Problem 1: Logistic Regression

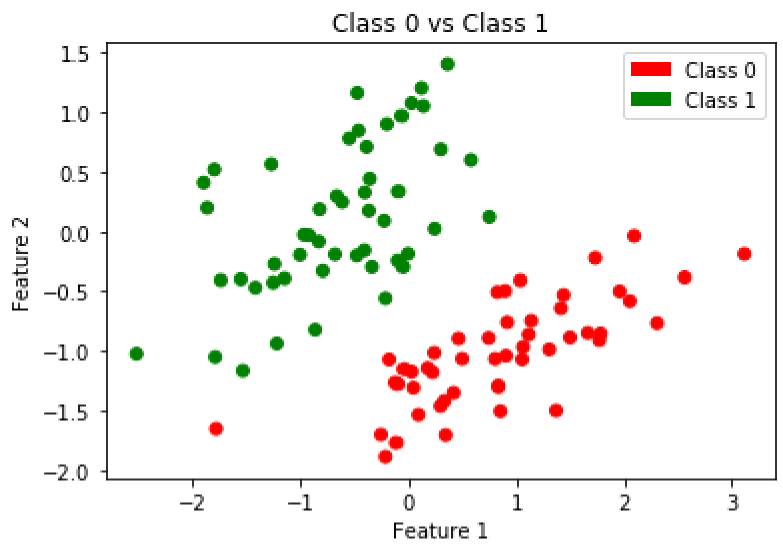
In [1]:

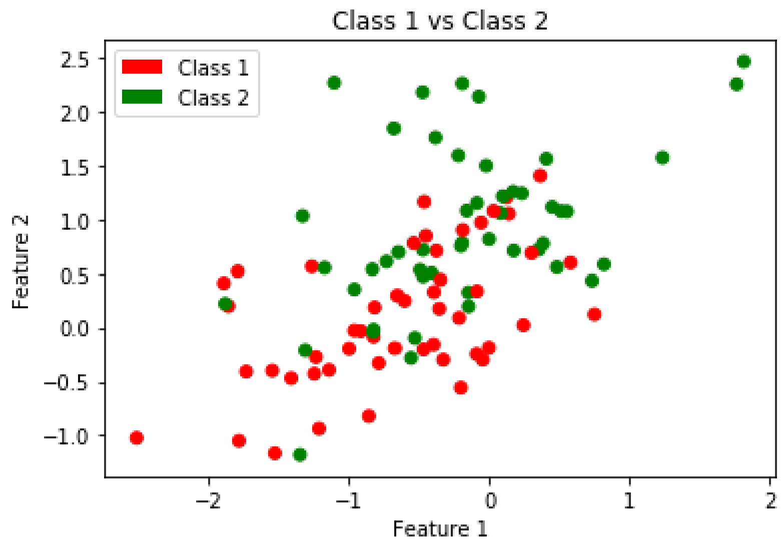
```
import numpy as np
import matplotlib.pyplot as pt
import mltools as ml
import matplotlib.patches as mpatches

%matplotlib inline
iris = np.genfromtxt("data/iris.txt", delimiter=None)
X, Y = iris[:, 0:2], iris[:, -1]
X, Y = ml.shuffleData(X, Y)
X, _ = ml.rescale(X)
XA, YA = X[Y < 2, :], Y[Y < 2]
XB, YB = X[Y > 0, :], Y[Y > 0]
```

1. Scatter plots

In [2]:

```
labels = ['Class 0' if y == 0 else 'Class 1' for y in YA]
colors = ['r' if y == 0 else 'g' for y in YA]
pt.scatter(XA[:, 0], XA[:, 1], color=colors, label=["Red", "Green"])
pt.title("Class 0 vs Class 1")
pt.xlabel("Feature 1")
pt.ylabel("Feature 2")
recs = [
    mpatches.Rectangle((0, 0), 1, 1, fc=c)
    for c in ['r', 'g']
]
pt.legend(recs, ["Class 0", "Class 1"])
pt.show()

colors = ['red' if y == 1 else 'green' for y in YB]
labels = ['Class 1' if y == 1 else 'Class 2' for y in YB]
pt.scatter(XB[:, 0], XB[:, 1], color=colors, label=labels)
pt.xlabel("Feature 1")
pt.ylabel("Feature 2")
pt.title("Class 1 vs Class 2")
pt.legend(recs, ["Class 1", "Class 2"])
pt.show()
```

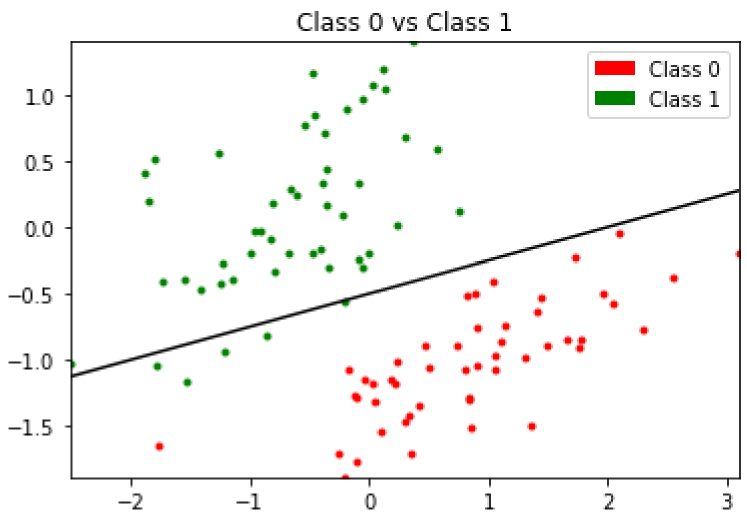


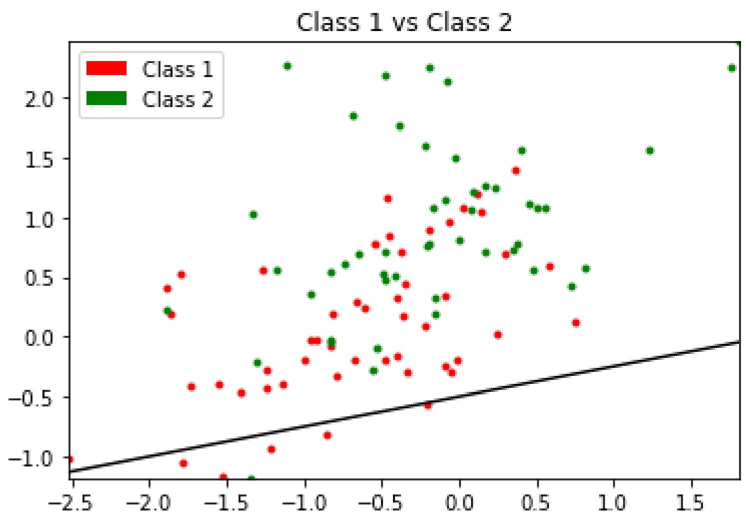


2. Plot Boundary

In [3]:

```
from logisticClassify2 import *

learnerA = logisticClassify2()
learnerA.classes = np.unique(YA)
wts = np.array([.5, -.25, 1])
learnerA.theta = wts

pt.title("Class 0 vs Class 1")
learnerA.plotBoundary(XA, YA)
pt.legend(recs, ["Class 0", "Class 1"])
pt.show()

learnerB = logisticClassify2()
learnerB.classes = np.unique(YB)
learnerB.theta = wts

pt.title("Class 1 vs Class 2")
learnerB.plotBoundary(XB, YB)
pt.legend(recs, ["Class 1", "Class 2"])
pt.show()
```





Plot Boundary Code

```
def plotBoundary(self,X,Y):
    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
    (t0, t1, t2) = self.theta
    x2b = ( -np.array([t0, t0]) - t1 * x1b) / t2
    ## Now plot the data and the resulting boundary:
    A = Y==self.classes[0]; # and plot it:
    plt.plot(X[A,0],X[A,1],'r.',X[~A,0],X[~A,1],'g.',x1b,x2b,'k-'); plt.a
xis(ax); plt.draw();
```

3. Predict

```
def predict(self, X):
    (t0, t1, t2) = self.theta
    g = lambda x: t0 + t1 * x[0] + t2 * x[1]
    return np.array([
        self.classes[1] if g(x) > 0 else self.classes[0]
        for x in X
    ])
```

In [4]:

```
errTrA = learnerA.err(XA, YA)
errTrB = learnerB.err(XB, YB)
print("Learner A Training error: {}".format(errTrA))
print("Learner B Training error: {}".format(errTrB))
```

```
Learner A Training error: 0.0505050505050504
Learner B Training error: 0.46464646464646464
```

From above, it is clear that,

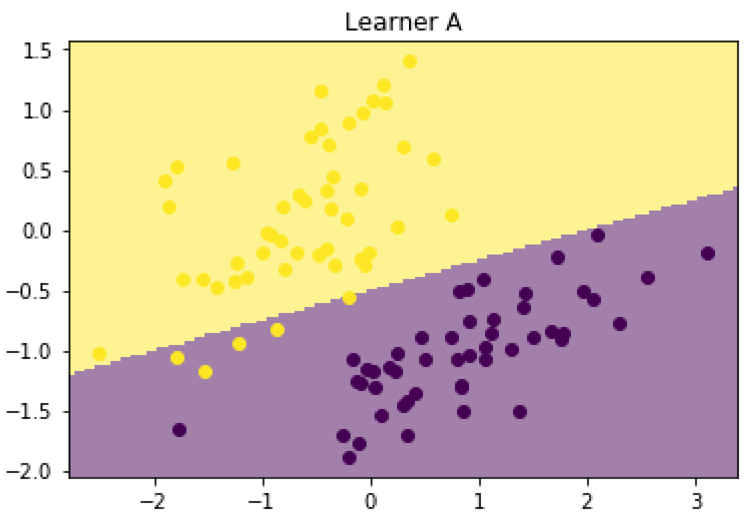
Learner A Training error: ~0.05

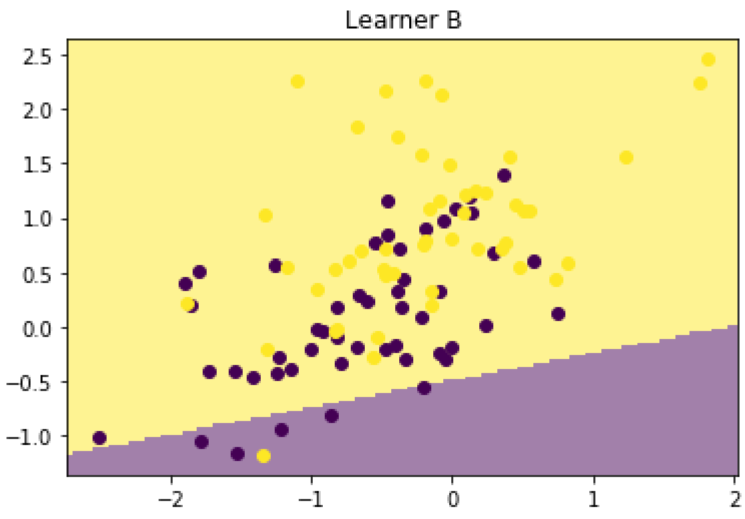
Learner B Training error: ~0.46

4. Plot Classify 2D

In [5]:

```
pt.title("Learner A")
ml.plotClassify2D(learnerA, XA, YA)
pt.show()

pt.title("Learner B")
ml.plotClassify2D(learnerB, XB, YB)
pt.show()
```





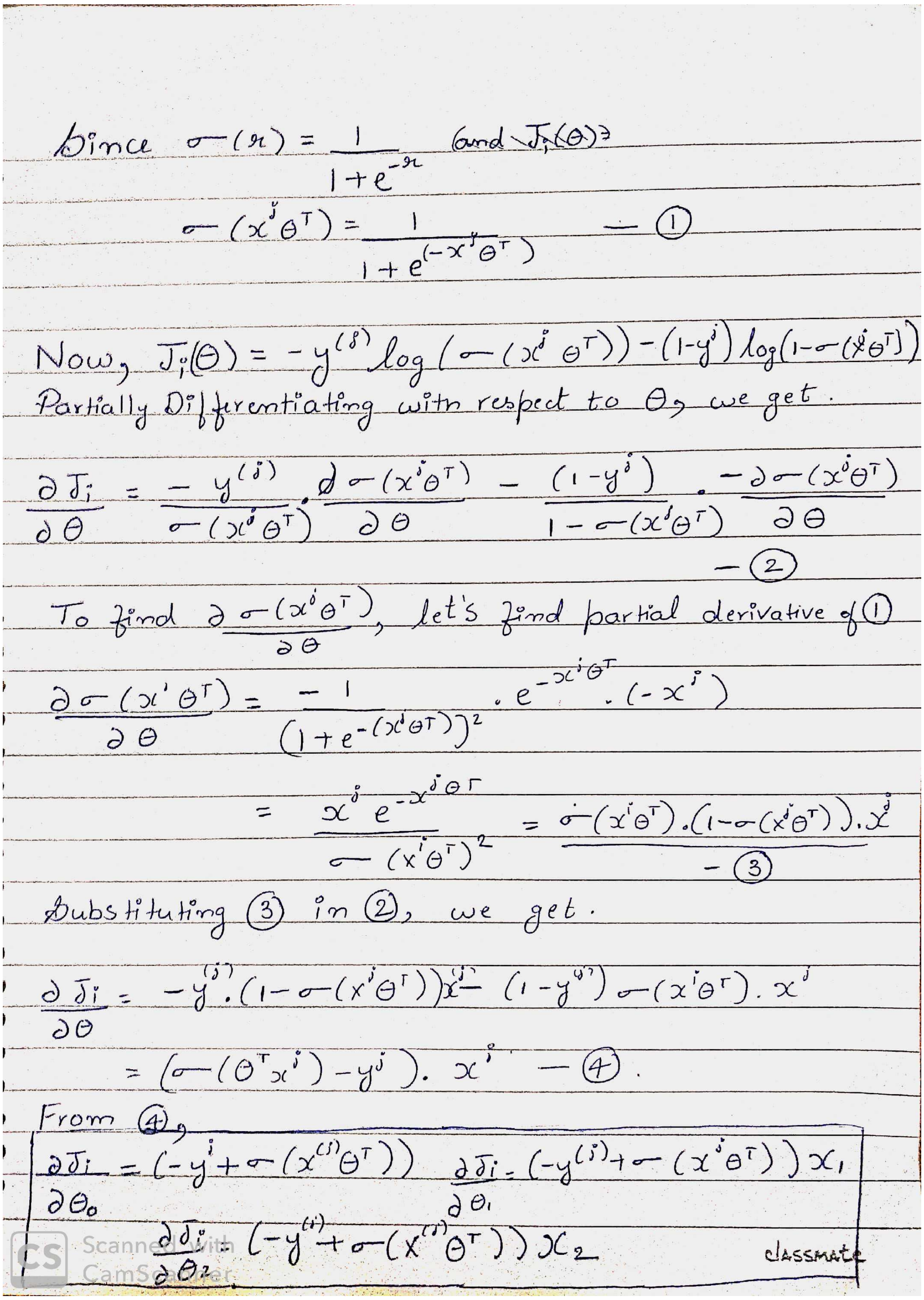
It is clear that this decision boundary matches our analytical decision boundary.

5. Gradient Equation

In [6]:

```python
from IPython.display import Image
Image(filename='q5.jpg')
```

Out[6]:



Since $\sigma(r) = \frac{1}{1+e^{-r}}$ and $J_i(\theta) \ni$

$$\sigma(x^j\theta^T) = \frac{1}{1+e^{(-x^j\theta^T)}} \quad — ①$$

Now, $J_i(\theta) = -y^{(i)}\log(\sigma(x^j\theta^T)) - (1-y^j)\log(1-\sigma(x^j\theta^T))$

Partially Differentiating with respect to θ, we get.

$$\frac{\partial J_i}{\partial\theta} = \frac{-y^{(j)}}{\sigma(x^j\theta^T)}\cdot\frac{\partial\sigma(x^j\theta^T)}{\partial\theta} - \frac{(1-y^j)}{1-\sigma(x^j\theta^T)}\cdot\frac{-\partial\sigma(x^j\theta^T)}{\partial\theta} \quad — ②$$

To find $\frac{\partial\sigma(x^j\theta^T)}{\partial\theta}$, let's find partial derivative of ①

$$\frac{\partial\sigma(x^j\theta^T)}{\partial\theta} = \frac{-1}{(1+e^{-(x^j\theta^T)})^2}\cdot e^{-x^j\theta^T}\cdot(-x^j)$$

$$= \frac{x^j e^{-x^j\theta^T}}{\sigma(x^j\theta^T)^2} = \sigma(x^j\theta^T).(1-\sigma(x^j\theta^T)).x^j \quad — ③$$

Substituting ③ in ②, we get.

