Name: Joyce Le
ID#: 82549113
UCI NetID: lejy

Example Template for HW3

This notebook contains the same template code as "logisticClassify2.py", but reorganized to make it simpler to edit and solve in iPython. Feel free to use this for your homework, or do it another way, as you prefer.

In [3]:
```
import numpy as np
import mltools as ml
import matplotlib.pyplot as plt   # use matplotlib for plotting with inline
 plots
%matplotlib inline
plt.set_cmap('jet');
import warnings
warnings.filterwarnings('ignore') # for deprecated matplotlib functions
```

```
<Figure size 432x288 with 0 Axes>
```

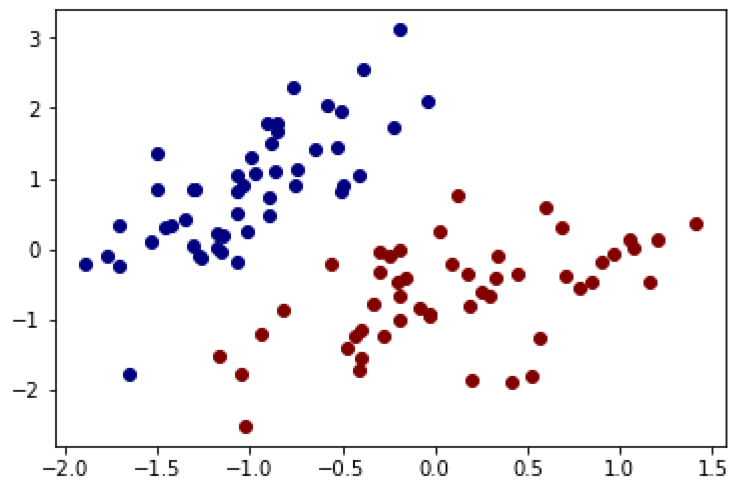
Problem 1

In [55]:
```
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 rather than by class la
bel
X,_  = ml.transforms.rescale(X)  # rescale to improve numerical stability,
 speed convergence

XA, YA = X[Y<2,:], Y[Y<2]        # Dataset A: class 0 vs class 1
XB, YB = X[Y>0,:], Y[Y>0]        # Dataset B: class 1 vs class 2
```

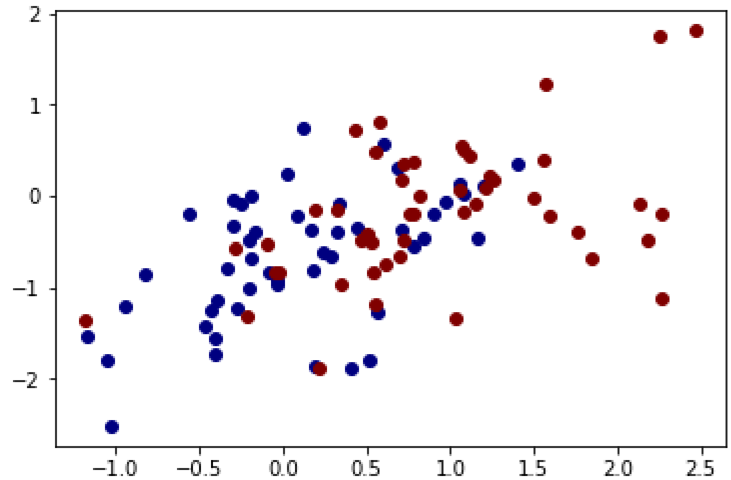
P1.1

In [56]:
```
print("Dataset A:")
ml.plotClassify2D(None, XA, YA)
plt.show()

print("Dataset B:")
ml.plotClassify2D(None, XB, YB)
plt.show()
```

Dataset A:



Dataset B:

