In [1]:
```
import numpy as np
from scipy.sparse import csc_matrix
from scipy.sparse.linalg import eigs

edges_file = open('wisconsin_edges.csv', "r")
nodes_file = open('wisconsin_nodes.csv', "r")

# create a dictionary where nodes_dict[i] = name of wikipedia page
nodes_dict = {}
for line in nodes_file:
    nodes_dict[int(line.split(',',1)[0].strip())] = line.split(',',1)[1].strip()

node_count = len(nodes_dict)

# create adjacency matrix
A = np.zeros((node_count, node_count))
for line in edges_file:
    from_node = int(line.split(',')[0].strip())
    to_node = int(line.split(',')[1].strip())
    A[to_node, from_node] = 1.0

## Add code below to (1) prevent traps and (2) find the most important pages
# Hint -- instead of computing the entire eigen-decomposition of a matrix X using
# s, E = np.linalg.eig(A)
# you can compute just the first eigenvector with:
# s, E = eigs(csc_matrix(A), k = 1)
```

1 a)

In [2]:
```
# make a new Array to hold the normalized matrix
Anew = np.zeros((node_count, node_count))

# remove traps by adding 0.001
for i in range(A.shape[0]):
    for j in range(A.shape[1]):
        A[i, j] = A[i, j] + 0.001

# normalize
for k in range(A.shape[1]):
    norm = np.sum(A[:,k])
    Anew[:,k] = (A[:,k])/norm

# compute Eigenvectors
s, E = eigs(csc_matrix(Anew), k = 1)
E = np.abs(E)
E = E.flatten()

# sort
E_sort = np.argsort(E)
```

In [3]:
```
# print sort, take last and third last elements and find their names
print(E_sort)
```

```
[2041 4298 3874 ... 1345 2312 5089]
```

1 b)

In [4]:
```
print("5089 has page title \"Wisconsin\"")
```

```
5089 has page title "Wisconsin"
```

1 c)

In [5]:
```
print("1345 has page title \"Madison, Wisconsin\"")
```

```
1345 has page title "Madison, Wisconsin"
```

In [ ]:

2.a) We assume that the labels are either 1 or -1 dependending on the result of the classifier.

$$\hat{y_i} = 1 \quad \text{if } x_i^T w > 0$$

$$\hat{y_i} = -1 \quad \text{if } x_i^T w < 0$$

Thus, $\hat{y_i} = \text{sign}(x_i^T w)$

When a point is easy to classify, if it is far from the decision boundary, then it will be correctly classify.

$$\hat{y_i} = \text{sign}(x_i^T w) = y_i$$

Given,

$$l_i(w) = \log(1 + e^{-y_i x_i^T w})$$

as $y_i = \text{sign}(x_i^T w)$

Thus, $l_i(w) = \log\left(1 + \frac{1}{e^{|x_i^T w|}}\right)$

As the point is easy to classify. $|x_i^T w|$ is very large and thus

$$l_i(w) = \log\left(1 + \underbrace{\frac{1}{e^{|x_i^T w|}}}_{\to 1}\right) \text{ becomes small.}$$

thus log tends to 0.

b) Given,

$$f(w) = \underbrace{\sum_{i=1}^{n} \log(1 + e^{-y_i x_i^T w})}_{l_1(w)} + \underbrace{\lambda \|w\|_2^2}_{r(w)}$$

If we take the derivative,

$$\frac{\partial f(w)}{\partial w_j} \left(\sum_{i=1}^{n} \log(1 + e^{-y_i x_i^T w}) + \lambda \|w\|_2^2 \right)$$

where w_j is an element from vector w

Thus we have,

$$\sum_{i=1}^{n} \frac{d(l_1(w))}{\partial w_j} + \lambda \frac{\partial(\|w\|_2^2)}{\partial w_j}$$

as the elements are seperable.

Now, $\|w\|_2^2 = w_1^2 + w_2^2 + w_3^2 + \dots + w_m^2$

$$\frac{\partial \|w\|_2^2}{\partial w_j} = 2w_j$$

Thus, $\nabla_w \|w\|_2^2 = \begin{bmatrix} 2w_1 \\ 2w_2 \\ 2w_3 \\ \vdots \\ 2w_n \end{bmatrix} = 2w$