$$\frac{\partial J_i}{\partial\theta} = -y^{(j)}.(1-\sigma(x^j\theta^T))x^j - (1-y^{(j)})\sigma(x^j\theta^T).x^j$$

$$= (\sigma(\theta^T x^j) - y^j).\,x^j \quad — ④.$$

From ④,

$$\boxed{\begin{array}{ll} \frac{\partial J_i}{\partial\theta_0} = (-y^j + \sigma(x^{(j)}\theta^T)) & \frac{\partial J_i}{\partial\theta_1} = (-y^{(j)} + \sigma(x^j\theta^T))x_1 \\ \frac{\partial J_i}{\partial\theta_2} = (-y^{(j)} + \sigma(x^{(j)}\theta^T))x_2 & \end{array}}$$

classmate

6. Stochastic Gradient Descent train implementation

Train Code

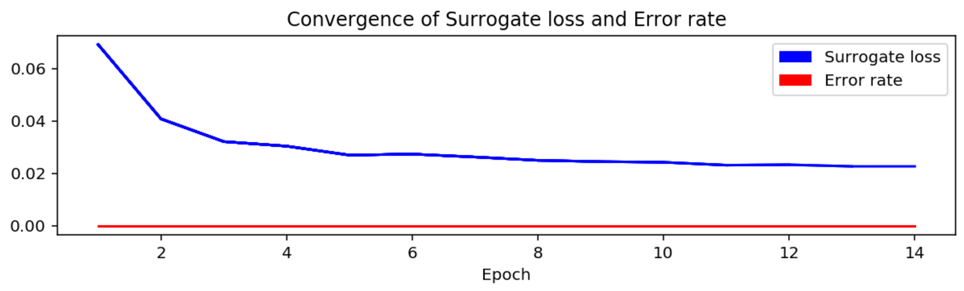
```
def train(self, X, Y, initStep=1.0, stopTol=1e-4, stopEpochs=5000, plot=No
ne):
    M,N = X.shape;                     # initialize the model if necessar
y:
    self.classes = np.unique(Y);       # Y may have two classes, any valu
es
    XX = np.hstack((np.ones((M,1)),X)) # XX is X, but with an extra colum
n of ones
    YY = ml.toIndex(Y,self.classes);   # YY is Y, but with canonical valu
es 0 or 1
    if len(self.theta)!=N+1: self.theta=np.random.rand(N+1);
    sigma = lambda r: 1 / (1+np.exp(-r))
    # init loop variables:
    epoch=0; done=False; Jnll=[float('inf')]; J01=[float('inf')];
    while not done:
        stepsize, epoch = initStep*2.0/(2.0+epoch), epoch+1; # update ste
psize
        Jsurr_i = 0
        # Do an SGD pass through the entire data set:
        for i in np.random.permutation(M):
            ri = np.dot(self.theta, XX[i, :])
            gradi = (-YY[i] + sigma(ri)) * XX[i, :]
            self.theta -= stepsize * gradi;  # take a gradient step
            Jsurr_i += (
                -YY[i] * np.log(sigma(np.dot(self.theta, XX[i, :]))) -
                ((1-YY[i])*np.log(1-sigma(np.dot(self.theta, XX[i, :]))))
            )
        J01.append( self.err(X,Y) )  # evaluate the current error rate
        Jsur = Jsurr_i / M
        Jnll.append(Jsur)
        plt.figure(1); plt.plot(Jnll,'b-',J01,'r-'); plt.draw();    # plo
t losses
        if N==2: plt.figure(2); self.plotBoundary(X,Y); plt.draw(); # & p
redictor if 2D
        plt.pause(.01);                    # let OS draw the plot
        ## For debugging: you may want to print current parameters & loss
es
        # print self.theta, ' => ', Jnll[-1], ' / ', J01[-1]
        # raw_input()   # pause for keystroke
        done = (
            epoch > stopEpochs or
            abs(Jnll[-2] - Jnll[-1]) < stopTol
        )
```

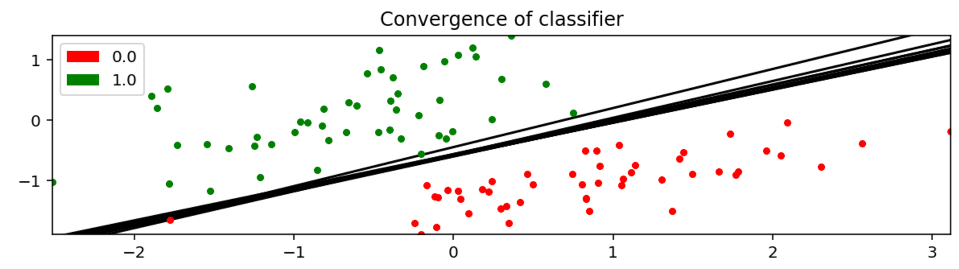
7. Run train

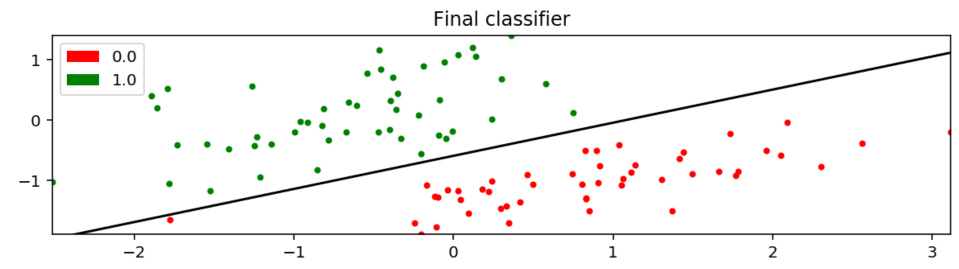
In [9]:

```
%matplotlib nbagg
from logisticClassify2 import *

learnerA = logisticClassify2()
learnerA.theta = np.array([.5, -.25, 1])
learnerA.train(XA, YA, initStep=1)
print("Learner A final theta: {}".format(learnerA.theta))

learnerB = logisticClassify2()
learnerB.theta = np.array([.5, -.25, 1])
learnerB.train(XB, YB, initStep=.1)
print("Learner B final theta: {}".format(learnerB.theta))
```

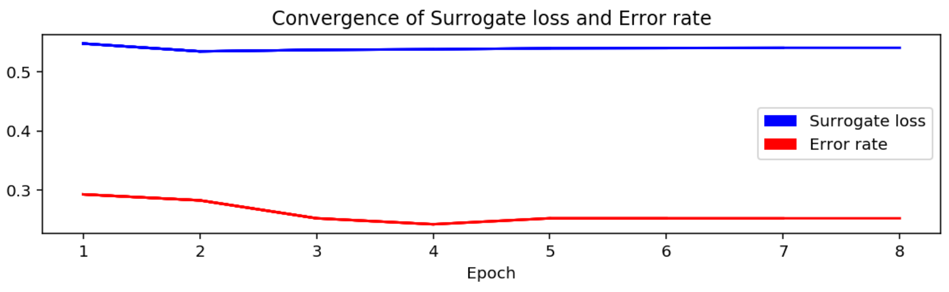
```
Warning: Cannot change to a different GUI toolkit: nbagg. Using note
book instead.
```

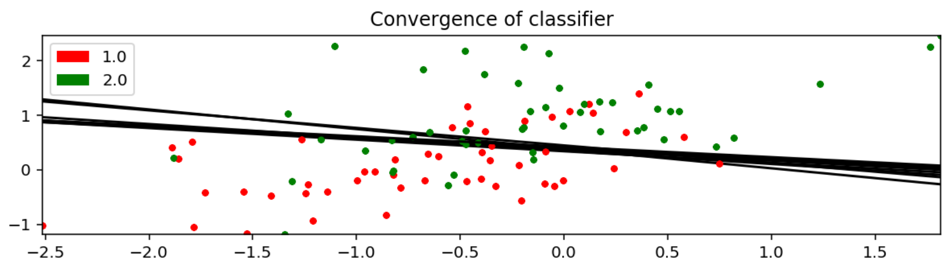


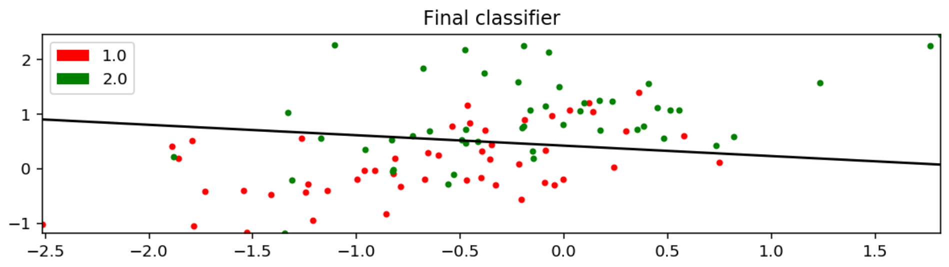




```
Learner A final theta: [ 3.96219722 -3.68191748  6.71565411]
```







```
Learner B final theta: [-0.60565896  0.27306821  1.42875881]
```

There were two parameters that helped the model reaching the optimality. When initial theta was not set, to reach the optimality in time, we needed a bigger step size. This increase in step size resulted in a trade-off by losing some optimality.

Hence, after analytical observations an initial theta value was set. Now, since set A is linearly separable, we could start with a relatively bigger step size. But for set B, we need to have a smaller step size so that it could reach a more optimal solution.

After working with a few values, initial value=1 for A and initial value=.1 for B seemed relatively optimal parameters.

8. L2 Reguralization

Since the reguralization term is, $+\alpha \sum_i \theta_i^2$

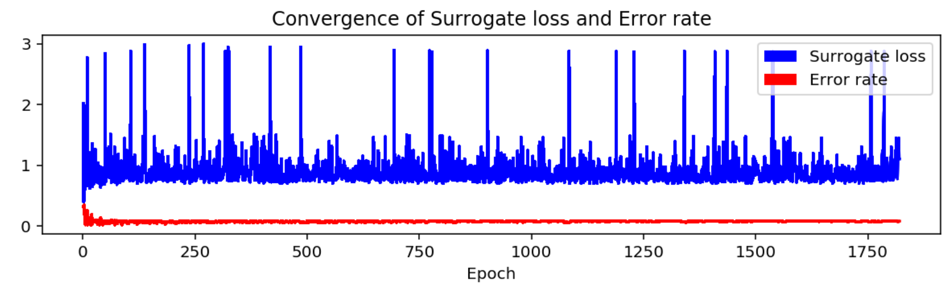
$$\frac{\partial \boldsymbol{J}}{\partial \theta_0} = -y^{(j)} + \sigma(x^{(j)}\theta^{(T)}) + 2\alpha\theta_0$$

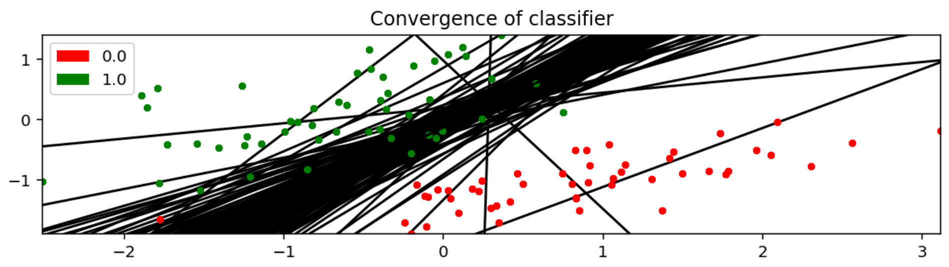
$$\frac{\partial \boldsymbol{J}}{\partial \theta_1} = (-y^{(j)} + \sigma(x^{(j)}\theta^{(T)}))x_1 + 2\alpha\theta_1$$

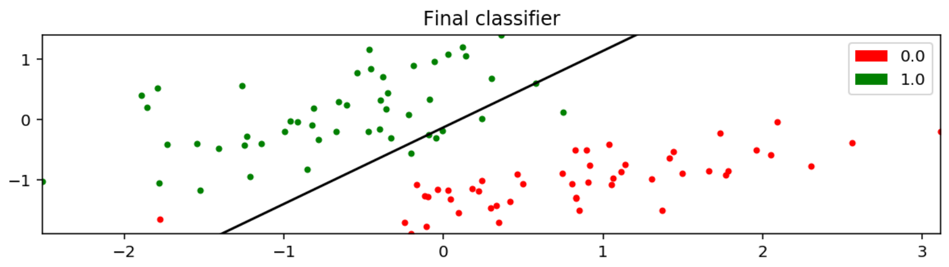
$$\frac{\partial \boldsymbol{J}}{\partial \theta_2} = (-y^{(j)} + \sigma(x^{(j)}\theta^{(T)}))x_2 + 2\alpha\theta_2$$

In [10]:

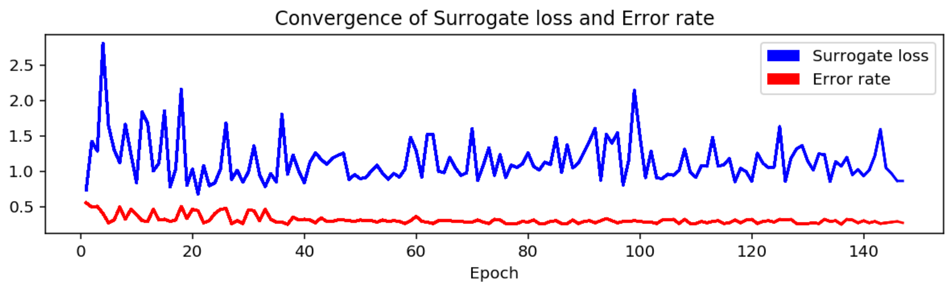
```
from logisticClassify2 import *

learnerA = logisticClassify2()
learnerA.train(XA, YA, alpha=1)
print("Learner A final theta: {}".format(learnerA.theta))

learnerB = logisticClassify2()
learnerB.train(XB, YB, alpha=1)
print("Learner B final theta: {}".format(learnerB.theta))
```

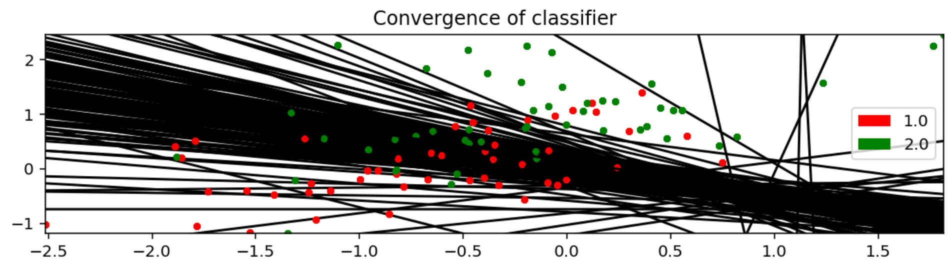


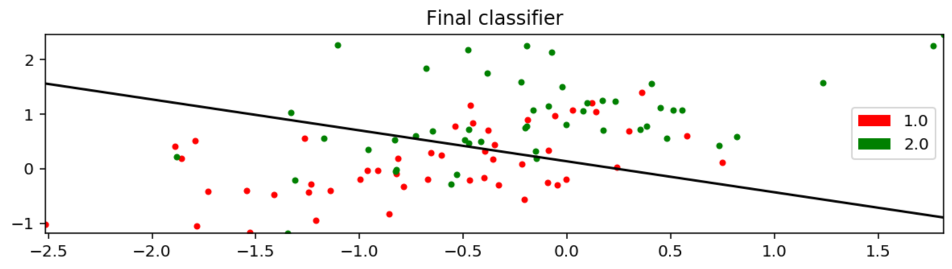




```
Learner A final theta: [ 0.02831366 -0.28236846  0.22281586]
```







```
Learner B final theta: [-0.02275089  0.0942304   0.16615179]
```

In case of set A, there are wrong classifications and in case of set B, it is clear that the convergance was much harder with the new alpha. This can be associated with the added weight to the L2 classifier by setting alpha=1 which resulted in a measure to avoid underfitting that never existed to begin with.

Problem 2

1.

This is a line parallel to the y-axis. It is clear that this can linearly separate a) and b). But, in (c) there is no way a vertical line like this can separate where the points at x1=2 and x1=6 are in the same class but x1=4 is not. In fact, there is no possible 3-set data points whose all-combinations this line can shatter. Hence the VC-Dimension is 2.

2.

This is a linear equation and by theory we know it can shatter d+1 = 3 points. Also looking at this line with variables for it's slope along both the axes, can shatter (a), (b) and (c). But it cannot shatter (d). Consider the points (2, 2) and (8, 6) in the same class while the rest are different. There is no way a straight line could shatter them. In fact, no 4-set data points exist such that this line can completely shatter all of their combinations. Hence, as expected, the VC Dimension is 3.

3.

This is an equation of a circle/elipse. It can shatter (a) and (b) trivially. It can also shatter (c) since the circle/elipse can bend around and between any two data points separating out the remaining one. But, it cannot shatter (d). There exists no possible way to include (2, 2) and (8, 6) but not (6, 4). Similarly, no such 4-set shatterable data points exist. Hence the VC-dimension is 3.

4.

This can easily shatter (a), (b) and (c) since there are two lines just like (2). However, it can also shatter (d). Looking at the points (2, 2) and (8, 6) that model-2 could not shatter, it is clear that this model can. Consider two parallel lines such that points in between them are class-1 and the rest are class-2. We can now consider (2,2) and (8,6) as a single class while the rest are different. Hence the VC-Dimension >=4.

Statement of collaboration

I have not collaborated with anyone for this homework.