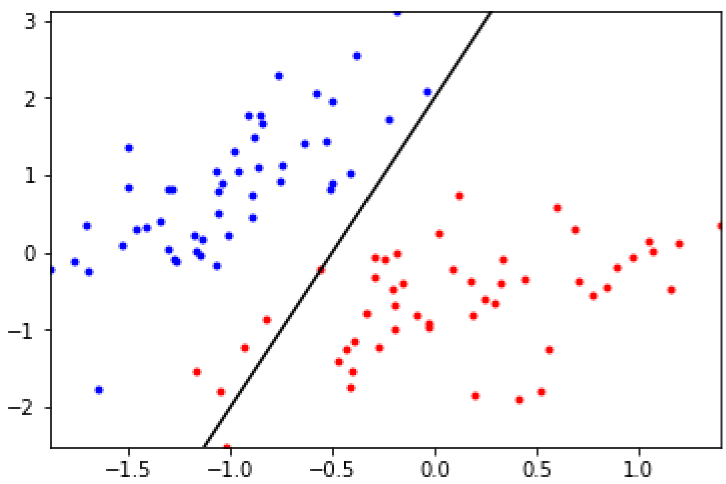


Dataset A is linearly seperable.

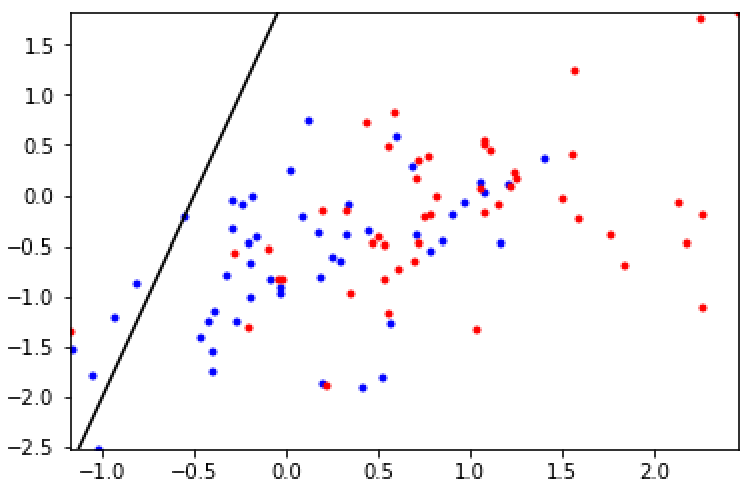
P1.2

In [57]:
```
def myPlotBoundary(self, X,Y):
    """ Plot the (linear) decision boundary of the classifier, along with d
ata """
    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
    # theta0 + theta1 X1 + theta2 X2 = 0
    # --> X2 = -theta0/theta2 - theta1/theta2 X1
    x1b = np.array([ax[0],ax[1]]);  # at X1 = points in x1b
    x2b = -self.theta[0]/self.theta[2] - (self.theta[1]/self.theta[2]) * x1
b
    ## Now plot the data and the resulting boundary:
    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.axi
s(ax); plt.draw();


# Create a shell classifier
class logisticClassify2(ml.classifier):
    classes = []
    theta = np.array( [-1, 0, 0] )   # initialize theta to something
    plotBoundary = myPlotBoundary    #
    predict = None                   # these functions will be implemented
 later
    train = None

learnerA = logisticClassify2()
learnerA.classes = np.unique(YA)       # store the class values for this pr
oblem
learnerA.theta = np.array([0.5, 1., -0.25])  # TODO: insert hard-coded valu
es
learnerA.plotBoundary(XA,YA)

print("Dataset A:")
plt.show()

learnerB = logisticClassify2()
learnerB.classes = np.unique(YB)
learnerB.theta = np.array([0.5, 1., -0.25])
learnerB.plotBoundary(XB,YB)

print("Dataset B:")
plt.show()
```

Dataset A:



Dataset B:



P1.3

In [62]:
```
# Should go in your logistic2 class:
def myPredict(self,X):
    """ Return the predictied class of each data point in X"""
    Yhat = np.zeros(X.shape[0])
    for i in range(X.shape[0]):
        response = self.theta[0] + self.theta[1] * X[i,0] + self.theta[2] *
 X[i,1]
        if response > 0:
            Yhat[i] = self.classes[1]
        else:
            Yhat[i] = self.classes[0]
    return Yhat


# Update our shell classifier definition
class logisticClassify2(ml.classifier):
    classes = []
    theta = np.array( [-1, 0, 0] )   # initialize theta to something
    plotBoundary = myPlotBoundary
    predict = myPredict
    train = None

learnerA = logisticClassify2()
learnerA.classes = np.unique(YA)       # store the class values for this pr
oblem
learnerA.theta = np.array([0.5, 1., -0.25])

learnerB = logisticClassify2()
learnerB.classes = np.unique(YB)       # store the class values for this pr
oblem
learnerB.theta = np.array([0.5, 1., -0.25])

print("Error rate for dataset A: ", learnerA.err(XA,YA))
print("Error rate for dataset B: ", learnerA.err(XB,YB))
```

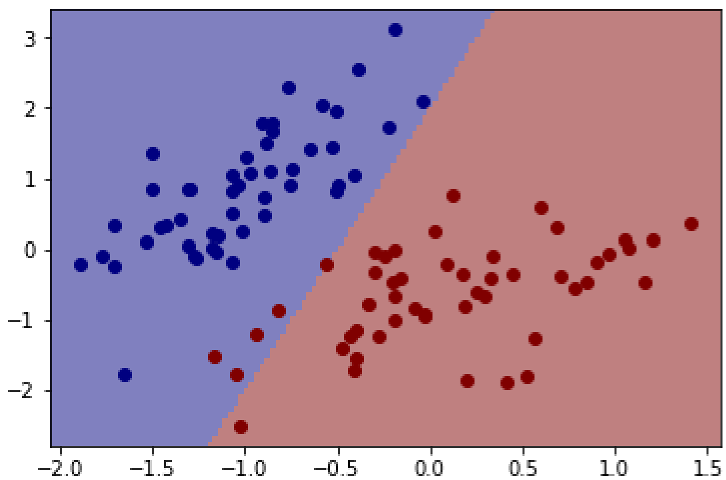
```
Error rate for dataset A:  0.050505050505050504
Error rate for dataset B:  0.5454545454545454
```

P1.4

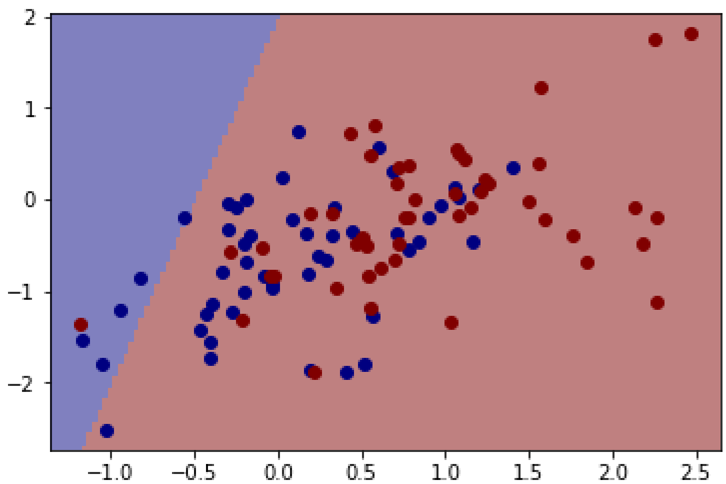
If predict is implemented, then the inherited 2D visualization function should work; you can verify your decision boundary from P1.2:

In [60]:
```
print("Dataset A:")
ml.plotClassify2D(learnerA,XA,YA)
plt.show()

print("Dataset B:")
ml.plotClassify2D(learnerB,XB,YB)
plt.show()
```

```
Dataset A:
```




```
Dataset B:
```