Let $x_i^T w = x_{i1} w_1 + x_{i2} w_2 + \dots + x_{in} w_n$

$$\frac{\partial l_i(w)}{\partial w_j} = \frac{1}{1+e^{-y_i x_i^T w}} \left(e^{-y_i x_i^T w}\right)\left(-y_i x_{ij}\right)$$

$$l_i(w) = \log\left(1+e^{-y_i x_i^T w}\right)$$

$$e^{l_i(w)} = 1 + e^{-y_i x_i^T w}$$

$$\frac{\partial l_i(w)}{\partial w_j} = \frac{1}{e^{l_i(w)}}\left(e^{l_i(w)} - 1\right)\left(-y_i x_{ij}\right)$$

$$\nabla_w l_i(w) = \left(1 - \frac{1}{e^{l_i(w)}}\right)\left(-y_i x_{ij}\right)$$

$$= -y_i \left(1 - \frac{1}{e^{l_i(w)}}\right) x_i$$

$$f(w) = \sum_{i=1}^{N} l_i(w) + \lambda \|w\|_2^2 \ \text{so}$$

$$\nabla_w f(w) = \sum_{i=1}^{N} -y_i\left(1 - \frac{1}{e^{l_i(w)}}\right) x_i + \lambda(2w)$$

$$= \sum_{i=1}^{N} -y_i\left(1 - \frac{1}{e^{\log(1+e^{-y_i x_i^T w})}}\right) x_i + \lambda(2w)$$

P.T.O

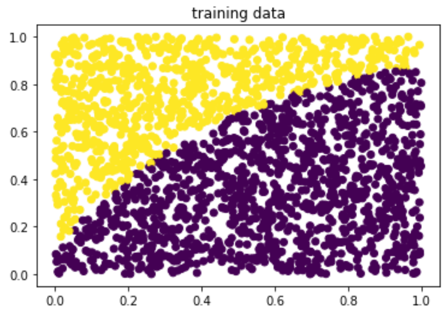
$$= \sum_{i=1}^{n} (-y_i)\left(1 - \frac{1}{1+e^{-y_i x_i^T w}}\right) x_i + \lambda(2w)$$

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

pkl_file = open('classifier_data.pkl', 'rb')
x_train, y_train = pickle.load(pkl_file)

n_train = np.size(y_train)

plt.scatter(x_train[:,0],x_train[:,1], c=y_train[:,0])
plt.title('training data')
plt.show()
```



In [2]:
```
def regularized_log_loss_function(X, y, w, lam):
    """
    function takes a matrix X with each column a feature vector, a vector y of labels,
    takes a w, and returns a scalar representing the value of function in 2b
    """
    m,n = np.shape(X)
    res = 0
    for i in range(n):
        res += np.log(1+np.exp(-y[i]*X[:,[i]].T@w))
    res += lam*w.T@w
    return res[0][0]

def gradient(X,y,w,lam,addDimension):
    """
    function takes a matrix X with each column a feature vector, a vector y of labels,
    takes a w, a lamda parameter which happens to be one in this whole program and a
    boolean value which judges if we need to append the dimensions of the matrix.
    There are two cases, one requires appending and one does not. This variable
    facilitates that result.
    """
    if(addDimension==True):
        grad = np.zeros((len(X[0])+1,1))
    else:
        grad = np.zeros((len(X[0]),1))

    # grad = np.zeros((len(X[0]),1))

    for i in range(len(y)):
        yi = y[i][0]
        xi = X[i]
        if(addDimension==True):
            xiT = np.append(xi, np.array([[1]]))
        else:
            xiT = xi


        curr = -1*yi*(1-(1/(1+np.exp(-1*yi*(xiT@w)))))*xiT.reshape((len(xiT),1))
        grad = grad + curr
    return (grad + 2*lam*w)

def graddescent(X,y,tau,w_init,it,addDimension=True):
    """
    compute 10 iterations of gradient descent starting at w1
    w_{k+1}= w_k - tau*X'*(X*w_k - y)
    There is a boolean value which judges if we need to append the dimensions of the matrix. There
    are two cases, one requires appending and one does not. This variable facilitates that result.
    """
    W = np.zeros((w_init.shape[0],it))
    W[:,[0]] = w_init
    for k in range(it-1):
        #X.T @ (X @ W[:,[k]] - y)
        W[:,[k+1]] = W[:,[k]] - tau * gradient(X,y,W[:,[k]],1,addDimension)
    return W
```

2 c)

```
In [3]: w_init = np.array([[1],[1],[1]])
        tau = 0.006
        w = graddescent(x_train, y_train, tau, w_init, 95, True)
        w = w[:,len(w[0])-1].reshape(len(w),1)
        print("w=\n", w)
```

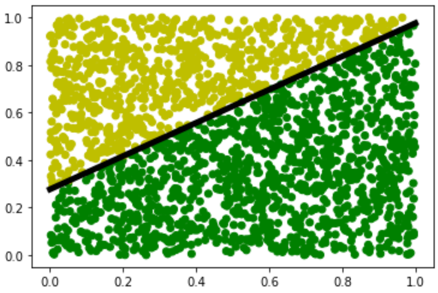
```
w=
 [[-6.44281082]
 [ 9.21295439]
 [-2.55074792]]
```

2 d)

```
In [4]: x_train = np.hstack((x_train, np.ones((len(x_train),1))))
        yhat = np.sign(x_train@w)
        plt.scatter(x_train[:,0],x_train[:,1],color=['g' if i==-1 else 'y' for i in yhat[:,0]])
        error_vector = [0 if i[0]==i[1] else 1 for i in np.hstack((yhat, y_train))]
        errors =  sum(error_vector)
        print("Rate of error: ", errors/len(error_vector))
        slope = -w[0,0]/w[1,0]
        y_int = -w[2,0]/w[1,0]
        plt.plot([0,1], [y_int, y_int + slope], linewidth = 5, color='black')
```

