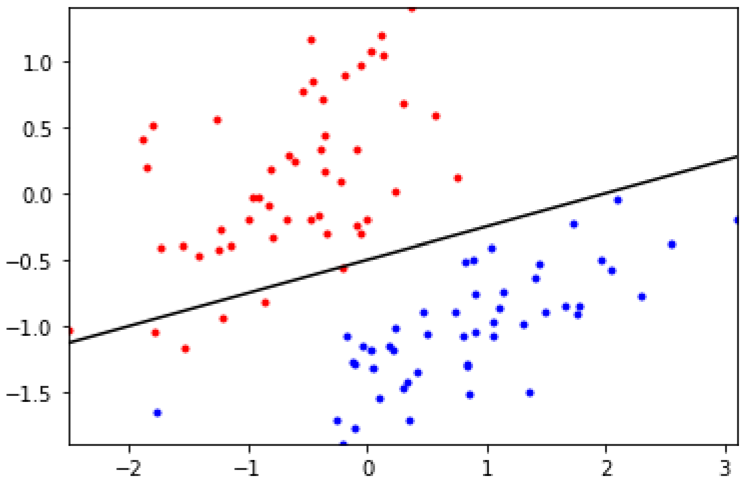
In [3]:
```
# classify btwn class 0&1,  linearly separable
plt.scatter(XA[YA==0,0], XA[YA==0,1],color='red')
plt.scatter(XA[YA==1,0], XA[YA==1,1],color='blue')
plt.show()

#classify btwn class 1&2, not linearly separable
plt.scatter(XB[YB==2,0], XB[YB==2,1],color='orange')
plt.scatter(XB[YB==1,0], XB[YB==1,1],color='blue')
plt.show()
```



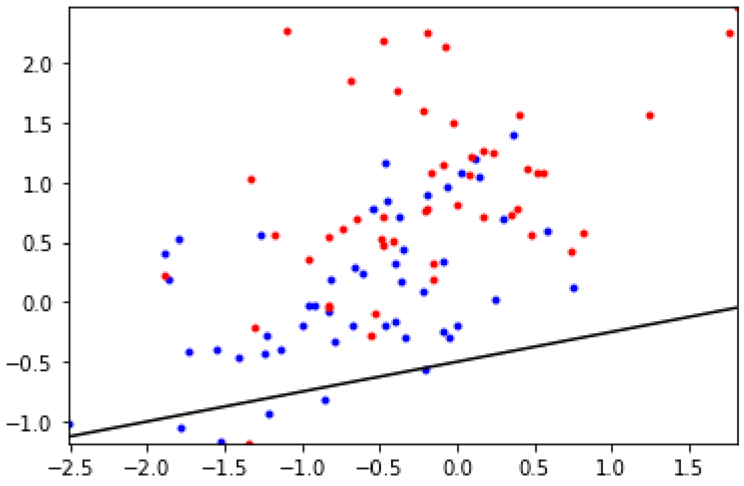


Q2.

In [4]:
```
#Q2
learner = logisticClassify2(); # create "blank" learner
learner.classes = np.unique(YA) # define class labels using YA or YB
theta0,theta1,theta2=np.array([0.5,-0.25,1])
wts = np.array([theta0,theta1,theta2]); # TODO: fill in values
learner.theta = wts; # set the learner’s parameters
learner.plotBoundary(XA, YA)
```



In [5]:
```
learner2 = logisticClassify2(); # create "blank" learner
learner2.classes = np.unique(YB) # define class labels using YA or YB
learner2.theta = wts; # set the learner’s parameters
learner2.plotBoundary(XB, YB)
```



Q3.

In [6]:
```
# learner.classes = [0,1]
learner.err(XA,YA)
```

Out[6]:
```
0.050505050505050504
```

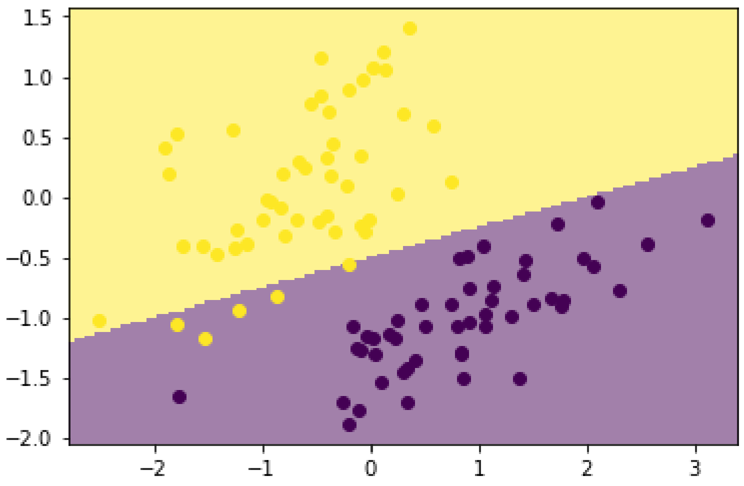
```
In [7]: # learner2.classes = [1,2]
        learner2.err(XB,YB)
```

```
Out[7]: 0.5454545454545454
```

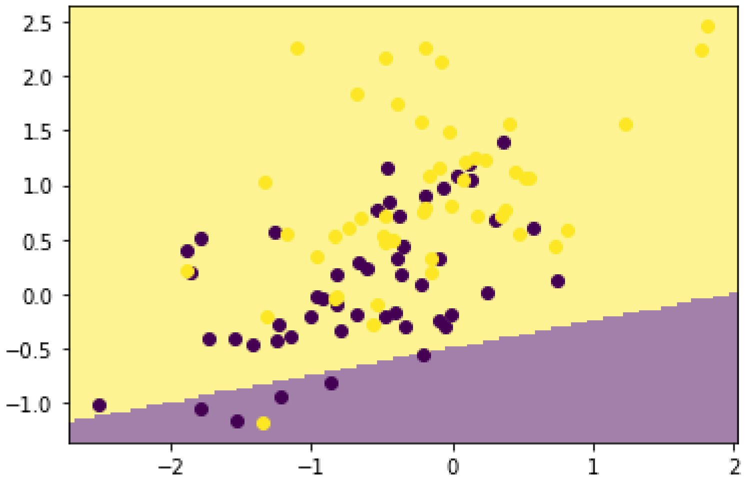
Q4. plotClassify2D

```
In [8]: ml.plotClassify2D(learner, XA, YA)
```

```
D:\UCIcourse\ML\hw3\mltools\plot.py:61: UserWarning: color is redundantly defined by
the 'color' keyword argument and the fmt string "ko" (-> color='k'). The keyword argu
ment will take precedence.
  axis.plot( X[Y==c,0],X[Y==c,1], 'ko', color=cmap(cvals[i]), **kwargs )
```



```
In [9]: ml.plotClassify2D(learner2, XB, YB)
```



Q5.

let $r = \theta_0 + \theta_1 \times x_1 + \theta_2 \times x_2, k = 0 or 1 or 2$

$$\frac{dJ(\theta)}{d\theta_k} = -y^j \cdot \frac{1}{\sigma(r)} \cdot \frac{d\sigma(r)}{d\theta} - (1-y)\frac{1}{1-\sigma(r)} \cdot \frac{-d\sigma(r)}{d\theta}$$

$$= -(y^j \cdot \frac{1}{\sigma(r)} - (1-y^j) \cdot \frac{1}{1-\sigma(r)}) \cdot \frac{d\sigma(r)}{d\theta}$$

$$= -(y^j \cdot \frac{1}{\sigma(r)} - (1-y^j) \cdot \frac{1}{1-\sigma(r)}) \cdot \sigma(r) \cdot (1-\sigma(r)) \cdot \frac{dr}{d\theta}$$

$$= -(y^j \cdot (1-\sigma(r)) - (1-y^j)\sigma(r)) \cdot \frac{dr}{d\theta}$$

$$= -(y^j - y^j\sigma(r) - \sigma(r) + y^j\sigma(r))x_k = -(y^j - \sigma(r))x_k = (\hat{y} - y)x_k$$

Hence,

$$\frac{dJ}{d\theta_0} = (\hat{y} - y) \cdot x_0, \frac{dJ}{d\theta_1} = (\hat{y} - y) \cdot x_1, \frac{dJ}{d\theta_2} = (\hat{y} - y) \cdot x_2$$

Q6. Train

```
def sigmoid(self, x):
    return 1 / (1 + np.exp(-x))

def train(self, X, Y, initStep=1.0, stopTol=1e-4, stopEpochs=5000,
plot=None, regul=None, alpha=0):
    """ 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:
        steosize = initStep
        #stepsize  = initStep*2.0/(2.0+epoch) # update stepsize
        epoch = epoch+1
        # Do an SGD pass through the entire data set:
        for i in np.random.permutation(M):
            ri    = self.sigmoid(self.theta.dot(XX[i]))     # TODO:
compute linear response r(x)
            gradi = (ri-YY[i])*XX[i];     # TODO: compute gradient of
NLL loss
            if regul == 1:
                grad_reg = [0 if self.theta[i]==0 else alpha *
(abs(self.theta[i])/self.theta[i]) for i in range(3)]
                gradi += grad_reg
            elif regul == 2:
                grad_reg = [2*alpha*self.theta[i] for i in range(3)]
                gradi += grad_reg
            self.theta -= stepsize * gradi;  # take a gradient step

        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)) ]
        Jsur = -np.mean([np.log(self.sigmoid(self.theta.dot(XX[i])))
if YY[i]==1 else np.log(1-self.sigmoid(self.theta.dot(XX[i]))) for i
in range(M)])

        if regul == 1:
            Jsur += alpha * sum([abs(self.theta[i]) for i in
range(3)])
        elif regul == 2:
            Jsur += alpha * sum([(self.theta[i])*(self.theta[i]) for
i in range(3)])

        Jnll.append(Jsur) # 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

        ## For debugging: you may want to print current parameters &
losses
        print(self.theta, ' => ', Jnll[-1], ' / ', J01[-1])
        # raw_input()   # pause for keystroke

        # TODO check stopping criteria: exit if exceeded # of epochs
( > stopEpochs)
        # or if Jnll not changing between epochs ( < stopTol )
        #if epoch>1: print(abs(Jnll[-1]-Jnll[-2]))
        done = (epoch>=stopEpochs) or (epoch>1 and abs(Jnll[-1]-
Jnll[-2]) <= stopTol)
```

Q7. Run train

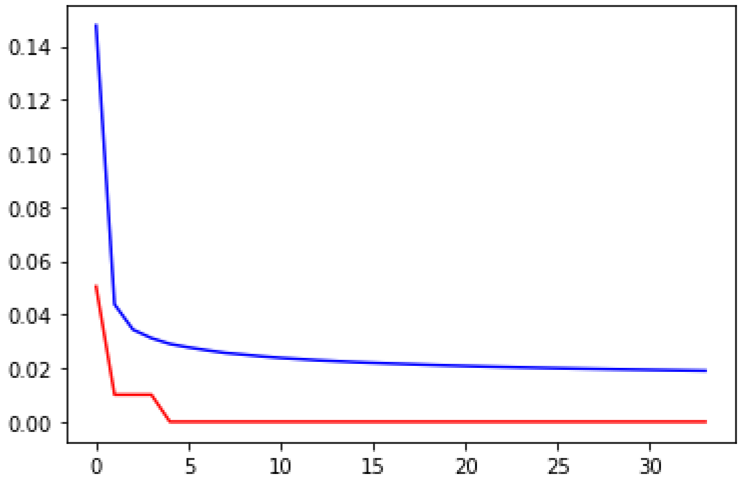
I tried fixed stepsize=1, 0.05 and 0.0005

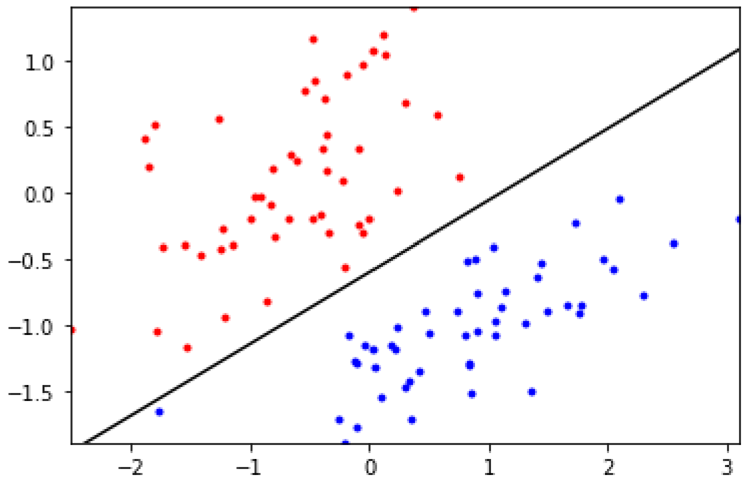
- stepsize=1 works well on first model but failed in second, it seems that the step is too large to find the local minimum.
- stepsize=0.05 works well on both models, even though it's slow on first model.
- stepsize=0.0005 was pretty slow on first model, it took about 1 minute to find the local minimum. However, it also works well on second model.

Then, I tried stepsize = initStep*2.0/(2.0+epoch) but with different initstep.

- initstep = 1 works well and fast on both models.
- initstep = 0.05 also shows good performance, but the speed of training is quite slow because the step size is slower.
- initstep = 0.0005 shows bad performance, it has high nll loss for both model, I guess it was stuck on a small local minimum, and we can avoid it by choosing larger stepsize.

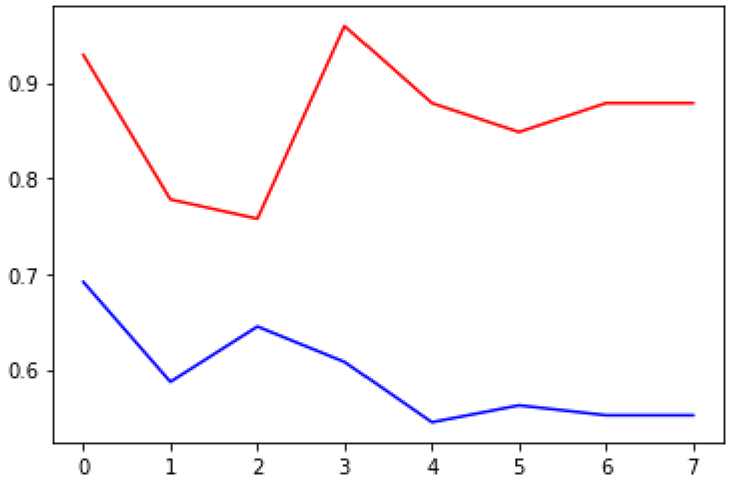
In [10]:
```
learner.train(XA,YA, initStep=1)
```

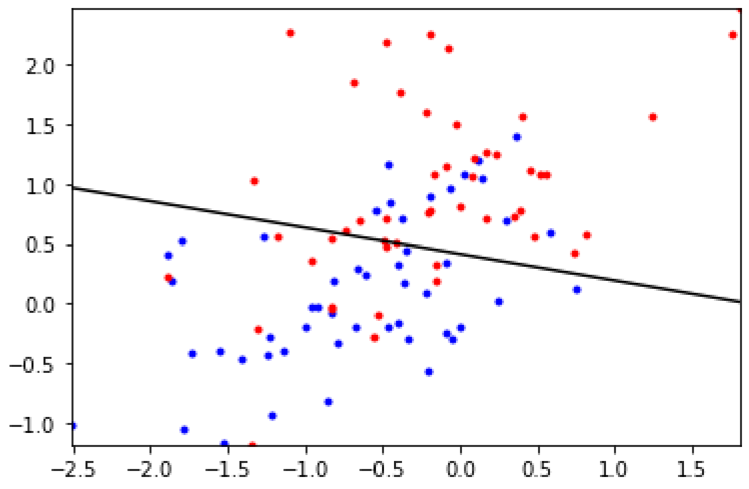




```
[ 4.44194085 -4.00784554  7.39808442]  =>  0.01905581050554889  /  0.0
```

In [11]:
```
learner2.train(XB,YB,initStep=1)
```

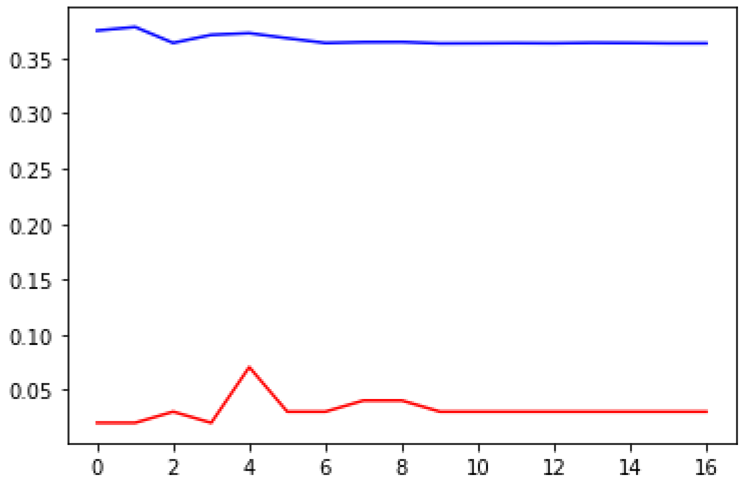


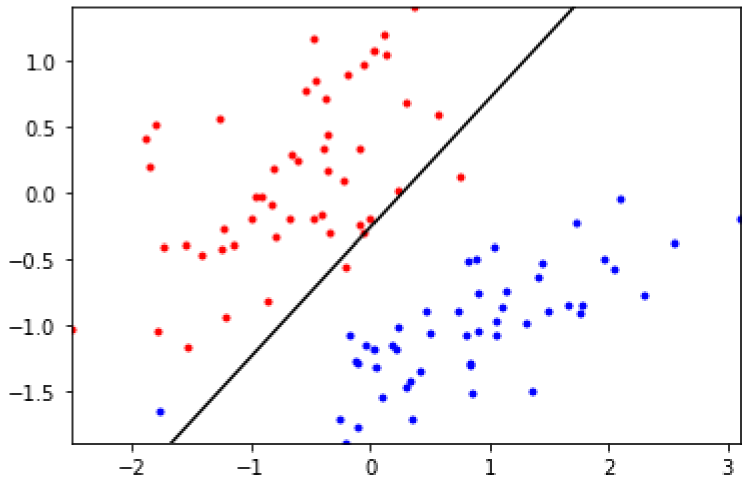


```
[-0.7986327   0.42626238  1.93585395]  =>  0.5519861763388799  /  0.8787878787878788
```

Q8. L1 regularization

In [12]:
```
learner.train(XA,YA,regul=1,alpha=0.05)
```

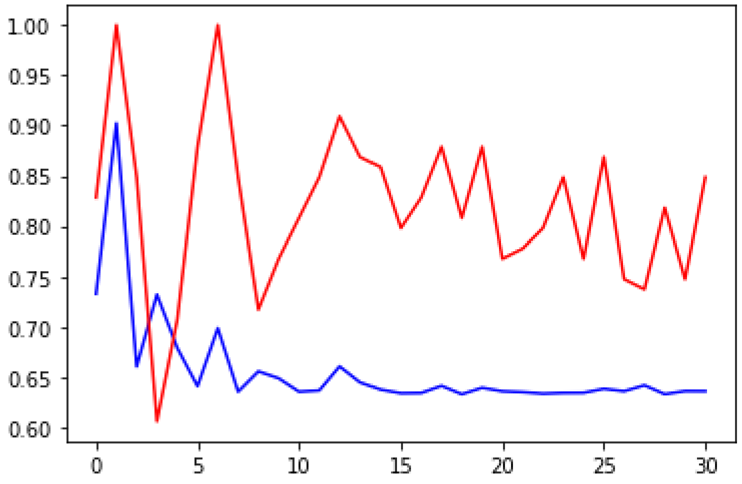


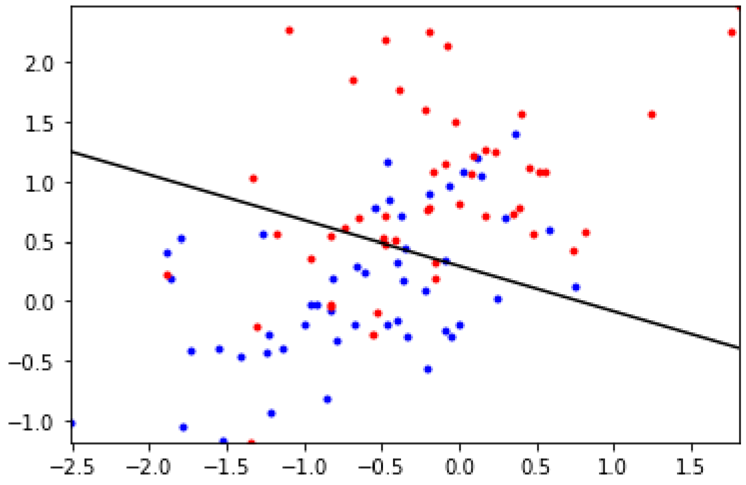


```
[ 0.43671206 -1.6104135   1.65186586]  =>  0.3635289558448035  /  0.030303030303030
4
```

In first trainer, I can see that the value of parameter significantly dropped(from[7.3 -5.7 11.5] to [0.4 -1.5 1.7]) For alpha, I tried 1,0.05 and 0.000005. alpha=1 can't even converge, and when alpha=0.000005, the parameter's value is [4.41535999 -3.98830163 7.36098412], which didn't significantly regularize the parameter.

```
In [13]: learner2.train(XB,YB,regul=1,alpha=0.05)
```





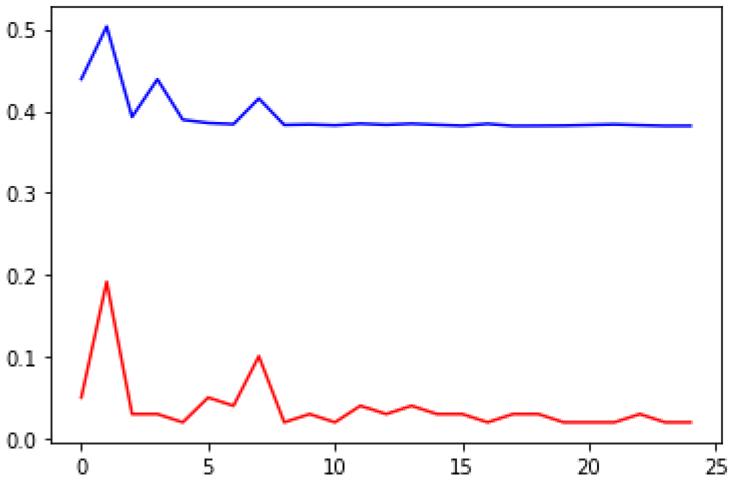
```
[-0.23907709  0.30865906  0.81287604]  =>  0.6360180636548911  /  0.8484848484848485
```

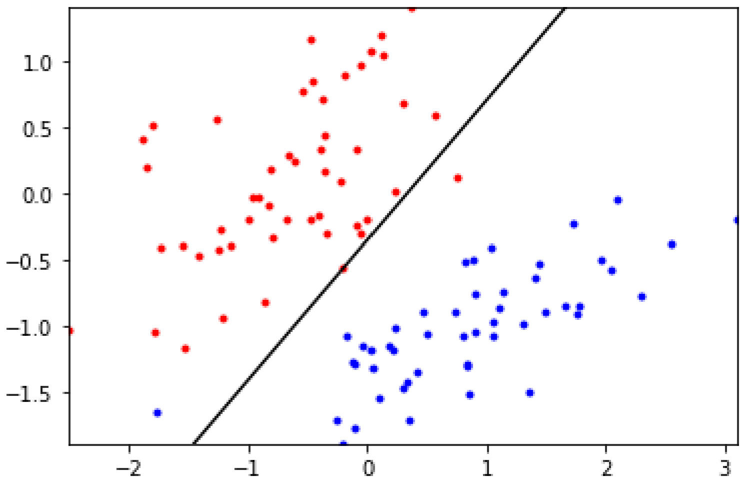
In second trainer, I tried alpha=0.5, 0.05 and 0.00005. alpha=0.5 cannot even converge, while in alpha=0.00005, the parameter's value is [-0.68822824 0.10276988 1.67702896], which also didn't really help. alpha=0.05 shows the best result, it gives us [-0.09707454 0.18329869 0.79438016], and the error rate is also good.

Q9. L2 regularization

Similar result to L1 regularization. The model cannot converge while alpha is too high (ex. alpha=2), but if the alpha is too low, the power of regularizaion is lower, then the parameter becomes higher.

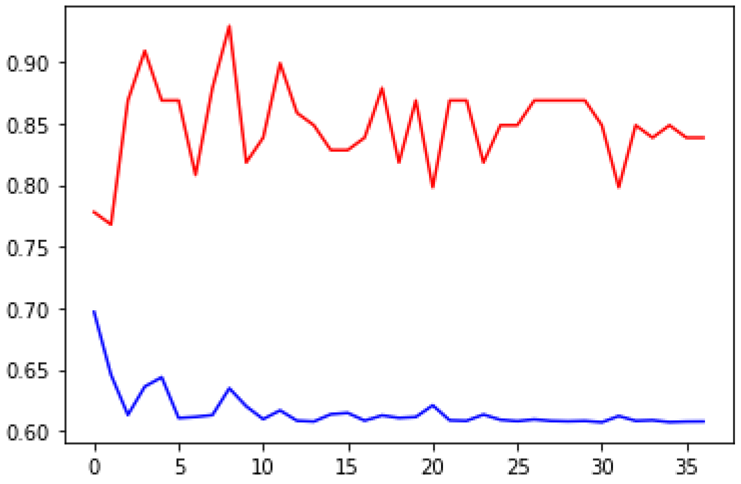
```
In [14]: learner.train(XA,YA,regul=2,alpha=0.05)
```

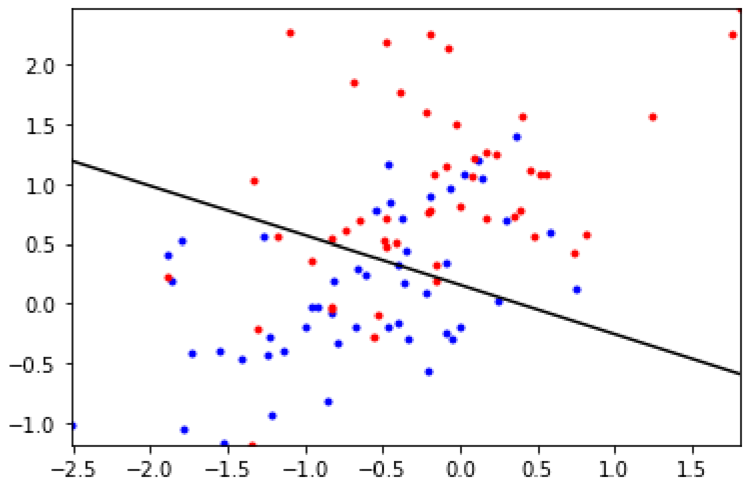




```
[ 0.3603609  -1.07421228  1.01719875]  =>  0.38190044907754783  /  0.020202020202020202
04
```

In [15]:
```
learner2.train(XB,YB,regul=2,alpha=0.05)
```





```
[-0.11247085  0.29621174  0.71852658]  =>  0.6077474908911222  /  0.8383838383838383
```

Q10.

The major differences are

1. L2 norm is square, so it puts more emphasis on high parameters. For example, for learner2, L1 norm gives [-0.14611992 0.2370104 0.86879008] while L2 norm gives [-0.25095201 0.32953775 0.69483565]. we can see that L2 norm punishes more on high parameters(0.86->0.69) while less on low parameters(0.14->0.25, 0.23->0.32)
2. L1 norm can perform feature selection, which mearns it allows some features become 0 while others become larger. However, this property is not useful here because in our case, three features are important.
3. L1 regularization is robust to outliers, L2 regularization is not.

I think first dataset is better in L2 regularization because it can be separated well, and it have no outliers. L2 norm can give better results on regularizing parameters. While the second model is better in L1 regularization because there were some outliers in the dataset.

statement of collaboration

Sabina Yang

- Discussed about regularizations and the gradients of them.