Table of Contents

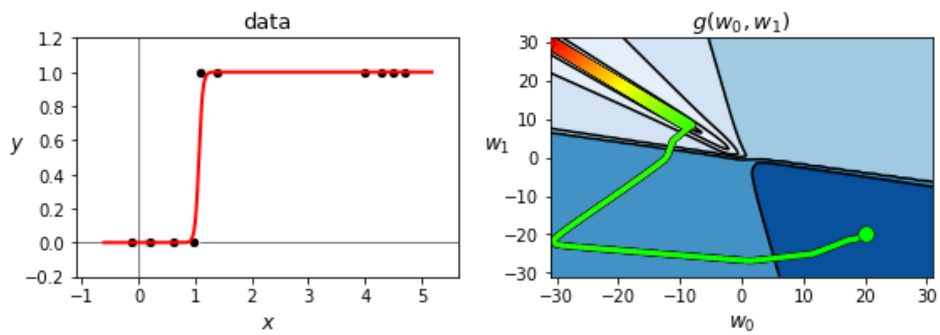
In [1]:
```
# basic imports
import sys
sys.path.append('../')
import matplotlib.pyplot as plt

# import autograd wrapped numpy
import autograd.numpy as np

# imports from custom library
from mlrefined_libraries import basics_library as baslib
from mlrefined_libraries import calculus_library as calib
from mlrefined_libraries import math_optimization_library as optlib
from mlrefined_libraries import superlearn_library as superlearn

# demos for this notebook
regress_plotter = superlearn.lin_regression_demos
optimizers = optlib.optimizers
static_plotter = optlib.static_plotter.Visualizer();
plotter = superlearn.multi_outupt_plotters

# load in baic libraries
from autograd import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
datapath = '../mlrefined_datasets/superlearn_datasets/'

# this is needed to compensate for matplotlib notebook's tendancy to blow up imag
es when plotted inline
%matplotlib notebook
from matplotlib import rcParams
rcParams['figure.autolayout'] = True
```

Exercise 6.1. Implementing sigmoidal Least Squares cost

In [2]:
```
# define sigmoid function
def sigmoid(t):
    return 1/(1 + np.exp(-t))

# sigmoid non-convex logistic least squares cost function
def sigmoid_least_squares(w):
    cost = 0
    for p in range(y.size):
        x_p = x[:,p]
        y_p = y[:,p]
        cost += (sigmoid(w[0] + w[1]*x_p) - y_p)**2
    return cost/y.size
```

In [3]:
```
# load in data
csvname = datapath + '2d_classification_data_v1_entropy.csv'
data = np.loadtxt(csvname,delimiter = ',')

# load in optimizer
opt = superlearn.optimizers.MyOptimizers()

# get input/output pairs
x = data[:-1,:]
y = data[-1:,:]

# run normalized gradient descent
w = np.asarray([20.0,-20.0])[:,np.newaxis]
w_hist = opt.gradient_descent(g = sigmoid_least_squares,w = w,version = 'normaliz
ed',max_its = 900, alpha = 1)
```

In [4]:
```
# create instance of logisic regression demo and load in data, cost function, and
descent history
demo2 = superlearn.classification_2d_demos_entropy.Visualizer(data,sigmoid_least_
squares)

# create a static figure illustrating gradient descent steps
demo2.static_fig(w_hist,num_contours = 25,viewmax = 31)
```



Exercise 6.2. Show the equivalence of the Log Error and Cross Entropy point-wise cost

Consider the following cases:

Case 1. $y_p = 1$

Plugging $y_p = 1$ into

$$-y_p \log \sigma\left(\mathring{\mathbf{x}}_p^T \mathbf{w}\right) - (1 - y_p) \log\left(1 - \sigma\left(\mathring{\mathbf{x}}_p^T \mathbf{w}\right)\right)$$

gives

$$-\log \sigma\left(\mathring{\mathbf{x}}_p^T \mathbf{w}\right)$$

Case 2. $y_p = 0$

Plugging $y_p = 0$ into

$$-y_p \log \sigma\left(\mathring{\mathbf{x}}_p^T \mathbf{w}\right) - (1 - y_p) \log\left(1 - \sigma\left(\mathring{\mathbf{x}}_p^T \mathbf{w}\right)\right)$$

gives

$$-\log\left(1 - \sigma\left(\mathring{\mathbf{x}}_p^T \mathbf{w}\right)\right)$$

Exercise 6.3. Implementing the Cross Entropy cost

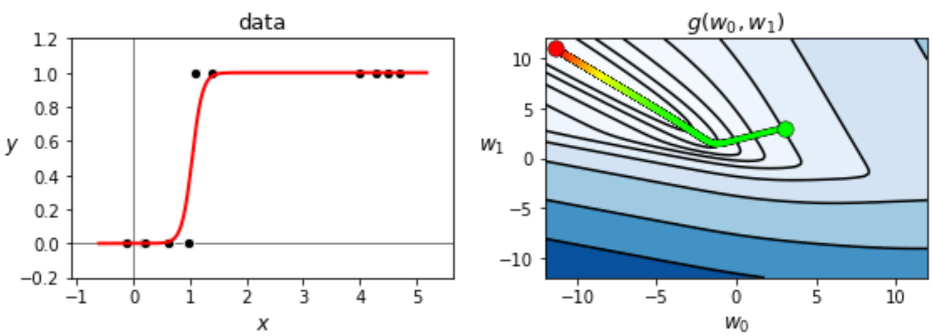
In [5]:
```
# compute linear combination of input point
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T
```

We can then implement the Cross Entropy cost function by e.g., implementing the Log Loss error and employing efficient and compact `numpy` operations (see the general discussion in Section 3.1.3) as

In [6]:
```
# define sigmoid function
def sigmoid(t):
    return 1/(1 + np.exp(-t))

# the convex cross-entropy cost function
def cross_entropy(w):
    # compute sigmoid of model
    a = sigmoid(model(x,w))

    # compute cost of label 0 points
    ind = np.argwhere(y == 0)[:,1]
    cost = -np.sum(np.log(1 - a[:,ind]))

    # add cost on label 1 points
    ind = np.argwhere(y==1)[:,1]
    cost -= np.sum(np.log(a[:,ind]))

    # compute cross-entropy
    return cost/y.size
```

In [7]:
```
# This code cell will not be shown in the HTML version of this notebook
# take input/output pairs from data
x = data[:-1,:]
y = data[-1:,:]

# run gradient descent to minimize the softmax cost
g = cross_entropy; w = np.array([3.0,3.0])[:,np.newaxis]; max_its = 100; alpha_ch
oice = 10**(0);
weight_history,cost_history = optimizers.gradient_descent(g,alpha_choice,max_its,
w)
```

In [8]:
```
# run gradient descent to minimize the softmax cost
g = cross_entropy; w = np.array([3.0,3.0])[:,np.newaxis]; max_its = 2000; alpha_c
hoice = 1;
weight_history,cost_history = optimizers.gradient_descent(g,alpha_choice,max_its,
w)

# create a static figure illustrating gradient descent steps
animator = superlearn.classification_2d_demos_entropy.Visualizer(data,cross_entro
py)
animator.static_fig(weight_history,num_contours = 25,viewmax = 12)
```



Exercise 6.4. Compute the Lipschitz constant of the Cross Entropy cost

Building on the analysis shown in the Endnotes of this Chapter showing that the cross-entropy cost is convex, we can likewise compute its largest possible eigenvalue by noting that the largest value σ_p (defined previously) can take is $\frac{1}{4}$

$$\sigma_k \leq \frac{1}{4}$$

Thus the largest value the \emph{Rayleigh quotient} can take is bounded above for any $\mathbf{z}$ as

$$\mathbf{z}^T \nabla^2 g\left(\mathbf{w}\right) \mathbf{z} \ \leq \frac{1}{4P} \mathbf{z}^T \left(\sum_{p=1}^{P} \mathring{\mathbf{x}}_p \mathring{\mathbf{x}}_p^T\right) \mathbf{z}$$

Since the maximum value $\mathbf{z}^T \left(\sum_{p=1}^{P} \mathring{\mathbf{x}}_p \mathring{\mathbf{x}}_p^T\right) \mathbf{z}$ can take is the maximum eigenvalue of the matrix $\sum_{p=1}^{P} \mathring{\mathbf{x}}_p \mathring{\mathbf{x}}_p^T$, thus a Lipschitz constant for the Cross Entropy cost is given as

$$L = \frac{1}{4P} \left\Vert \sum_{p=1}^{P} \mathring{\mathbf{x}}_p \mathring{\mathbf{x}}_p^T \right\Vert_2^2$$

Exercise 6.5. Confirm gradient and Hessian calculations

The Cross Entropy cost function is given as

$$g(\mathbf{w}) = -\frac{1}{P}\sum_{p=1}^{P} y_p \,\log \sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right) + (1-y_p)\log\left(1-\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right)$$

First, let us focus on the pth summand

$$g_p(\mathbf{w}) = y_p \,\log \sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right) + (1-y_p)\log\left(1-\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right)$$

and compute its partial derivative with respect to the jth entry in $\mathbf{w}$, as

$$\frac{\partial g_p}{\partial w_j} = y_p\, x_{p,j}\frac{\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\left(1-\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right)}{\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)} - (1-y_p)\,x_{p,j}\frac{\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\left(1-\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right)}{1-\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)}$$

where we have used the fact that $\frac{\mathrm{d}}{\mathrm{d}w}\sigma = \sigma(w)\,(1-\sigma(w))$.

Simplifying gives

$$\frac{\partial g_p}{\partial w_j} = \left(y_p - \sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right)x_{p,j}$$

Forming the full gradient vector we have

$$\nabla g_p(\mathbf{w}) = \left(y_p - \sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right)\mathring{\mathbf{x}}_p$$

Taking the sum over all datapoints we have the final form of the gradient as

$$\nabla g(\mathbf{w}) = -\frac{1}{P}\sum_{p=1}^{P}\left(y_p - \sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right)\mathring{\mathbf{x}}_p$$

To compute the (i,j)th entry in the Hessian matrix, we take the partial derivative with respect to w_i of $\frac{\partial g_p}{\partial w_j}$ (whose form is already computed above), as

$$\frac{\partial}{\partial w_i}\frac{\partial g_p}{\partial w_j} = \frac{\partial}{\partial w_i}\left(y_p - \sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right)x_{p,j} = -\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\left(1-\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right)x_{p,i}\,x_{p,j}$$

Taking the sum over all datapoints, we can write the final Hessian matrix as a sum of outer-product matrices of the form

$$\begin{aligned}\nabla^2 g(\mathbf{w}) &= -\frac{1}{P}\sum_{p=1}^{P} -\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\left(1-\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right)\mathring{\mathbf{x}}_p\,\mathring{\mathbf{x}}_p^T \\ &= \frac{1}{P}\sum_{p=1}^{P} \sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\left(1-\sigma\left(\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right)\mathring{\mathbf{x}}_p\,\mathring{\mathbf{x}}_p^T\end{aligned}$$

Exercise 6.6. Show the equivalence of the Log Error and Softmax point-wise cost

Consider the following cases:

Case 1. $y_p = +1$

Plugging $y_p = +1$ into

$$\log\left(1 + e^{-y_p \mathring{\mathbf{x}}_p^T \mathbf{w}}\right)$$

we have

$$\log\left(1 + e^{-y_p \mathring{\mathbf{x}}_p^T \mathbf{w}}\right) = \log\left(1 + e^{-\mathring{\mathbf{x}}_p^T \mathbf{w}}\right) = -\log\left(\frac{1}{1 + e^{-\mathring{\mathbf{x}}_p^T \mathbf{w}}}\right) = -\log \sigma\left(\mathring{\mathbf{x}}_p^T \mathbf{w}\right)$$

Case 2. $y_p = -1$

Plugging $y_p = -1$ into

$$\log\left(1 + e^{-y_p \mathring{\mathbf{x}}_p^T \mathbf{w}}\right)$$

we have

$$\log\left(1 + e^{-y_p \mathring{\mathbf{x}}_p^T \mathbf{w}}\right) = \log\left(1 + e^{\mathring{\mathbf{x}}_p^T \mathbf{w}}\right) = -\log\left(\frac{1}{1 + e^{\mathring{\mathbf{x}}_p^T \mathbf{w}}}\right) = -\log \sigma\left(-\mathring{\mathbf{x}}_p^T \mathbf{w}\right)$$

Exercise 6.7. Implementing the Softmax cost

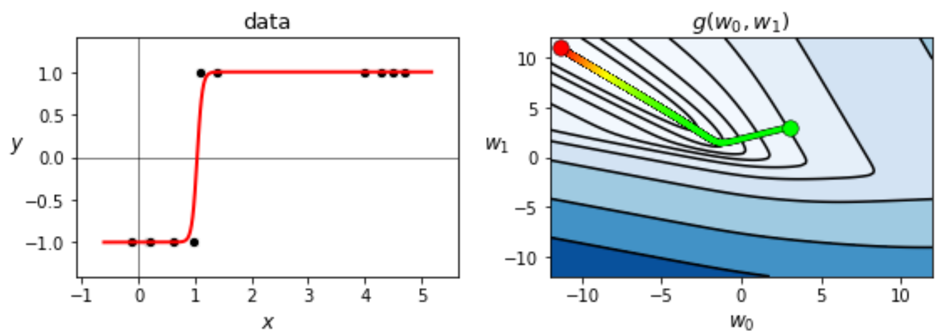
In [9]:
```
# compute linear combination of input point
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T
```

In [10]:
```
# the convex softmax cost function
def softmax(w):
    cost = np.sum(np.log(1 + np.exp(-y*model(x,w))))
    return cost/float(np.size(y))
```

In [11]:
```
# load in data
csvname = datapath + '2d_classification_data_v1.csv'
data = np.loadtxt(csvname,delimiter = ',')

# take input/output pairs from data
x = data[:-1,:]
y = data[-1:,:]

# run gradient descent to minimize the softmax cost
g = softmax; w = np.array([3.0,3.0])[:,np.newaxis]; max_its = 100; alpha_choice = 
1;
weight_history,cost_history = optimizers.gradient_descent(g,alpha_choice,max_its,
w)
```

Below we show the result of running gradient descent with the same initial point and fixed steplength parameter for 2000 iterations, which results in a better fit.

In [12]:
```
# run gradient descent to minimize the softmax cost
g = softmax; w = np.array([3.0,3.0])[:,np.newaxis]; max_its = 2000; alpha_choice
= 1;
weight_history,cost_history = optimizers.gradient_descent(g,alpha_choice,max_its,
w)

# create a static figure illustrating gradient descent steps
animator = superlearn.classification_2d_demos.Visualizer(data,g)
animator.static_fig(weight_history,num_contours = 25,viewmax = 12)
```

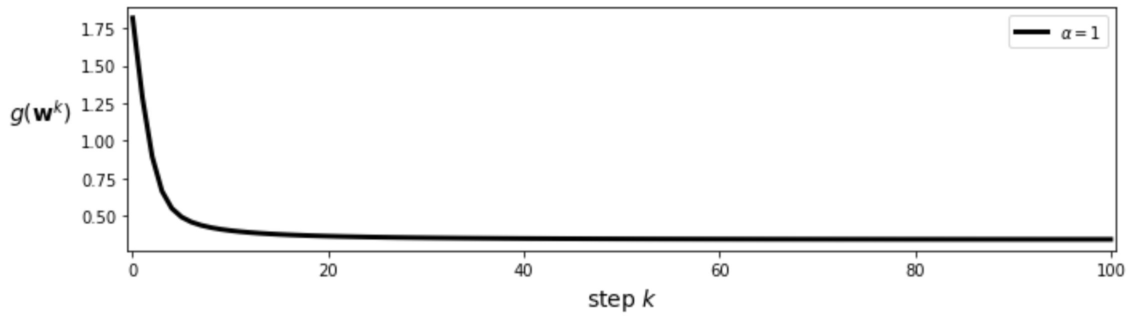


Exercise 6.8. Implementing the Log Error version of Softmax

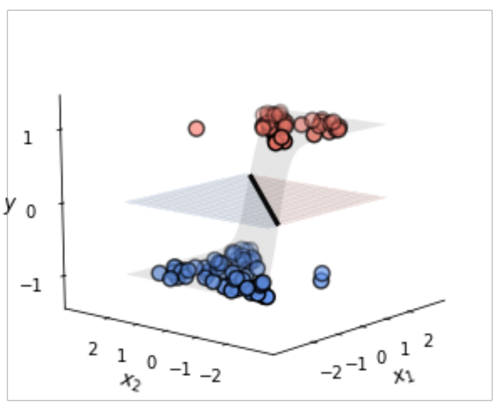
In [13]:
```
# define sigmoid function
def sigmoid(t):
    return 1/(1 + np.exp(-t))

# the convex cross-entropy cost function
def softmax(w):
    # compute sigmoid of model
    a = sigmoid(model(x,w))

    # compute cost of label 0 points
    ind = np.argwhere(y == -1)[:,1]
    cost = -np.sum(np.log(1 - a[:,ind]))

    # add cost on label 1 points
    ind = np.argwhere(y==+1)[:,1]
    cost -= np.sum(np.log(a[:,ind]))

    # compute cross-entropy
    return cost/y.size
```

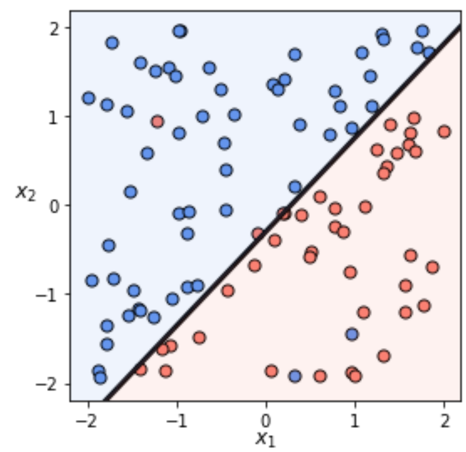
In [14]:
```
# load in dataset
data = np.loadtxt(datapath + '3d_classification_data_v0.csv',delimiter = ',')

# create instance of linear regression demo, used below and in the next examples
demo = superlearn.classification_3d_demos.Visualizer(data)
```

In [15]:
```
# get input/output pairs
x = data[:-1,:]
y = data[-1:,:]

# run gradient descent to minimize the softmax cost
g = softmax; w = np.random.randn(3,1); max_its = 100; alpha_choice = 1;
weight_history,cost_history = optimizers.gradient_descent(g,alpha_choice,max_its,
w)

# plot the cost function history for a given run
static_plotter.plot_cost_histories([cost_history],start = 0,points = False,labels
= [r'$\alpha = 1$'])
```



In [16]:
```
# create instance of 3d demos
demo = superlearn.classification_3d_demos.Visualizer(data)

# draw the final results
demo.static_fig(weight_history[-1],view = [15,-140])
```





Exercise 6.9. Using gradient descent to minimize the Perceptron cost

In [17]:
```
data = np.loadtxt(datapath + '3d_classification_data_v0.csv',delimiter = ',')
x = data[:-1,:]
y = data[-1:,:]
```

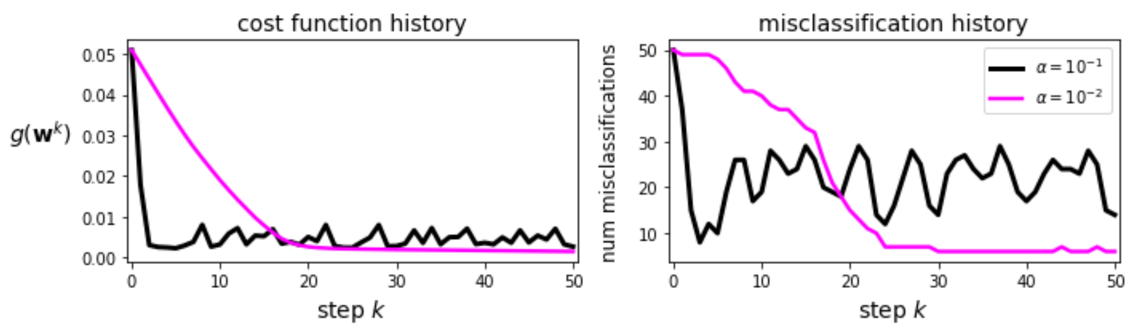
In [18]:
```
# compute linear combination of input points
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T

# an implementation of the perceptron cost
def perceptron(w):
    # compute the least squares cost
    cost = np.sum(np.maximum(0,-y*model(x,w)))
    return cost/float(np.size(y))
```

In [23]:
```
# setup optimizer input (besides cost)
alpha = 10**(-1)
max_its = 50
w = 0.1*np.random.randn(2+1,1)

# run gradient descent to minimize the Least Squares cost for linear regression
g = perceptron;
weight_history_1,cost_history_1 = optimizers.gradient_descent(g,alpha,max_its,w)
alpha = 10**(-2)
weight_history_2,cost_history_2 = optimizers.gradient_descent(g,alpha,max_its,w)
```

In [24]:
```
### cost functions ###
def counting_cost(w,x,y):
    # compute predicted labels
    y_hat = np.sign(model(x,w))

    # compare to true labels
    ind = np.argwhere(y != y_hat)
    ind = [v[1] for v in ind]

    cost = np.sum(len(ind))
    return cost
```

In [25]:
```
count_history_1 = [counting_cost(v,x,y) for v in weight_history_1]
count_history_2 = [counting_cost(v,x,y) for v in weight_history_2]
```

In [26]:
```
# plot history
classif_plotter = superlearn.classification_static_plotter.Visualizer()

cost_histories = [cost_history_1,cost_history_2]
count_histories = [count_history_1,count_history_2]
classif_plotter.plot_cost_histories(cost_histories,count_histories,start = 0,poin
ts = False,labels = [r'$\alpha = 10^{-1}$',r'$\alpha = 10^{-2}$'])
```



Exercise 6.10. The Perceptron cost is convex

a) Defining the constant vector $\mathbf{c} = -y_p\mathring{\mathbf{x}}_p$, we can write each summand as $h\left(\mathbf{w}\right) = \max\left(0,\ \mathbf{c}^T\mathbf{w}\right)$. According to the zeroth order definition of convexity, h is convex if\noindent

$$h\left(\lambda\mathbf{w}_1 + \left(1-\lambda\right)\mathbf{w}_2\right) \leq \lambda h\left(\mathbf{w}_1\right) + \left(1-\lambda\right)h\left(\mathbf{w}_2\right),$$

for all $0 \leq \lambda \leq 1$. Using the definition of h we need to show that

$$\max\left(0,\ \mathbf{c}^T\left(\lambda\mathbf{w}_1 + \left(1-\lambda\right)\mathbf{w}_2\right)\right) \leq \lambda\max\left(0,\ \mathbf{c}^T\mathbf{w}_1\right) + \left(1-\lambda\right)\max\left(0,\ \mathbf{c}^T\mathbf{w}_2\right).$$

or equivalently

$$\max\left(0,\ \lambda\mathbf{c}^T\mathbf{w}_1 + \left(1-\lambda\right)\mathbf{c}^T\mathbf{w}_2\right) \leq \max\left(0,\ \lambda\mathbf{c}^T\mathbf{w}_1\right) + \max\left(0,\ \left(1-\lambda\right)\mathbf{c}^T\mathbf{w}_2\right)$$

using the fact that $\lambda\max\left(0,\ \zeta\right) = \max\left(0,\ \lambda\zeta\right)$ when λ is nonnegative. Now to see that this does indeed holds, we break it down into four cases:

Case 1. $\lambda\mathbf{c}^T\mathbf{w}_1 \geq 0$ and $\left(1-\lambda\right)\mathbf{c}^T\mathbf{w}_2 \geq 0$

In this case both sides of the above simplify to $\lambda\mathbf{c}^T\mathbf{w}_1 + \left(1-\lambda\right)\mathbf{c}^T\mathbf{w}_2$ and we have equality.

Case 2. $\lambda\mathbf{c}^T\mathbf{w}_1 < 0$ and $\left(1-\lambda\right)\mathbf{c}^T\mathbf{w}_2 < 0$

In this case both sides of the above simplify to 0 and we have equality again.

Case 3. $\lambda\mathbf{c}^T\mathbf{w}_1 \geq 0$ and $\left(1-\lambda\right)\mathbf{c}^T\mathbf{w}_2 < 0$

In this case we have

$$\max\left(0,\ \lambda\mathbf{c}^T\mathbf{w}_1 + \left(1-\lambda\right)\mathbf{c}^T\mathbf{w}_2\right) \leq \max\left(0,\ \lambda\mathbf{c}^T\mathbf{w}_1\right) \leq \max\left(0,\ \lambda\mathbf{c}^T\mathbf{w}_1\right) + \max\left(0,\ \left(1-\lambda\right)\mathbf{c}^T\mathbf{w}_2\right).$$

Case 4. $\lambda\mathbf{c}^T\mathbf{w}_1 < 0$ and $\left(1-\lambda\right)\mathbf{c}^T\mathbf{w}_2 \geq 0$

In this case we have

$$\max\left(0,\ \lambda\mathbf{c}^T\mathbf{w}_1 + \left(1-\lambda\right)\mathbf{c}^T\mathbf{w}_2\right) \leq \max\left(0,\ \left(1-\lambda\right)\mathbf{c}^T\mathbf{w}_2\right) \leq \max\left(0,\ \lambda\mathbf{c}^T\mathbf{w}_1\right) + \max\left(0,\ \left(1-\lambda\right)\mathbf{c}^T\mathbf{w}_2\right).$$

Exercise 6.11. The Softmax cost is convex

The Hessian for the logistic regression cost g is given by\noindent

$$\nabla^2 g\left(\mathbf{w}\right) = \sum_{p=1}^{P}\sigma\left(-y_p\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\left(1-\sigma\left(-y_p\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right)\mathring{\mathbf{x}}_p\mathring{\mathbf{x}}_p^T.$$

To prove g is convex, we use the second order definition of convexity and show $\nabla^2 g\left(\mathbf{w}\right)$ is positive semidefinite by forming\noindent

$$\psi\left(\mathbf{z}\right) = \mathbf{z}^T\nabla^2 g\left(\mathbf{w}\right)\mathbf{z} = \mathbf{z}^T\left[\sum_{p=1}^{P}\sigma\left(-y_p\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\left(1-\sigma\left(-y_p\mathring{\mathbf{x}}_p^T\mathbf{w}\right)\right)\mathring{\mathbf{x}}_p\mathring{\mathbf{x}}_p^T\right]\mathbf{z}$$

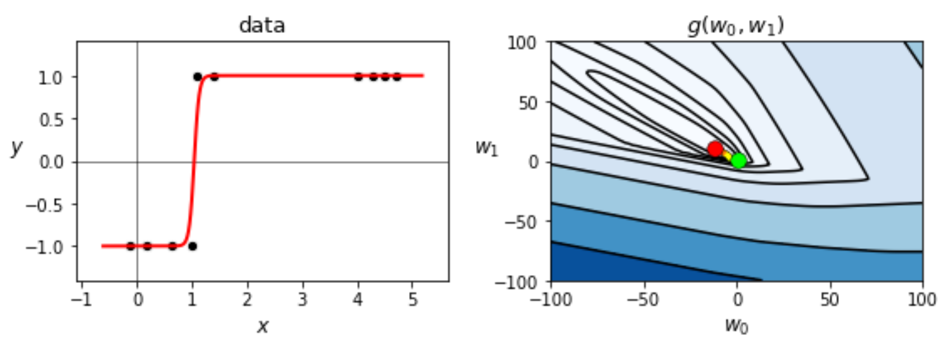
$$= \sum_{p=1}^{P} \delta_p \left(\mathring{\mathbf{x}}_p^T \mathbf{z} \right)^2,$$

where $\delta_p = \sigma\left(-y_p \mathring{\mathbf{x}}_p^T \mathbf{w}\right)\left(1 - \sigma\left(-y_p \mathring{\mathbf{x}}_p^T \mathbf{w}\right)\right)$. Now note that since $0 \leq \sigma\left(\zeta\right) \leq 1$ we have that $\delta_p \geq 0$, and therefore \noindent

$$\mathbf{z}^T \nabla^2 g\left(\mathbf{w}\right) \mathbf{z} = \sum_{p=1}^{P} \delta_p \left(\mathring{\mathbf{x}}_p^T \mathbf{z} \right)^2 \geq 0.$$

Exercise 6.12. The regularized Softmax

In [27]:
```
# This code cell will not be shown in the HTML version of this notebook
### define softmax cost ###
# compute linear combination of input point
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T

# the convex softmax cost function
lam = 2*10**(-3)
def softmax(w):
    # compute cost value
    cost = np.sum(np.log(1 + np.exp(-y*model(x,w))))

    # add regularizer
    cost += lam*np.sum(w[1:]**2)
    return cost/float(np.size(y))

# load in dataset
data = np.loadtxt(datapath + '2d_classification_data_v1.csv',delimiter = ',')

# get input/output pairs
x = data[:-1,:]
y = data[-1:,:]

# run gradient descent to minimize the softmax cost
g = softmax; w = np.ones((2,1)); max_its = 5;
weight_history,cost_history = optimizers.newtons_method(g,max_its,w,epsilon = 1
0**(-7))

# create instance of logisic regression demo and load in data, cost function, and
descent history
animator = superlearn.classification_2d_demos.Visualizer(data,softmax)

# create a static figure illustrating gradient descent steps
animator.static_fig(weight_history,num_contours = 25,viewmax = 100)
```



Exercise 6.13. Compare the efficacy of two-class cost functions I

Below we load in the breast cancer dataset - [a description of which you can find here (https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original)](https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original)). The input datapoints are stacked *column-wise* in this dataset, with the final row being the label of each point.

In [28]:
```
# data input
csvname = datapath + 'breast_cancer_data.csv'
data1 = np.loadtxt(csvname,delimiter = ',')

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

Here `x` contains an input point in each column - there are $N = 18$ input features and $P = 699$ datapoints.

In [58]:
```
print (x.shape)
```

```
(8, 699)
```

Here `y` contains the labels for each point.

In [59]:
```
print (y.shape)
```

```
(1, 699)
```

We will use the gradient descent module defined in Chapter 3 here as a backend file.

Below we implement each of the required cost functions.

In [60]:
```
# compute linear combination of input points
def model(x,w):
    a = w[0] + np.dot(x.T,w[1:])
    return a.T
```

In [61]:
```
# an implementation of the perceptron cost
def perceptron(w):
    # compute the least squares cost
    cost = np.sum(np.maximum(0,-y*model(x,w)))
    return cost/float(np.size(y))
```

In [62]:
```
# an implementation of the softmax cost
def softmax(w):
    # compute the least squares cost
    cost = np.sum(np.log(1 + np.exp(-y*model(x,w))))
    return cost/float(np.size(y))
```

Now we run gradient descent to minimize each over the first (breast cancer dataset), plotting the resulting cost function and misclassification histories.

In [63]:
```
# setup data
N = x.shape[0]

# setup optimizer input (besides cost)
alpha = 10**(-1)
max_its = 1000
w = 0.1*np.random.randn(N+1,1)

# run gradient descent to minimize the Least Squares cost for linear regression
g = perceptron;
weight_history_1,cost_history_1 = optimizers.gradient_descent(g,alpha,max_its,w)

alpha = 10**(0)
g = softmax;
weight_history_2,cost_history_2 = optimizers.gradient_descent(g,alpha,max_its,w)
```

Construct misclassification counter.

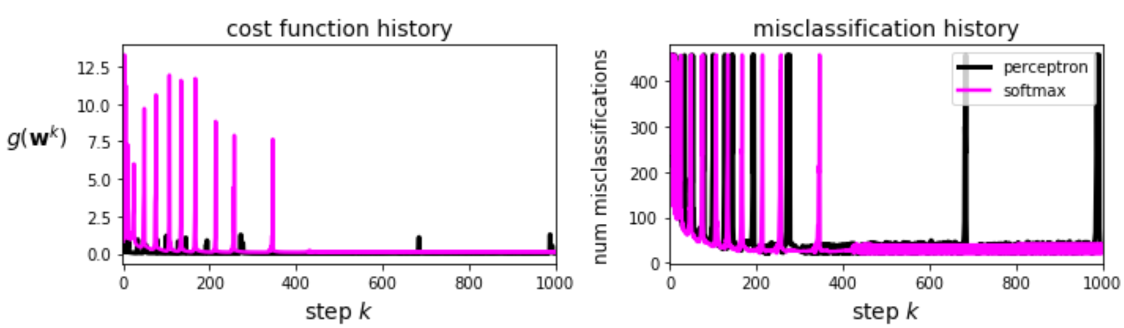
In [64]:
```
### cost functions ###
def counting_cost(w,x,y):
    # compute predicted labels
    y_hat = np.sign(model(x,w))

    # compare to true labels
    ind = np.argwhere(y != y_hat)
    ind = [v[1] for v in ind]

    cost = np.sum(len(ind))
    return cost
```

Create misclassification history for each run.

In [65]:
```
count_history_1 = [counting_cost(v,x,y) for v in weight_history_1]
count_history_2 = [counting_cost(v,x,y) for v in weight_history_2]
```

Plot cost and count histories.

In [66]:
```
# plot history
classif_plotter = superlearn.classification_static_plotter.Visualizer()

cost_histories = [cost_history_1,cost_history_2]
count_histories = [count_history_1,count_history_2]
classif_plotter.plot_cost_histories(cost_histories,count_histories,start = 0,poin
ts = False,labels = ['perceptron','softmax'])
```



In [67]:
```
best_percept = np.min(count_history_1)
best_soft = np.min(count_history_2)

print ('the smallest number of misclassifications provided by minimizing the perc
eptron ' + str(best_percept))
print ('the smallest number of misclassifications provided by minimizing the soft
max ' + str(best_soft))
```

```
the smallest number of misclassifications provided by minimizing the perceptron
20
the smallest number of misclassifications provided by minimizing the softmax 21
```

Exercise 6.14. Compare the efficacy of two-class cost functions II

In [29]:
```
# standard normalization function - with nan checker / filler in-er
def standard_normalizer(x):
    # compute the mean and standard deviation of the input
    x_means = np.nanmean(x,axis = 1)[:,np.newaxis]
    x_stds = np.nanstd(x,axis = 1)[:,np.newaxis]

    # check to make sure thta x_stds > small threshold, for those not
    # divide by 1 instead of original standard deviation
    ind = np.argwhere(x_stds < 10**(-2))
    if len(ind) > 0:
        ind = [v[0] for v in ind]
        adjust = np.zeros((x_stds.shape))
        adjust[ind] = 1.0
        x_stds += adjust

    # fill in any nan values with means
    ind = np.argwhere(np.isnan(x) == True)
    for i in ind:
        x[i[0],i[1]] = x_means[i[0]]

    # create standard normalizer function
    normalizer = lambda data: (data - x_means)/x_stds

    # create inverse standard normalizer
    inverse_normalizer = lambda data: data*x_stds + x_means

    # return normalizer
    return normalizer,inverse_normalizer
```

Below we load in a spam email dataset - [a description of which you can find here (https://archive.ics.uci.edu/ml/datasets/Spambase)](https://archive.ics.uci.edu/ml/datasets/Spambase). The input datapoints are stacked *column-wise* in this dataset, with the final row being the label of each point.

In [30]:
```
# data input
csvname = datapath + 'spambase_data.csv'
data = np.loadtxt(csvname,delimiter = ',')

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

In [31]:
```
ind0 = np.argwhere(y==-1)
ind1 = np.argwhere(y==+1)
print(len(ind0),len(ind1))
```

```
2788 1813
```

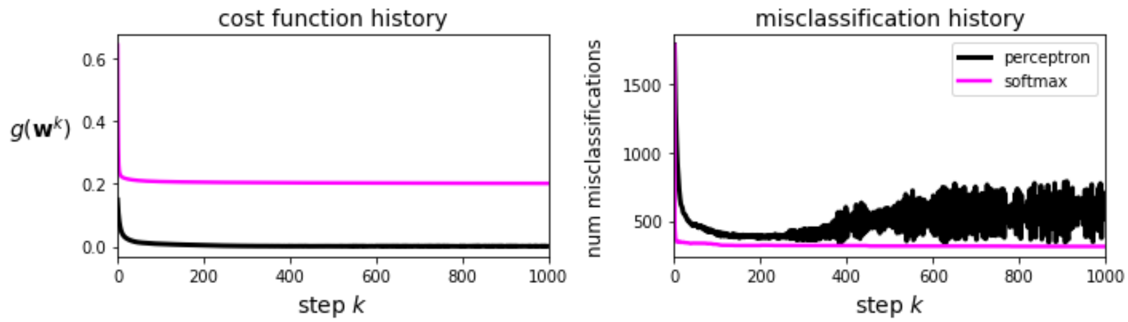
Standard normalize input.

In [152]:
```
normalizer,inverse_normalizer = standard_normalizer(x)
x = normalizer(x)
```

In [153]:
```
# setup data
N = x.shape[0]

# setup optimizer input (besides cost)
alpha = 10**(-1)
max_its = 1000
w = 0.1*np.random.randn(N+1,1)

# run gradient descent to minimize the Least Squares cost for linear regression
g = perceptron;
weight_history_1,cost_history_1 = optimizers.gradient_descent(g,alpha,max_its,w)

alpha = 10**(1)
g = softmax;
weight_history_2,cost_history_2 = optimizers.gradient_descent(g,alpha,max_its,w)
```

In [154]:
```
### cost functions ###
def counting_cost(w,x,y):
    # compute predicted labels
    y_hat = np.sign(model(x,w))

    # compare to true labels
    ind = np.argwhere(y != y_hat)
    ind = [v[1] for v in ind]

    cost = np.sum(len(ind))
    return cost

count_history_1 = [counting_cost(v,x,y) for v in weight_history_1]
count_history_2 = [counting_cost(v,x,y) for v in weight_history_2]
```

In [155]:
```
# plot history
classif_plotter = superlearn.classification_static_plotter.Visualizer()

cost_histories = [cost_history_1,cost_history_2]
count_histories = [count_history_1,count_history_2]
classif_plotter.plot_cost_histories(cost_histories,count_histories,start = 0,poin
ts = False,labels = ['perceptron','softmax'])
```



In [156]:
```
best_percept = np.min(count_history_1)
best_soft = np.min(count_history_2)

print ('the smallest number of misclassifications provided by minimizing the perc
eptron ' + str(best_percept))
print ('the smallest number of misclassifications provided by minimizing the soft
max ' + str(best_soft))
```

```
the smallest number of misclassifications provided by minimizing the perceptron
339
the smallest number of misclassifications provided by minimizing the softmax 315
```

In [1]:
```
(1 - 315/4601)
```

Out[1]:
```
0.9315366224733753
```

In [217]:
```
import copy
def confusion_matrix(y,y_hat):
    labels = np.unique(y)
    num_labels = len(labels)
    c = np.zeros((num_labels,num_labels))
    inds = []
    inds_hat = []
    for l in labels:
        ind0 = np.argwhere(y==l)
        if len(ind0) > 0:
            ind0 = [v[1] for v in ind0]
        inds.append(ind0)
        ind1 = np.argwhere(y_hat==l)
        if len(ind1) > 0:
            ind1 = [v[1] for v in ind1]
        inds_hat.append(ind1)

    for i in range(num_labels):
        ind0 = set(inds[i])
        for j in range(num_labels):
            ind1 = set(inds_hat[j])
            misclass = len(ind1.intersection(ind0))
            c[i,j] = misclass
    return c
```

In [222]:
```
ind_best = np.argmin(count_history_2)
w_best = weight_history_2[ind_best]
y_hat = np.sign(model(x,w_best))
c = confusion_matrix(y,y_hat)
print(c)
```

```
[[2664.  124.]
 [ 191. 1622.]]
```

Exercise 6.15. Credit check

In [428]:
```
# load in dataset
csvname = datapath + 'credit_dataset.csv'
data = np.loadtxt(csvname,delimiter = ',')
x = data[:-1,:]
y = data[-1:,:]
```

In [429]:
```
ind0 = np.argwhere(y==-1)
ind1 = np.argwhere(y==+1)
print(len(ind0),len(ind1))
```

```
300 700
```