```
Rate of error:  0.035
```

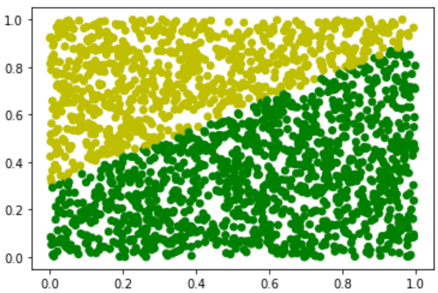
```
Out[4]: [<matplotlib.lines.Line2D at 0x266fc621550>]
```



2 e)

```
In [5]: wLS = np.linalg.inv(x_train.T@x_train+np.eye(len(x_train.T)))@x_train.T@y_train
        print("weights of least squares: ", wLS)
        yhat = np.sign(x_train@wLS)
        plt.scatter(x_train[:,0], x_train[:,1], color=['g' if i==-1 else 'y' for i in yhat[:,0]])
        error_vector = [0 if i[0]==i[1] else 1 for i in np.hstack((yhat, y_train))]
        errors = sum(error_vector)
        print("Rate of error: " )
        print(errors/len(error_vector))
```

```
weights of least squares:  [[-1.39655874]
 [ 2.29068083]
 [-0.6846757 ]]
Rate of error:
0.0455
```



The error rate is greater (0.0455>0.035) than the one trained by logistic loss. The performance however is similiar.

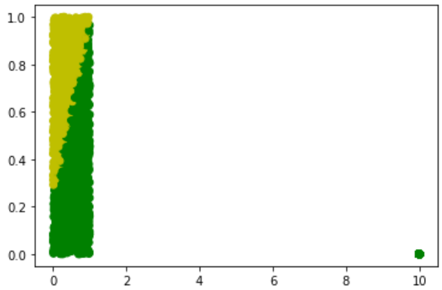
2 f)

In [6]:
```
#add new data
pkl_file = open('classifier_data.pkl', 'rb')
x_train, y_train = pickle.load(pkl_file)
x_train = np.hstack((x_train, np.ones((len(x_train),1))))
x_train_mod = x_train
y_train_mod = y_train
for i in range(1000):
    x_train_mod = np.vstack((x_train_mod, np.array([[10,0,1]])))
    y_train_mod = np.vstack((y_train_mod, np.array([[-1]])))

#logistic loss classifier
w_init = np.array([[1],[1],[1]])
tau = 0.006
#x_train_mod = np.vstack((x_train_mod, np.ones((1,len(x_train_mod)))))
w = graddescent(x_train_mod, y_train_mod, tau, w_init, 95,False)
w = w[:,len(w[0])-1].reshape(len(w),1)

yhat = np.sign(x_train_mod@w)
plt.scatter(x_train_mod[:,0],x_train_mod[:,1],color=['g' if i==-1 else 'y' for i in yhat[:,0]])
error_vector = [0 if i[0]==i[1] else 1 for i in np.hstack((yhat, y_train_mod))]
errors =  sum(error_vector)
print("Rate of error: ", errors/len(error_vector))
```

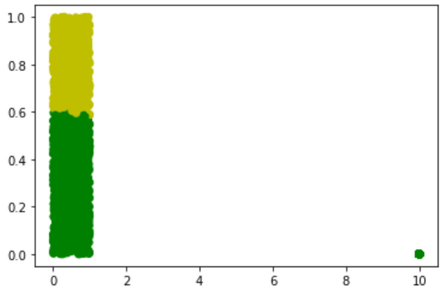
```
Rate of error:  0.024666666666666667
```



The logistic classifier handles points which are very simple to classify, very well, and even sees the error rate go down.

In [7]:
```
wLS = np.linalg.inv(x_train_mod.T@x_train_mod+np.eye(len(x_train_mod.T)))@x_train_mod.T@y_train_mod
print("weights of least squares: ", wLS)
yhat = np.sign(x_train_mod@wLS)
plt.scatter(x_train_mod[:,0], x_train_mod[:,1], color=['g' if i==-1 else 'y' for i in yhat[:,0]])
error_vector = [0 if i[0]==i[1] else 1 for i in np.hstack((yhat, y_train_mod))]
errors = sum(error_vector)
print("Rate of error: " )
print(errors/len(error_vector))
```

```
weights of least squares:  [[ 0.03355812]
 [ 2.25926458]
 [-1.36073308]]
Rate of error:
0.11933333333333333
```



The square error classifier does not handle the easy-to-classify points very well, and thus sees a huge error rate. It places a large importance on the distance from the boundary. The newly added points in the mod data points, thus have a huge effect on the decision boundary. Thus the error rate is high as we can see.

In [ ]: