In [1]:
```
import sympy as sym
import numpy as np
import matplotlib.pyplot as plt

# import autograd 's automatic differentiator
from autograd import grad
from autograd import hessian

# datapath to data
datapath = '/home/michaelrencheck/EE475/machine_learning_refined-gh-p
ages/mlrefined_exercises/ed_2/mlrefined_datasets/superlearn_dataset
s/'

sym.init_printing()

# Standard Normalize the data
def std_normalize(in_arr):
    u = np.mean(in_arr, axis=1, dtype=np.float64)
    sig = np.std(in_arr, axis=1, dtype=np.float64)

    out = np.zeros(in_arr.shape)

    for i, row in enumerate(in_arr):
        for j, element in enumerate(row):
            out[i,j] = (element - u[i])/ sig[i]

    return np.squeeze(out)
```

6.5

$$g(w) = -\frac{1}{P}\sum_{p=1}^{P} y_p \log\left(\sigma\left(\tilde{x_p}^T \tilde{w}\right)\right) + (1-y_p)\log\left(1-\sigma\left(\tilde{x_p}^T \tilde{w}\right)\right)$$

Using the fact that: $\sigma'(\tilde{x_p}^T \tilde{w}) = \sigma\left(\tilde{x_p}^T \tilde{w}\right) * \left(1-\sigma\left(\tilde{x_p}^T \tilde{w}\right)\right)$

We can greatly simplify the process of taking the gradient and the hessian.

To find the gradiant:

$$\nabla g(w) = -\frac{1}{P}\sum_{p=1}^{P} y_p \left(\frac{1}{1-\sigma(\cdot)}\right)\sigma(\cdot)(1-\sigma(\cdot))\tilde{x_p} + (1+yp)\left(\frac{1}{1-\sigma(\cdot)}\right)(-\sigma(\cdot))(1-\sigma(\cdot))\tilde{x_p}$$

$$\nabla g(w) = -\frac{1}{P}\sum_{p=1}^{P} y_p(1-\sigma(\cdot))\tilde{x_p} + (1+yp)(-\sigma(\cdot))\tilde{x_p}$$

$$\nabla g(w) = \frac{-1}{P}\sum_{p=1}^{P}(y_p - y_p\sigma(\cdot))\tilde{x_p} + (-\sigma(\cdot) + y_p\sigma(\cdot))\tilde{x_p}$$

$$\nabla g(w) = -\frac{1}{P}\sum_{p=1}^{P}(y_p - \sigma(\cdot))\tilde{x_p}$$

Now to find the Hessian:

$$\nabla^2 g(w) = \frac{1}{P}\sum_{p=1}^{P}\sigma(\cdot)(1-\sigma(\cdot))\tilde{x_p}\tilde{x_p}^T$$

The gradient of y_p is 0 and the negative sign from the sigmoid can be moved outside the summation.

6.10

The zero-order definition of convexity states that:

$$g(\lambda w_1 + (1-\lambda)w_2) \leq \lambda g(w_1) + (1-\lambda)g(w_2)$$

Meaning that the value of g evaluated at some proportional combination of w's is less than or equal to the proportional combination of $g(w_1)$ and $g(w_2)$.

Since the shape of the perceptron is defined by $max(0, -y_p x_p^t w)$ we know that for any w resulting in a negative value for $-y_p x_p^t w$ will result in a value of 0 and for any w resulting in a positive value of $-y_p x_p^t w$ will result in a positive number. This positive result will be linear with the slope of $y_p x_p^T$

Due to attributes of the perception, there are only three cases of w_1 and w_2 that need to be considered:

1. w_1 and w_2 result in a negative number for $-y_p x_p^t w$: the $g(w_1) = 0$ and $g(w_2) = 0$ resulting in a flat line meaning $g(w_\lambda) = 0$ and the definition holds.
2. w_1 will result in a negative number for $-y_p x_p^t w$ and w_2 result in a positive number for $-y_p x_p^t w$: the $g(w_1) = 0$ and $g(w_2) = +number$ connecting these two points with a line will alway result in a value of $g(w_\lambda)$ will always be less than or equal to $0 + (+number) * (1-\lambda)$ meaining the definition holds.
3. w_1 and w_2 result in a positive number for $-y_p x_p^t w$: Since the positive side of the funtion will result in a linear shape, the line connecting $g(w_1)$ and $g(w_2)$ will have the same slope as the perceptron meaning $g(w_\lambda)$ will be equal to that value of the perceptron and the definition holds.

6.13

In [2]:
```
def grad_descent(f, df, x, y,  w_init, n=500, epsilon=1e-3, alpha=0.5
, class_limit=23, normalizer=1.0):

    w = np.copy(w_init)

    cost_history = []
    res_history = []

    i = 0

    done = False

    change = True

    while( not done):

        grad_res = np.zeros([x.shape[0]+1, 1])

        cost = 0

        res = 0

        for j, yp in enumerate(y[0]): # for each data point

            xp = np.concatenate((np.array([1.0]), x[:,j])) # append a
1
            xp = np.reshape(xp, [len(xp), 1]) # make into a column

            cost += f(xp, w, yp)
            grad_res += df(xp, w, yp)

            y_pred = np.sign(np.dot(xp.transpose(),w))

            if (y_pred == 0 ):
                pass
            else:
                res += (y_pred != yp)

        cost /= float(y.shape[1])
        grad_res /= (normalizer * float(y.shape[1]))

        cost_history.append(np.copy(np.squeeze(cost)))
        res_history.append(np.copy(np.squeeze(res)))


        norm = np.linalg.norm(grad_res, 2)

        if res < class_limit:
            done = True
            print("Under Classification Acceptibility", i)
        if norm < epsilon:
            done = True
            print("Iterations to complete:", i)
        elif(i > n):
            done = True
            print("Iteration Limit Exceeded")
```

```
                print("Norm of most recent grad is ", np.linalg.norm(grad
_res, 2))
            else:
                w -= alpha * grad_res
                i += 1

        return w, cost_history, res_history
```

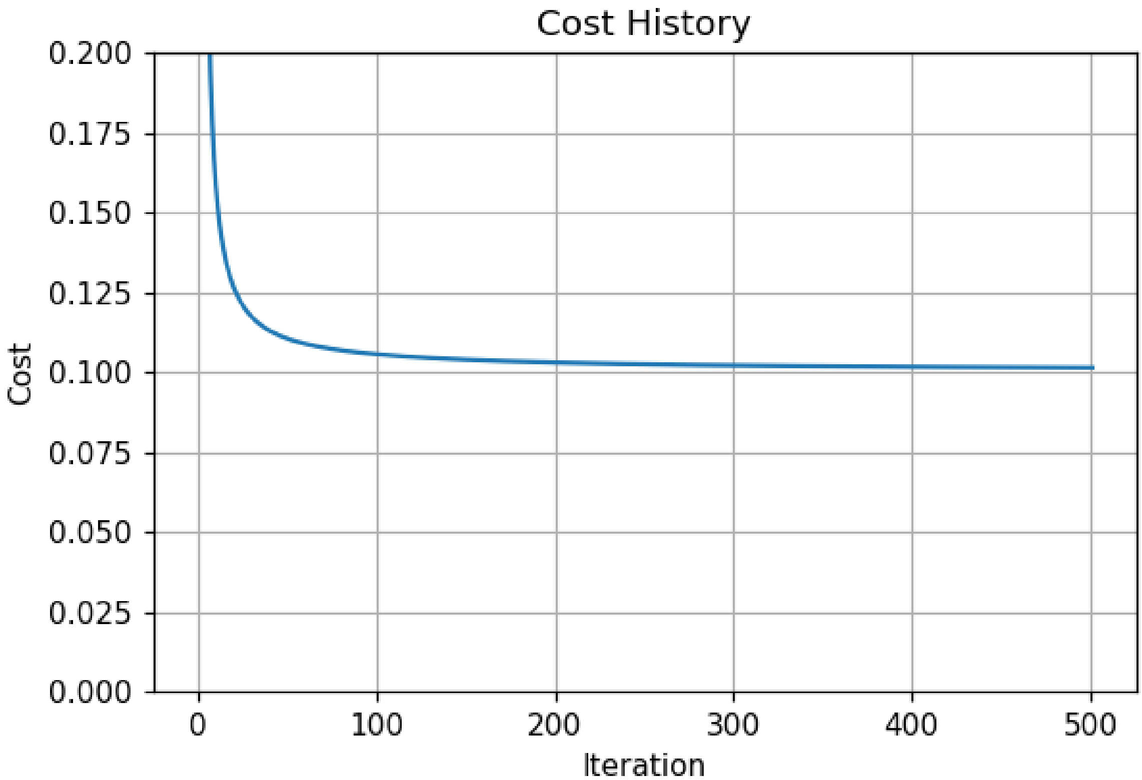
In [3]:
```
# data input
csvname = datapath + 'breast_cancer_data.csv'
data = np.loadtxt(csvname,delimiter = ',')

# get input and output of dataset
x = data[:-1,:]
y = data[-1:,:]

x = std_normalize(x)

print(np.shape(x))
print(np.shape(y))
```

```
(8, 699)
(1, 699)
```

Softmax

In [4]:
```
def softmax(x,w,y):
    """
    x and w are column vectors
    y is a scalar
    """
    exponent = np.dot(-y, np.dot(x.transpose(),w))

    total = np.log(1 + np.exp(exponent))

    return total

def grad_softmax(x,w,y):
    """
    x and w are column vectors
    y is a scalar
    """

    exponent = np.squeeze(-y *  np.dot(x.transpose(), w))

    num = np.exp(exponent)
    denom = 1 + np.exp(exponent)

    total = (num/denom) * y * x

    return total

w_init = np.ones([x.shape[0]+1, 1])

w, costs, res  = grad_descent(softmax, grad_softmax, x, y, w_init, no
rmalizer=-1.0, class_limit=23)

print("Number of misclassifications: ", res[-1])

fig1 = plt.figure(dpi=110,facecolor='w')
plt.grid(True)
plt.plot(np.arange(len(costs)),costs)
plt.title("Cost History")
plt.xlabel("Iteration")
plt.ylabel("Cost")
plt.ylim([0,0.2])
plt.show()
```

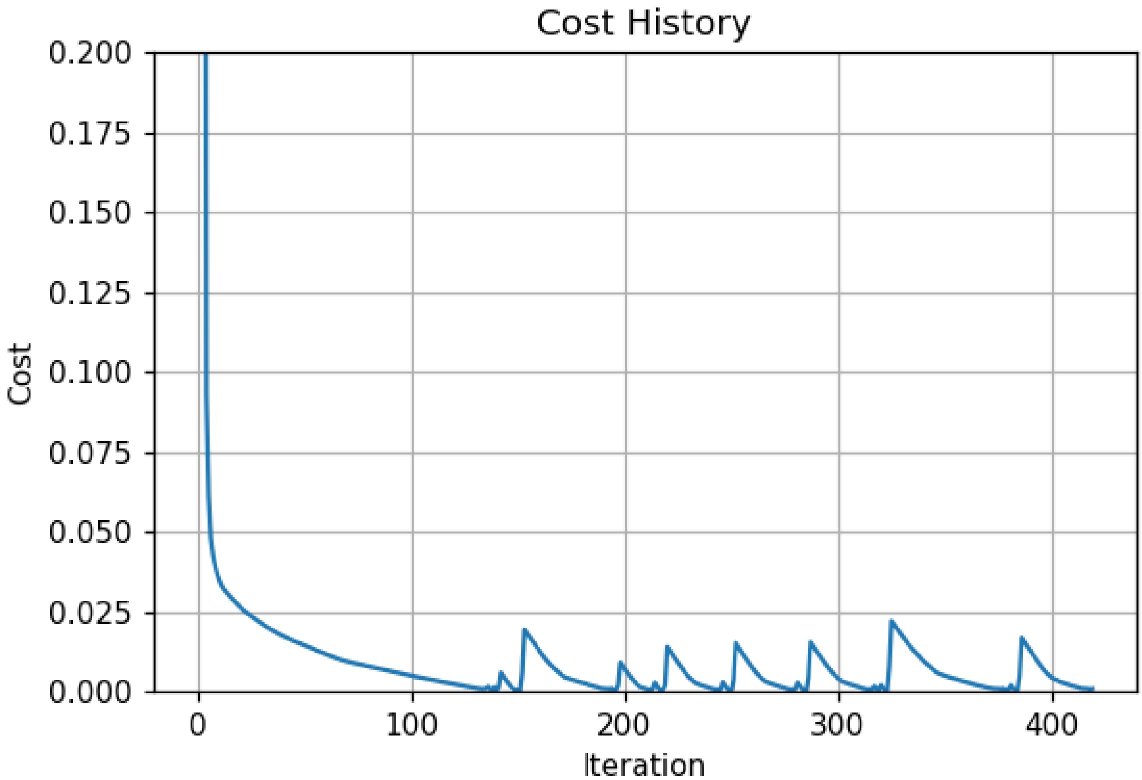
```
Iteration Limit Exceeded
Norm of most recent grad is  0.00213835675325
Number of misclassifications:  26
```



Perceptron

In [5]:
```
def perceptron(x, w, y):
    """
    x and w are column vectors
    y is a scalar
    """
    return max(0, -1.0 * y * np.dot(x.transpose(), w))

def grad_percep(x, w, y):
    """
    x and w are column vectors
    y is a scalar
    """
    if (-1.0 * y * np.dot(x.transpose(), w)) > 0:

        return -1.0 * y * x
    else:
        return 0

w_init = np.ones([x.shape[0]+1, 1])

w, costs, res = grad_descent(perceptron, grad_percep, x, y, w_init, c
lass_limit=21)


print("Number of misclassifications: ", res[-1])

fig1 = plt.figure(dpi=110,facecolor='w')
plt.grid(True)
plt.plot(np.arange(len(costs)),costs)
plt.title("Cost History")
plt.xlabel("Iteration")
plt.ylabel("Cost")
plt.ylim([0,0.2])
plt.show()
```

```
Under Classification Acceptibility 419
Number of misclassifications:  20
```



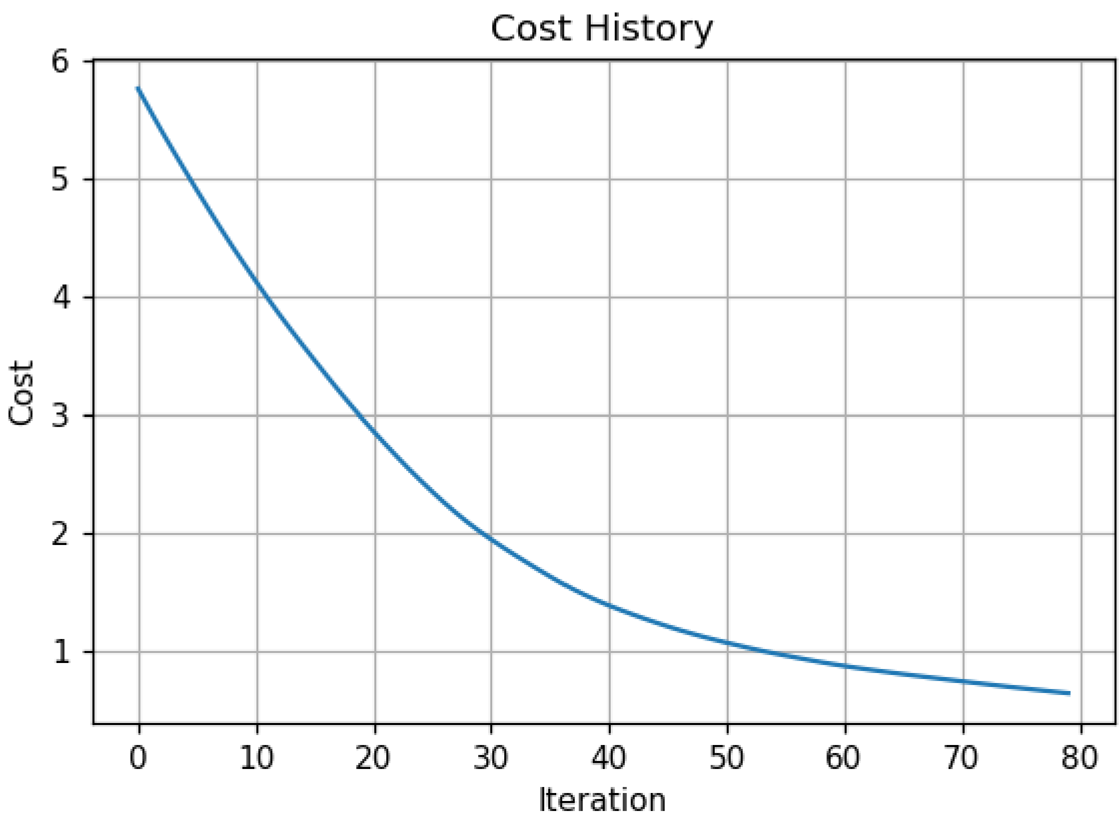
6.15

In [6]:
```
# load in dataset
csvname = datapath + 'credit_dataset.csv'
data = np.loadtxt(csvname,delimiter = ',')
x = data[:-1,:]
y = data[-1:,:]

x = std_normalize(x)

print(np.shape(x))
print(np.shape(y))
```

```
(20, 1000)
(1, 1000)
```

In [111]:
```
w_init = np.ones([x.shape[0]+1, 1])*3.0

w, costs, res = grad_descent(perceptron, grad_percep, x, y, w_init, n
=160, alpha=0.5, class_limit=235)

TP = 0
FP = 0
TN = 0
FN = 0

for j, yp in enumerate(y[0]):

    xp = np.concatenate((np.array([1.0]), x[:,j])) # append a 1

    y_pred = np.sign(np.dot(xp,w))


    if (y_pred == 0):
        TP += 1
    elif(y_pred == yp and yp == 1):
        TP += 1
    elif(y_pred == yp and yp == -1):
        TN += 1
    elif(y_pred != yp and yp == 1):
        FP += 1
    elif(y_pred != yp and yp == -1):
        FN += 1
    else:
        print("error!")

a_plus = TP/(TP+FP)
a_minus = TN/(TN+FN)

a_bal = (a_plus + a_minus)/2

A = (TP + TN)/(TP+FP+TN+FN)

print("\nAccuracy: ", round(A, 3)*100, "%")
print("Balanced Accuracy: ", round(a_bal, 3)*100, "%")
print("\t +1 Accuracy: ", round(a_plus, 3)*100, "%")
print("\t -1 Accuracy: ", round(a_minus, 3)*100, "%")

print("\n Confusion Matrix ==================")
print("{:<6} {:<6} {:<6}".format("    ", "Bad", "Good"))
print("{:<6} {:<6} {:<6}".format("Bad", TN, FN))
print("{:<6} {:<6} {:<6}".format("Good", FP, TP))

fig1 = plt.figure(dpi=110,facecolor='w')
plt.grid(True)
plt.plot(np.arange(len(costs)),costs)
plt.title("Cost History")
plt.xlabel("Iteration")
plt.ylabel("Cost")
plt.show()
```

```
Under Classification Acceptibility 79

Accuracy:  76.6 %
Balanced Accuracy:  68.30000000000001 %
         +1 Accuracy:  89.0 %
         -1 Accuracy:  47.699999999999996 %

 Confusion Matrix =================
       Bad    Good
Bad    143    157
Good   77     623
```



6.16

In [7]:
```
# data input
csvname = datapath + '3d_classification_data_v2_mbalanced.csv'
data1 = np.loadtxt(csvname,delimiter = ',')

# get input and output of dataset
x = data1[:-1,:]
y = data1[-1:,:]

x = std_normalize(x)

print(np.shape(x))
print(np.shape(y))
```

```
(2, 55)
(1, 55)
```

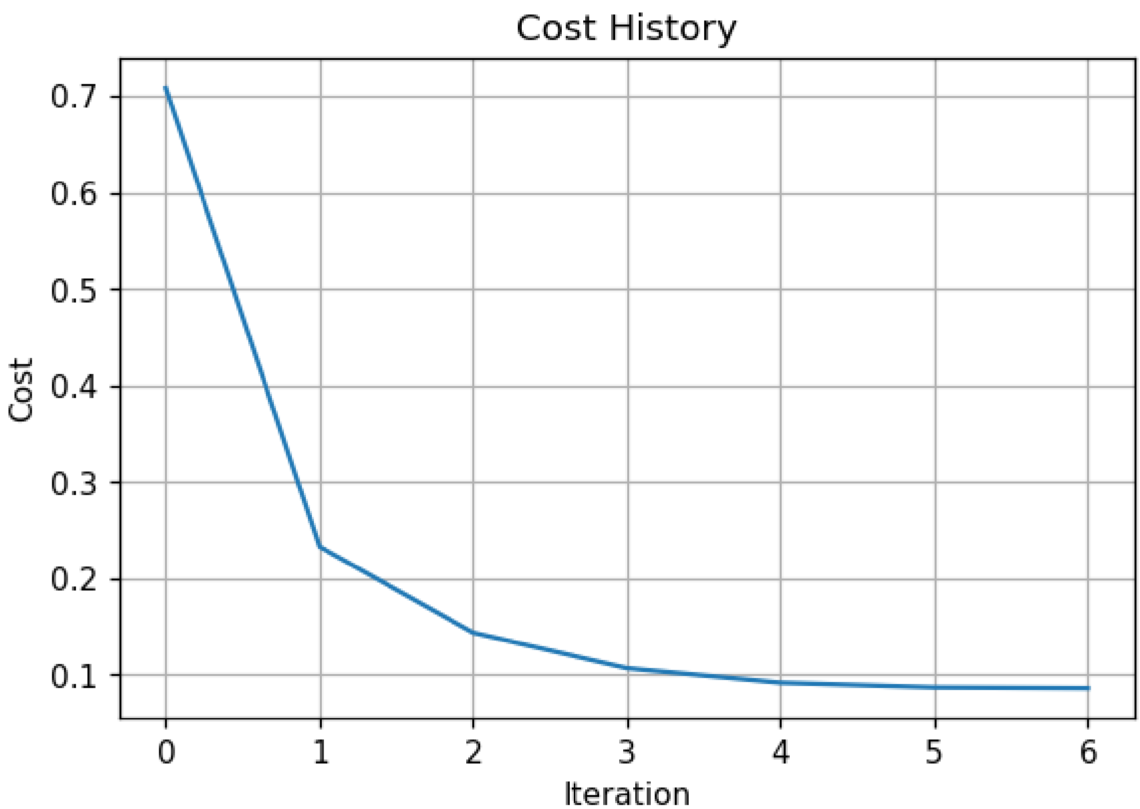
In [26]:
```
def hess_softmax(x, w, y):

    exponent = np.squeeze(y *  np.dot(x.transpose(), w))

    sigma = 1 / (1 +  np.exp(exponent))
    hess = sigma * (1-sigma) * np.dot(x,x.transpose())

    return hess


def newtons(f, df, ddf, x, y, w_init, beta, n=5, normalizer=-1.0):

    w = np.copy(w_init)

    cost_history = []
    res_history = []

    i = 0

    done = False

    while(not done):

        grad_res = np.zeros([x.shape[0]+1, 1])
        hess_res = np.zeros([x.shape[0]+1, x.shape[0]+1])

        cost = 0
        res = 0

        for j, yp in enumerate(y[0]): # for each data point

            xp = np.concatenate((np.array([1.0]), x[:,j])) # append a
1

            xp = np.reshape(xp, [len(xp), 1]) # make into a column

            cost += beta[j] * f(xp, w, yp)
            grad_res += beta[j] * df(xp, w, yp)
            hess_res += beta[j] * ddf(xp, w, yp)

            y_pred = np.sign(np.dot(xp.transpose(),w))

            if (y_pred == 0 ):
                pass
            else:
                res += (y_pred != yp)

        cost /= float(y.shape[1])
        grad_res /= (normalizer * float(y.shape[1]))
        hess_res /= (-normalizer * float(y.shape[1]))

        cost_history.append(np.copy(np.squeeze(cost)))
        res_history.append(np.copy(np.squeeze(res)))

        if(i > n):
            done = True
        else:
```

```
            inv_h_res = np.linalg.inv(hess_res)
            w -= np.dot(inv_h_res, grad_res)
            i += 1

    return w, cost_history, res_history


def calc_accuracy(x, y, w, costs):

    TP = 0
    FP = 0
    TN = 0
    FN = 0

    for j, yp in enumerate(y[0]):

        xp = np.concatenate((np.array([1.0]), x[:,j])) # append a 1

        y_pred = np.sign(np.dot(xp,w))


        if (y_pred == 0):
            TP += 1
        elif(y_pred == yp and yp == 1):
            TP += 1
        elif(y_pred == yp and yp == -1):
            TN += 1
        elif(y_pred != yp and yp == 1):
            FP += 1
        elif(y_pred != yp and yp == -1):
            FN += 1
        else:
            print("error!")

    a_plus = TP/(TP+FP)
    a_minus = TN/(TN+FN)

    a_bal = (a_plus + a_minus)/2

    A = (TP + TN)/(TP+FP+TN+FN)

    print("\nAccuracy: ", round(A, 3)*100, "%")
    print("Balanced Accuracy: ", round(a_bal, 3)*100, "%")
    print("\t +1 Accuracy: ", round(a_plus, 3)*100, "%")
    print("\t -1 Accuracy: ", round(a_minus, 3)*100, "%")

    print("\n Confusion Matrix =================")
    print("{:<6} {:<6} {:<6}".format("    ", "Bad", "Good"))
    print("{:<6} {:<6} {:<6}".format("Bad", TN, FN))
    print("{:<6} {:<6} {:<6}".format("Good", FP, TP))

    fig1 = plt.figure(dpi=110,facecolor='w')
    plt.grid(True)
    plt.plot(np.arange(len(costs)),costs)
    plt.title("Cost History")
    plt.xlabel("Iteration")
```

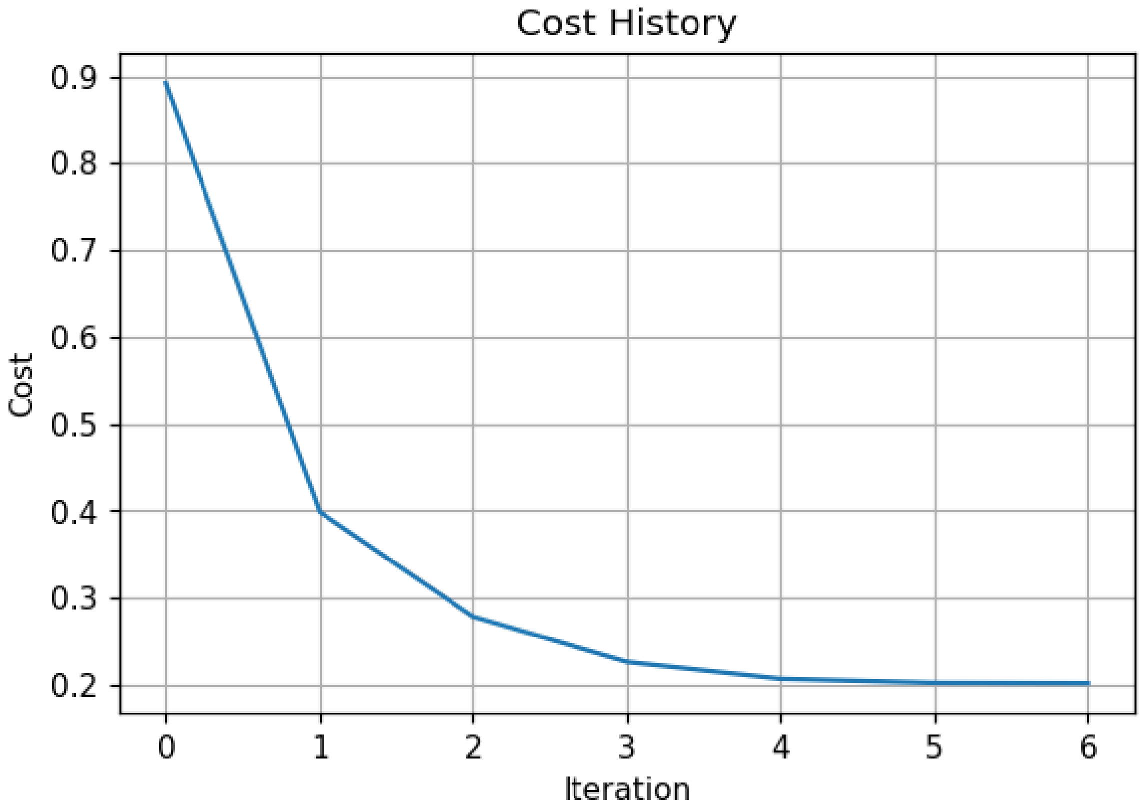
```
plt.ylabel("Cost")
plt.show()
```

In [38]:
```
# build beta array
num = np.array([1,5,10])

w_init = np.ones([x.shape[0]+1, 1])*0.1

for i in range(len(num)):
    print("Beta = ", num[i], " for the minority class")
    beta = num[i] * (y >= 0.0)
    beta = np.squeeze(beta + (beta == 0))

    w, costs, res = newtons(softmax, grad_softmax, hess_softmax, x, y
, w_init, beta)
    calc_accuracy(x, y, w, costs)
    print("==========================================================
==========\n\n")
```

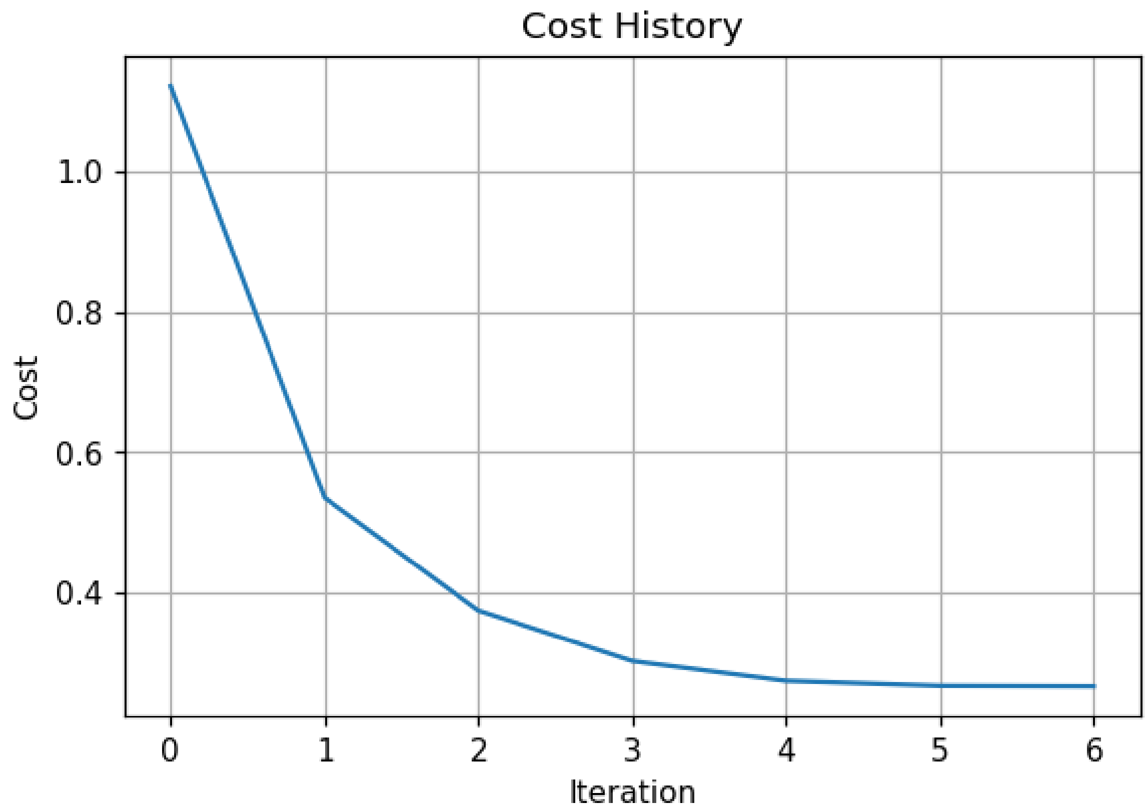
```
Beta =  1  for the minority class

Accuracy:  94.5 %
Balanced Accuracy:  79.0 %
         +1 Accuracy:  60.0 %
         -1 Accuracy:  98.0 %

 Confusion Matrix =================
       Bad    Good
Bad    49     1
Good   2      3
```




```
====================================================================


Beta =  5  for the minority class

Accuracy:  92.7 %
Balanced Accuracy:  87.0 %
         +1 Accuracy:  80.0 %
         -1 Accuracy:  94.0 %

 Confusion Matrix =================
       Bad    Good
Bad    47     3
Good   1      4
```




```
====================================================================


Beta =  10  for the minority class

Accuracy:  92.7 %
Balanced Accuracy:  96.0 %
         +1 Accuracy:  100.0 %
         -1 Accuracy:  92.0 %

 Confusion Matrix =================
       Bad    Good
Bad    46     4
Good   0      5
```


Cost History
1.0
0.8
0.6
0.4
Cost
0
1
2
3
4
5
6
Iteration

===