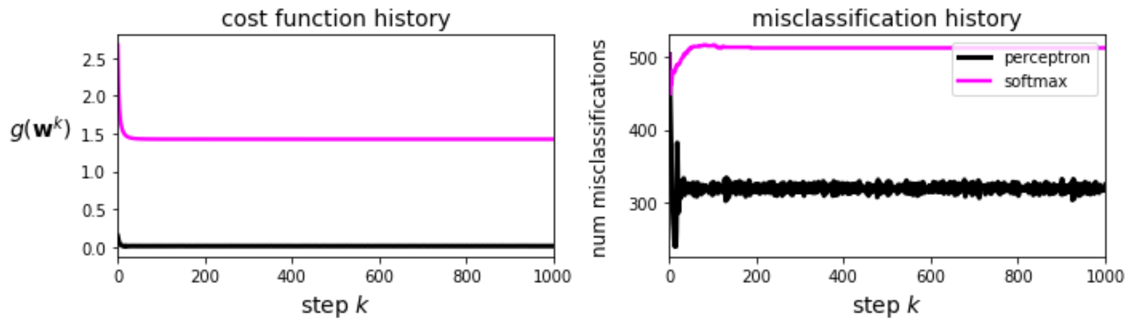
Standard normalize input.

In [430]:
```
normalizer,inverse_normalizer = standard_normalizer(x)
x = normalizer(x)
```

In [435]:
```
# setup data
N = x.shape[0]

# setup optimizer input (besides cost)
alpha = 10**(-1)
max_its = 1000
w = 0.1*np.random.randn(N+1,1)

# run gradient descent to minimize the Least Squares cost for linear regression
g = perceptron;
weight_history_1,cost_history_1 = optimizers.gradient_descent(g,alpha,max_its,w)
```

In [436]:
```
### cost functions ###
def counting_cost(w,x,y):
    # compute predicted labels
    y_hat = np.sign(model(x,w))

    # compare to true labels
    ind = np.argwhere(y != y_hat)
    ind = [v[1] for v in ind]

    cost = np.sum(len(ind))
    return cost

count_history_1 = [counting_cost(v,x,y) for v in weight_history_1]
count_history_2 = [counting_cost(v,x,y) for v in weight_history_2]
```

In [437]:
```
# plot history
classif_plotter = superlearn.classification_static_plotter.Visualizer()

cost_histories = [cost_history_1,cost_history_2]
count_histories = [count_history_1,count_history_2]
classif_plotter.plot_cost_histories(cost_histories,count_histories,start = 0,poin
ts = False,labels = ['perceptron','softmax'])
```



In [438]:
```
best_percept = np.min(count_history_1)
best_percept_acc = (1 - best_percept/y.size)

print ('the smallest number of misclassifications provided by minimizing the perc
eptron ' + str(best_percept))
print ('best acc by minimizing the perceptron ' + str(best_percept_acc))
```

```
the smallest number of misclassifications provided by minimizing the perceptron
239
best acc by minimizing the perceptron 0.761
the smallest number of misclassifications provided by minimizing the softmax 449
best acc by minimizing the softmax 0.5509999999999999
```

In [439]:
```
ind_best = np.argmin(count_history_1)
w_best = weight_history_2[ind_best]
y_hat = np.sign(model(x,w_best))
c = confusion_matrix(y,y_hat)
print(c)
```

```
[[285.  15.]
 [466. 234.]]
```

In [426]:
```
y.size
```

Out[426]:
```
1000
```

Exercise 6.16. Weighted classification and balanced accuracy

In [148]:
```
def balanced_accuracy(w,x,y):
    # make predictions
    y_hat = np.sign(model(x,w))
    print(y_hat.shape)

    # press predictions against real results
    ind0 = np.argwhere(y == -1)
    ind0 = [v[1] for v in ind0]
    num0 = len(ind0)

    ind = np.argwhere(np.abs(y[:,ind0] - y_hat[:,ind0]) > 0)
    cost0 = len(ind)

    ind1 = np.argwhere(y == +1)
    ind1 = [v[1] for v in ind1]
    num1 = len(ind1)
    ind = np.argwhere(np.abs(y[:,ind1] - y_hat[:,ind1]) > 0)
    cost1 = len(ind)

    # compute accuracies
    acc0 = 1 - cost0/num0
    acc1 = 1 - cost1/num1
    return (acc0 + acc1)/2
```

In [122]:
```
# 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:,:]
```

In [123]:
```
ind0 = np.argwhere(y==-1)
ind1 = np.argwhere(y==+1)
print(len(ind0),len(ind1))
```

```
50 5
```

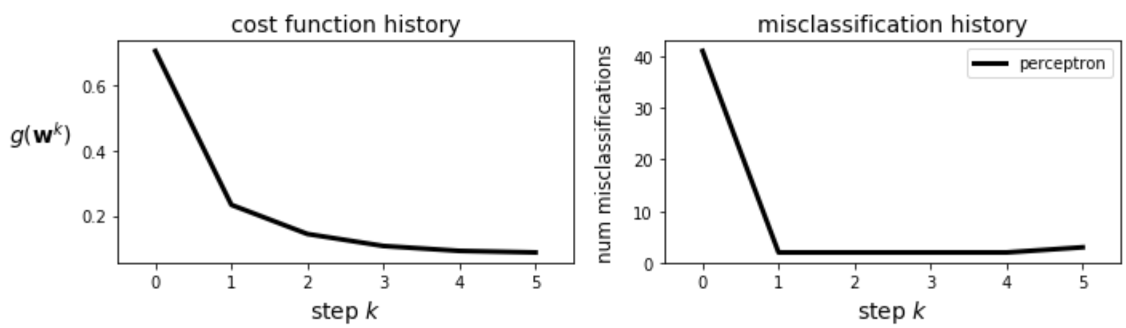
In [139]:
```
betas = np.array([1.0,5.0])

# define sigmoid function
def sigmoid(t):
    return 1/(1 + np.exp(-t))

# the convex cross-entropy cost function
def weighted_softmax(w,betas):
    # compute sigmoid of model
    a = sigmoid(model(x,w))

    # compute cost of label 0 points
    ind = np.argwhere(y == -1)[:,1]
    cost = -betas[0]*np.sum(np.log(1 - a[:,ind]))

    # add cost on label 1 points
    ind = np.argwhere(y==+1)[:,1]
    cost -= betas[1]*np.sum(np.log(a[:,ind]))

    # compute cross-entropy
    return cost/y.size
```

In [140]:
```
betas = np.array([1.0,1.0])
softmax = lambda w,betas = betas: weighted_softmax(w,betas)
```

In [141]:
```
# setup data
N = x.shape[0]

# setup optimizer input (besides cost)
max_its = 5
w = 0.1*np.random.randn(N+1,1)

# run gradient descent to minimize the Least Squares cost for linear regression
g = softmax;
weight_history_1,cost_history_1 = optimizers.newtons_method(g,max_its,w)
```

In [142]:
```
### cost functions ###
def counting_cost(w,x,y):
    # compute predicted labels
    y_hat = np.sign(model(x,w))

    # compare to true labels
    ind = np.argwhere(y != y_hat)
    ind = [v[1] for v in ind]

    cost = np.sum(len(ind))
    return cost

count_history_1 = [counting_cost(v,x,y) for v in weight_history_1]
```

In [143]:
```
# plot history
classif_plotter = superlearn.classification_static_plotter.Visualizer()

cost_histories = [cost_history_1]
count_histories = [count_history_1]
classif_plotter.plot_cost_histories(cost_histories,count_histories,start = 0,poin
ts = False,labels = ['perceptron','softmax'])
```

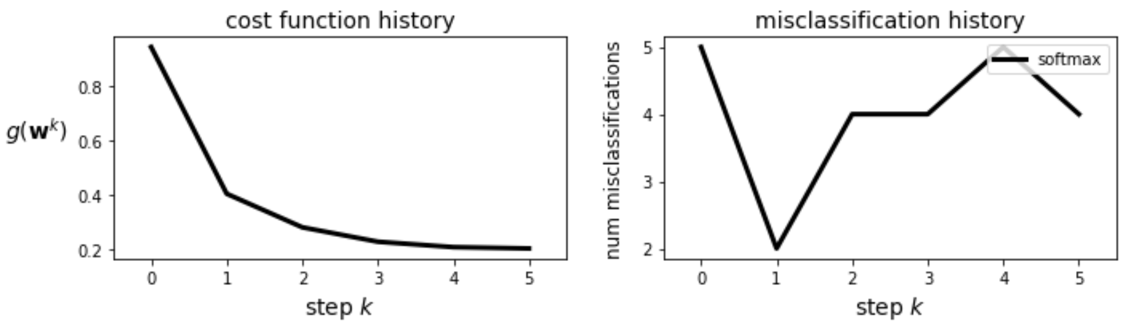


In [149]:
```
w_best = weight_history_1[-1]
best_soft_count = count_history_1[-1]
best_soft_acc = (1 - best_soft_count/y.size)
best_balanced = balanced_accuracy(w_best,x,y)

print ('the smallest number of misclassifications provided by minimizing the soft
mxa ' + str(best_soft_count))
print ('best acc by minimizing the softmax ' + str(best_soft_acc))
print ('best balanced acc by minimizing the softmax ' + str(best_balanced))
```

```
(1, 55)
the smallest number of misclassifications provided by minimizing the softmxa 3
best acc by minimizing the softmax 0.9454545454545454
best balanced acc by minimizing the softmax 0.79
```

$\beta = 5$

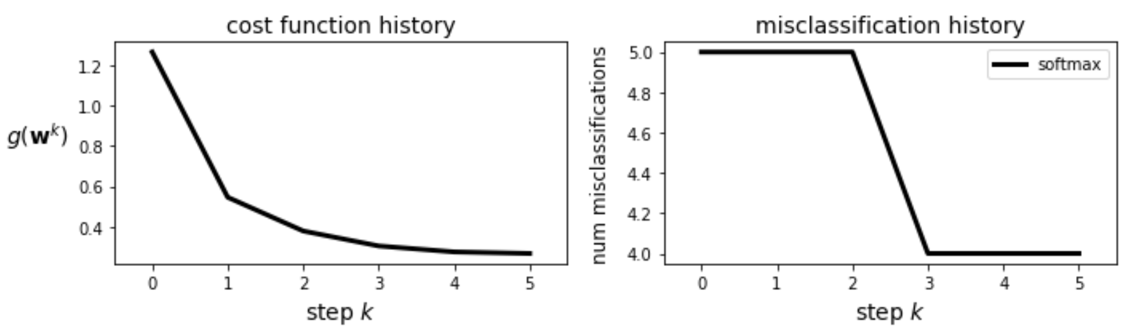
In [150]:
```
# setup data
N = x.shape[0]

# setup optimizer input (besides cost)
max_its = 5
w = 0.1*np.random.randn(N+1,1)

# run gradient descent to minimize the Least Squares cost for linear regression
betas = np.array([1.0,5.0])
softmax = lambda w,betas = betas: weighted_softmax(w,betas)
g = softmax;
weight_history_2,cost_history_2 = optimizers.newtons_method(g,max_its,w)
count_history_2 = [counting_cost(v,x,y) for v in weight_history_2]

# plot history
classif_plotter = superlearn.classification_static_plotter.Visualizer()

cost_histories = [cost_history_2]
count_histories = [count_history_2]
classif_plotter.plot_cost_histories(cost_histories,count_histories,start = 0,poin
ts = False,labels = ['softmax'])

w_best = weight_history_2[-1]
best_soft_count = count_history_2[-1]
best_soft_acc = (1 - best_soft_count/y.size)
best_balanced = balanced_accuracy(w_best,x,y)

print ('the smallest number of misclassifications provided by minimizing the soft
mxa ' + str(best_soft_count))
print ('best acc by minimizing the softmax ' + str(best_soft_acc))
print ('best balanced acc by minimizing the softmax ' + str(best_balanced))
```




```
(1, 55)
the smallest number of misclassifications provided by minimizing the softmxa 4
best acc by minimizing the softmax 0.9272727272727272
best balanced acc by minimizing the softmax 0.87
```

$\beta = 10$

In [151]:
```
# setup data
N = x.shape[0]

# setup optimizer input (besides cost)
max_its = 5
w = 0.1*np.random.randn(N+1,1)

# run gradient descent to minimize the Least Squares cost for linear regression
betas = np.array([1.0,10.0])
softmax = lambda w,betas = betas: weighted_softmax(w,betas)
g = softmax;
weight_history_3,cost_history_3 = optimizers.newtons_method(g,max_its,w)
count_history_3 = [counting_cost(v,x,y) for v in weight_history_3]

# plot history
classif_plotter = superlearn.classification_static_plotter.Visualizer()

cost_histories = [cost_history_3]
count_histories = [count_history_3]
classif_plotter.plot_cost_histories(cost_histories,count_histories,start = 0,poin
ts = False,labels = ['softmax'])

w_best = weight_history_3[-1]
best_soft_count = count_history_3[-1]
best_soft_acc = (1 - best_soft_count/y.size)
best_balanced = balanced_accuracy(w_best,x,y)

print ('the smallest number of misclassifications provided by minimizing the soft
mxa ' + str(best_soft_count))
print ('best acc by minimizing the softmax ' + str(best_soft_acc))
print ('best balanced acc by minimizing the softmax ' + str(best_balanced))
```




```
(1, 55)
the smallest number of misclassifications provided by minimizing the softmxa 4
best acc by minimizing the softmax 0.9272727272727272
best balanced acc by minimizing the softmax 0.96
```