The decision boundary is the same as that plotted in P1.2

P1.5: Gradient of NLL

Our negative log-likelihood loss is:

$$J_j(\theta) = -\begin{cases} \log(\sigma(x^{(j)} \cdot \theta)) & \text{if } y^{(j)} = 1 \\ \log(1 - \sigma(x^{(j)} \cdot \theta)) & \text{if } y^{(j)} = 0 \end{cases}$$

Thus, its gradient is:

$$\nabla J_j(\theta) = \frac{\partial J_j(\theta)}{\partial \theta_j} = \frac{\partial}{\partial \theta_j} \log(\sigma(x^{(j)} \cdot \theta)) \text{ if } y^{(j)} = 1, \ \frac{\partial}{\partial \theta_j} \log(1 - \sigma(x^{(j)} \cdot \theta)) \text{ if } y^{(j)} = 0$$

$$\text{Therefore, } \nabla J_j(\theta) = -\begin{cases} (1 - \sigma(x^{(j)} \cdot \theta))\, x^{(j)} & \text{if } y^{(j)} = 1 \\ -\sigma(x^{(j)} \cdot \theta)\, x^{(j)} & \text{if } y^{(j)} = 0 \end{cases}$$

P1.6

Now define the train function and complete its missing code.

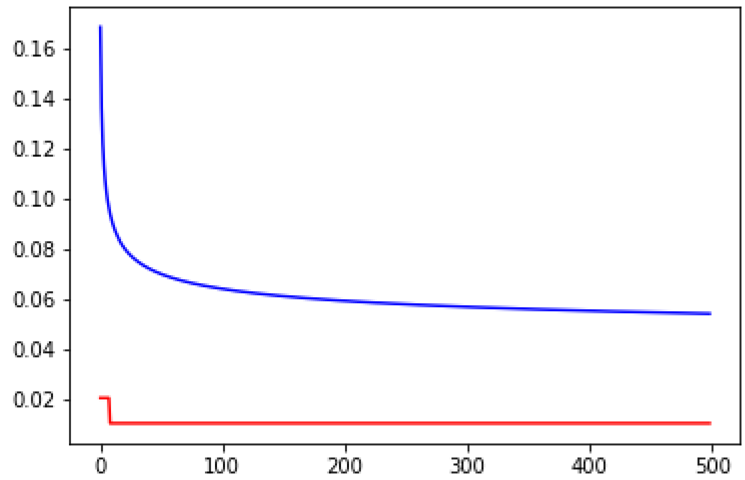
In [63]:

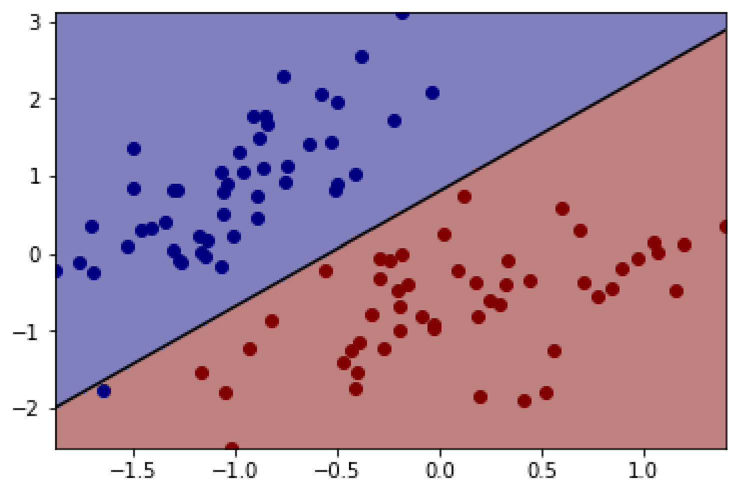
```
def myTrain(self, X, Y, initStep=1.0, stopTol=1e-4, stopEpochs=5000, plot=None):
    """ Train the logistic regression using stochastic gradient descent """
    M,N = X.shape;                     # initialize the model if necessary:
    self.classes = np.unique(Y);       # Y may have two classes, 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
        # Do an SGD pass through the entire data set:
        for i in np.random.permutation(M):
            ri    = XX[i].dot(self.theta)     # TODO: compute linear response r(x)
            sigma = 1. / (1. + np.exp(-ri))
            gradi = -(1-sigma) * XX[i,:] if YY[i] else sigma * XX[i,:]    # TODO: compute gradient of NLL loss
            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)
        ##  Jnll = sum_i [ (log si) if yi==1 else (log(1-si)) ]
        si = 1./(1.+np.exp(-(XX.dot(self.theta))))
        Jsur = -np.mean(YY*np.log(si) + (1-YY)*np.log(1-si))
        Jnll.append(Jsur) # TODO evaluate the current NLL loss

        ## 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 )

        done = (epoch >= stopEpochs) | ((epoch > 1) & (abs(Jnll[epoch-1] - Jnll[epoch-2]) < stopTol))

    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
```

P1.7

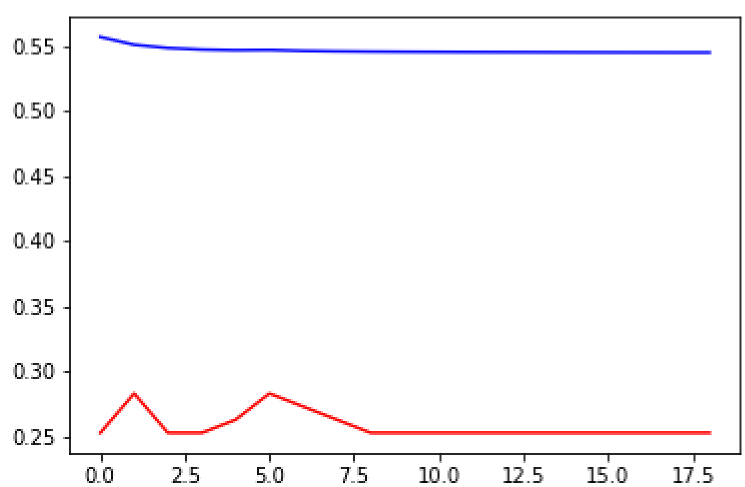
In [64]:
```
# Update our shell classifier definition
class logisticClassify2(ml.classifier):
    classes = []
    theta = np.array( [-1, 0, 0] )   # initialize theta to something
    plotBoundary = myPlotBoundary    #
    predict = myPredict              # Now all parts are implemented
    train = myTrain

print("Dataset A:")
learnerA = logisticClassify2()
learnerA.theta = np.array([0.5, 1., -0.25])
learnerA.train(XA,YA,initStep=1e-1,stopEpochs=1000,stopTol=1e-5);

ml.plotClassify2D(learnerA,XA,YA)
print("Training error rate: ",learnerA.err(XA,YA))

plt.show()

print("Dataset B:")
learnerB = logisticClassify2()
learnerB.theta = np.array([0.5, 1., -0.25])
learnerB.train(XB,YB,initStep=1e-1,stopEpochs=1000,stopTol=1e-5);

ml.plotClassify2D(learnerB,XB,YB)
print("Training error rate: ",learnerA.err(XB,YB))

plt.show()
```

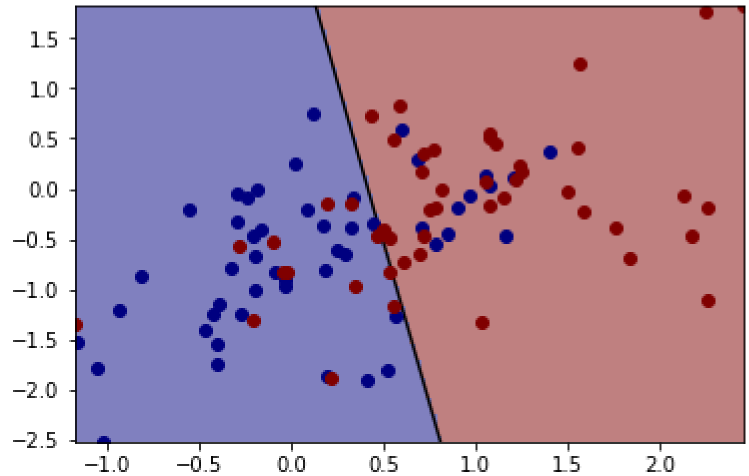
```
Dataset A:
Training error rate:  0.010101010101010102
```






```
Dataset B:
Training error rate:  0.494949494949495
```



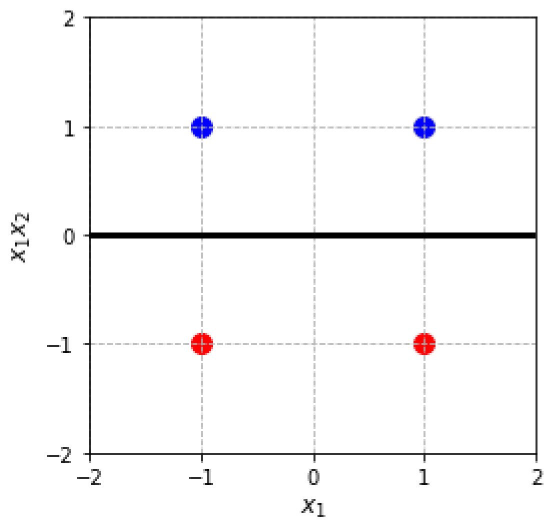


0.1 was chosen as the step sizes for both datasets and 1000 was chosen as the stopping criteria.

Problem 2

P2.1

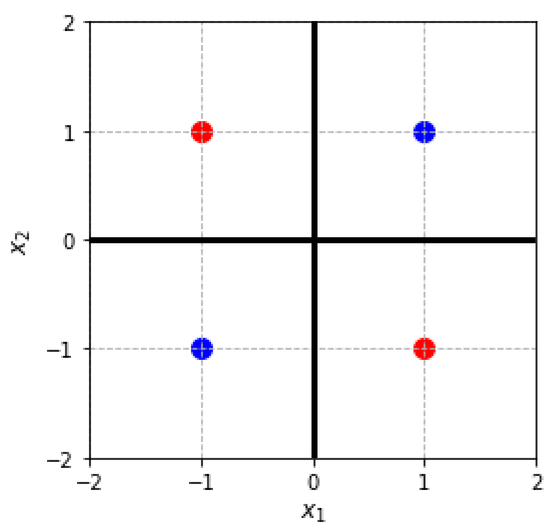
In [4]:
```
f, ax = plt.subplots(1,1, figsize=(4,4))
ax.set_xlim([-2, 2])
ax.set_ylim([-2, 2])
ax.set_xlabel('$x_1$', fontsize=12)
ax.set_ylabel('$x_1x_2$', fontsize=12)
ax.xaxis.set_ticks(range(-2, 3))
ax.yaxis.set_ticks(range(-2, 3))
ax.grid(linestyle='--')

x = np.array([-1, 1,])
y = np.array([1, 1])

ax.scatter(x, y, s=100, c='b')   # class +1
ax.scatter(x, -y, s=100, c='r')  # class -1

# hyperplane
plt.plot([-2,2], [0,0], linewidth=3, color='black')

plt.show()
```



The maximum margin separating hyperplane is where $x_1x_2 = 0$. The max-margin vector w is the vector $[0\ 1]$. The corresponding margin is 2.

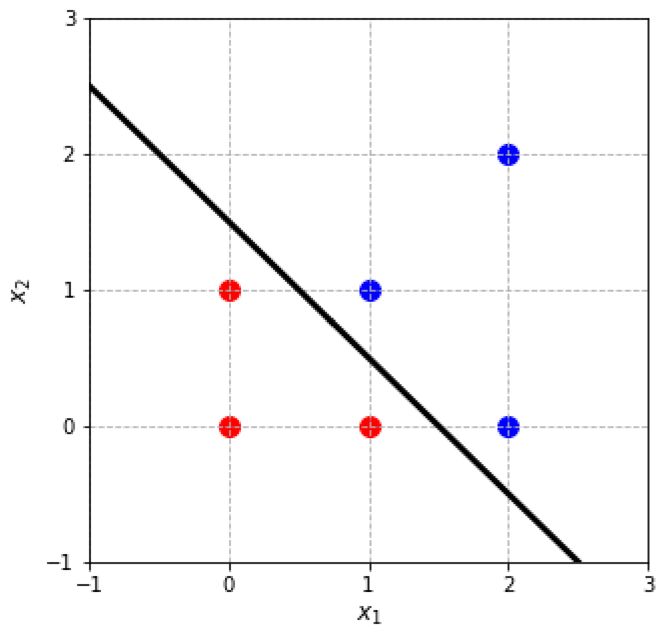
P2.2

In [5]:
```
f, ax = plt.subplots(1,1, figsize=(4,4))
ax.set_xlim([-2, 2])
ax.set_ylim([-2, 2])
ax.set_xlabel('$x_1$', fontsize=12)
ax.set_ylabel('$x_2$', fontsize=12)
ax.xaxis.set_ticks(range(-2, 3))
ax.yaxis.set_ticks(range(-2, 3))
ax.grid(linestyle='--')

x = np.array([-1, 1,])
y = np.array([1, -1,])

ax.scatter(x, x, s=100, c='b')  # class +1
ax.scatter(x, y, s=100, c='r')  # class -1

# hyperplane
y_axis = np.array([-2, 2])
x_axis = np.zeros(2)
plt.plot(y_axis, x_axis, linewidth=3, color='black')
plt.plot(x_axis, y_axis, linewidth=3, color='black')

plt.show()
```



P2.3

In [65]:
```
f, ax = plt.subplots(1,1, figsize=(5,5))
ax.set_xlim([-1, 3])
ax.set_ylim([-1, 3])
ax.xaxis.set_ticks(range(-1, 4))
ax.yaxis.set_ticks(range(-1, 4))
ax.grid(linestyle='--')

ax.set_xlabel('$x_1$', fontsize=12)
ax.set_ylabel('$x_2$', fontsize=12)

# class +1
x1 = np.array([1, 2, 2])
y1 = np.array([1, 0, 2])
ax.scatter(x1, y1, s=100, c='b')

# class -1
x2 = np.array([0, 0, 1])
y2 = np.array([0, 1, 0])
ax.scatter(x2, y2, s=100, c='r')

# hyperplane
y_intercept = np.array([-1, 2.5])
x_intercept = np.array([2.5, -1])
plt.plot(y_intercept, x_intercept, linewidth=3, color='black')

plt.show()
```



The max-margin weight vector w is $[-3\ 2\ 2]$. The corresponding margin is $\sqrt{2}/2 = 0.707$.

P2.4

The support vectors are (0,1), (1,1), (1,0), and (2,0). If we remove the points (1,1) or (1,0), the margin would increase to 1 because the separator would become a vertical line that separates the two classes. If we remove the points (0,1) or (2,0), however, the margin would remain the same because the separator would not move.

Problem 3

I collaborated with Ryan Sivoraphonh on parts 2.1 and 2.2 concerning the hyperplanes of the two problems. I looked extensively on Campuswire and on Google for help on how to write the train function for part 1.6. I also searched Google to learn how to take the derivative of logarithms for part 1.5.