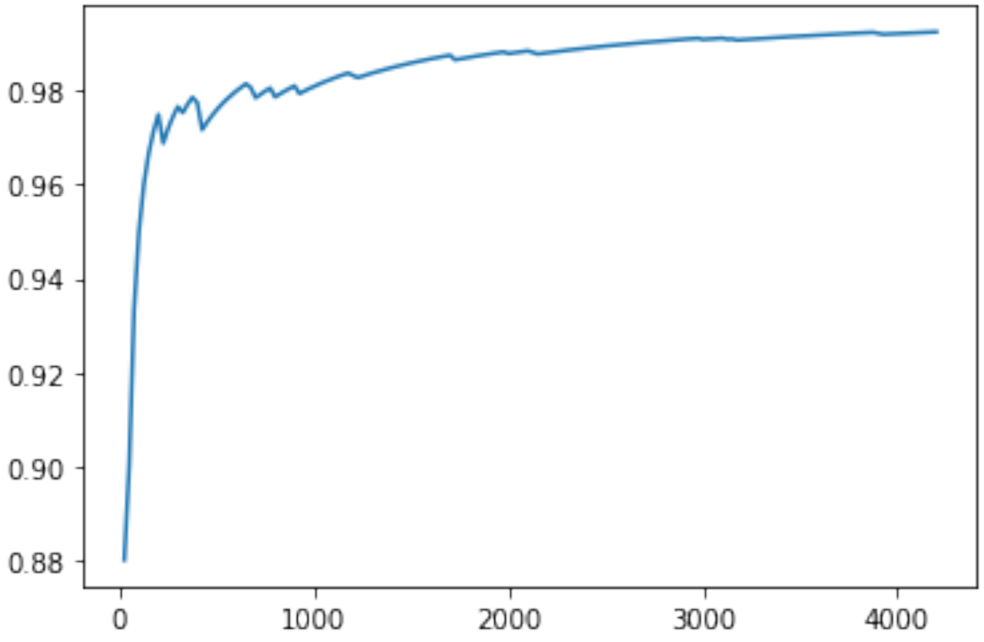
In [52]:

```
#Question 1
import os
import numpy as np
import matplotlib.pyplot as plt

DIM = (28,28) #dimensions of the image
def load_image_files(n, path="images/"):
    # helper file to help load the images
    # returns a list of numpy vectors
    images = []
    for f in os.listdir(os.path.join(path,str(n))): # read files
in the path
        p = os.path.join(path,str(n),f)
        if os.path.isfile(p):
            i = np.loadtxt(p)
            assert i.shape == DIM # just check the dimensions he
re
            # i is loaded as a matrix, but we are going to flatt
en it
            # into a single vector
            images.append(i.flatten())
    return images

#Load up these image files
A = load_image_files(0)
B = load_image_files(1)


N = len(A[0]) # the total size

assert N == DIM[0] * DIM[1] # just check our sizes to be sure

#set up random initial weights
weights = np.random.normal(0, 1, size=N)
```

In [65]:

```
#citation for starter code: Sec103_109_Discussion_7
#w = weights, x = image vector
def perceptron_helper(weights, image):
    #1. calculate N (dot product)
```

```python
    if np.dot(weights, image) >= 0:
        return 1
    else:
        return 0

#define needed lists
weights = np.random.normal(0,1,size=N)
indices = np.arange(len(A) + len(B))
np.random.shuffle(indices)
images = A + B
accuracy = []
iterations = []

#define needed variables
output = 0
actual = 0
correct = 0
images_seen = 0
x = 1
diff_in_accuracy = 1
prev_accuracy = 0
index = 0

while images_seen < len(indices)/3:
    index = indices[images_seen]
    # run perceptron algorithm
    output = perceptron_helper(weights, images[index])

    # determine what the "actual" value is
    if index < len(A):
        actual = 0
    else:
        actual = 1

    #update weights accordingly
    if output == 1 and actual == 0:
        weights = weights - images[index]
    elif output == 0 and actual == 1:
        weights = weights + images[index]
    else:
        correct += 1
    images_seen += 1
    if (images_seen%25 == 0):
        current_accuracy = correct/images_seen
        accuracy.append(current_accuracy)
        diff_in_accuracy = current_accuracy - prev_accuracy
        prev_accuracy = current_accuracy
        iterations.append(25 * x)
        x += 1
plt.plot(iterations, accuracy)
plt.show()
```

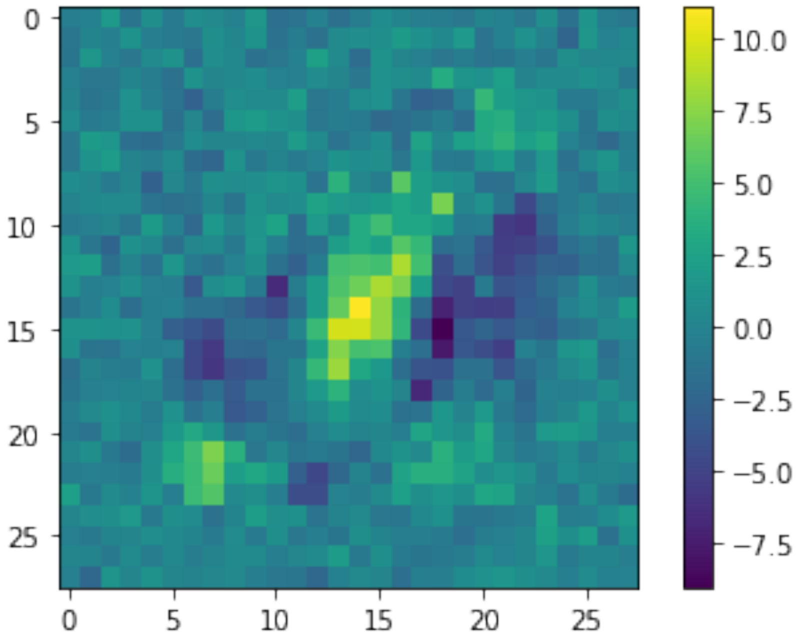


Question 2

My solution in Q1 does converge on 100% accuracy with enough trials. Because distinguishing between a 0 and a 1 is a linearly separable problem, my perceptron algorithm was able to find a linear decision boundary to categorize the handwritten digits with close to 100% accuracy. This is due to the fact that the perceptron algorithm can provably find this perfect line that divids to categories and because I was able to feed it an adequate amount of characters for it to find the category boundary for this particular problem.

In [5]:

```
# Question 3
plt.imshow(np.reshape(weights, (28, 28)))
plt.colorbar()
plt.show()
```



As you can see, the image produced by plt.imshow() of my reshaped trained weights is a faint looking 0 with bright yellow in the center. Large negative values correspond to weights that are highly trained to recognized 0s, while large positive values correspond to the weights that are highly trained to recognize 1s. Numbers near 0 represent weights that are not highly trained one way or the other, and are ambiguous. They correspond to "white space" on the page where there is not ink, while the highly positive/negative numbers correspond to where there is likely to be ink on the page. The negative numbers somewhat resemble the shape of a zero, since whenever the algorithm incorrectly guessed, those weights became more negative. And vice versa for the positive weights.

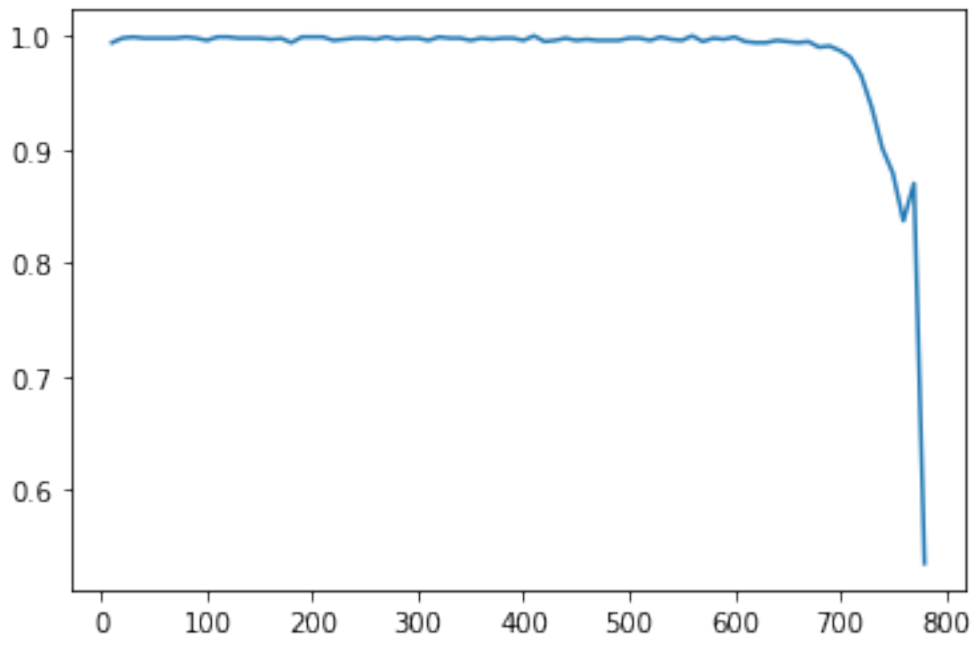
Question 4. My expectations: Setting the elements of the weight vector closest to zero to actually be zero likely does not change the results of this algorithm for MOST weights, only those that are largely positive or negative. What I can observe after running the algorithm is that pretty much the ~80 or so most positive/negative weights are the most diagnostic about determining whether the character is a 0 or 1. Meaning that 80/780 weights actually have an important say as to which number the character is.

In [66]:

```
def weights_changer(weights):
    #changes the 10 closest weights to 0 to be actually 0
    #store 10 closest weights to 0 and their corresponding index
in tuple
    tenClosest = []
    absvalWeights = np.absolute(weights)
    iterations = len(weights)
    i = 0
    while len(tenClosest) < 10 and i < 784:
        if absvalWeights[i] != 0:
            tenClosest.append((absvalWeights[i], i))
        i += 1
    tenClosest.sort()

    while i < iterations:
        if absvalWeights[i] != 0 and absvalWeights[i] < tenClose
st[9][0]:
            tenClosest[9] = (absvalWeights[i], i)
            tenClosest.sort()
        i += 1

    for t in tenClosest:
        weights[t[1]] = 0

    return weights

#define needed lists
indices = np.arange(len(A) + len(B))
np.random.shuffle(indices)
images = A + B
accuracy = []
iterations_plot = []
```

```python
#define needed variables
output = 0
actual = 0
correct = 0
images_seen = 0
x = 0
index = 0
iterations = 10
weight_indices = []
weight_thresh = 0

while iterations < 781:
    np.random.shuffle(indices)
    correct = 0
    images_seen = 0

    weights_changer(weights)

    for x in range(1000):
        index = indices[x]
        # run perceptron algorithm
        output = perceptron_helper(weights, images[index])

        # determine what the "actual" value is
        if index < len(A):
            actual = 0
        else:
            actual = 1

        #update weights accordingly
        if output == 1 and actual == 0:
            correct += 0
        elif output == 0 and actual == 1:
            correct += 0
        else:
            correct += 1
        images_seen += 1

    current_accuracy = correct/images_seen
    accuracy.append(current_accuracy)
    iterations_plot.append(iterations)
    iterations += 10
```

```
plt.plot(iterations_plot, accuracy)
plt.show()
```



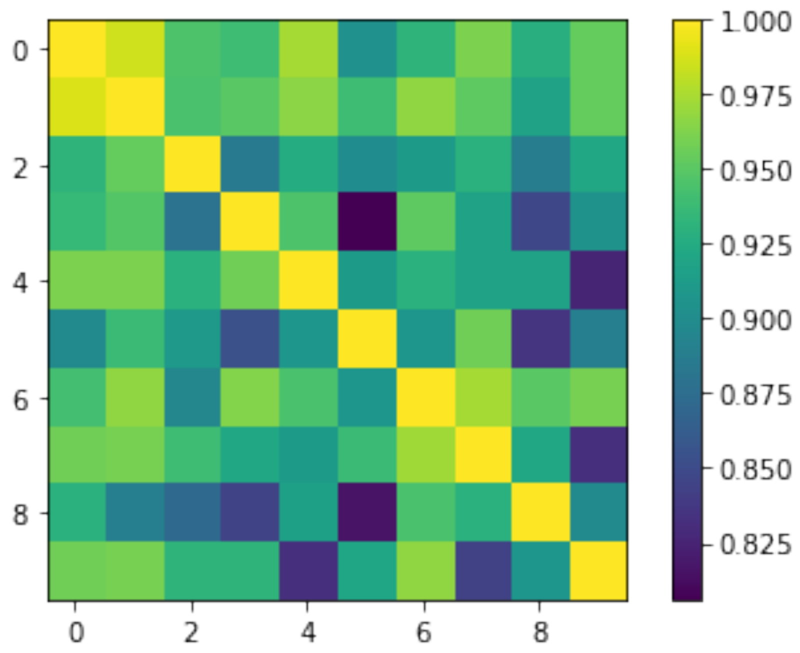
1. [20pts, SOLO] Next show a matrix of the classification accuracy of each pair of digits after enough training. Make this a plot (with colors for accuracy rather than numbers). Does it match your intuitions about which pairs should be easy vs. hard? Why or why not?

In [76]:

```
#matrix of classification accuracy
#  0  1  2  3
#0 1  x  y  z
#1 p  1 ...
#2       1
#3          1
#

size = 10
matrix = [[0 for i in range(size)] for j in range(size)]
i = 0
j = 0
indices = 0
images = []
accuracy = []

#define needed variables
output = 0
```

```
actual = 0

correct = 0
images_seen = 0
index = 0

for i in range(10):
    for j in range(10):
        if i == j:
            matrix[i][j] = 1
            print("matrix updated")
        #calculate the accuracy
        else:
            #Load up these image files
            A = load_image_files(i)
            B = load_image_files(j)
            N = len(A[0]) # the total size
            assert N == DIM[0] * DIM[1] # just check our sizes t
o be sure

            #set up random initial weights
            weights = np.random.normal(0, 1, size=N)

            #set up indeces
            indices = np.arange(len(A) + len(B))
            np.random.shuffle(indices)
            #collect all images
            images = A + B

            #initialize accuracy/iterations
            accuracy = []
            iterations = []
            images_seen = 0
            correct = 0

            while images_seen < 500:
                index = indices[images_seen]
                # run perceptron algorithm
                output = perceptron_helper(weights, images[index
])

                # determine what the "actual" value is
                if index < len(A):
                    actual = 0
                else:
                    actual = 1

                #update weights accordingly
                if output == 1 and actual == 0:
                    weights = weights - images[index]
                elif output == 0 and actual == 1:
                    weights = weights + images[index]
                else:
                    correct += 1
                images_seen += 1

            matrix[i][j] = correct/images_seen
            print("matrix updated")
print(matrix)
plt.imshow(matrix)
plt.colorbar()
plt.show()
```

```
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
matrix updated
[[1, 0.986, 0.946, 0.94, 0.974, 0.904, 0.932, 0.962,
0.928, 0.954], [0.99, 1, 0.944, 0.95, 0.966, 0.94, 0
.968, 0.952, 0.918, 0.954], [0.932, 0.954, 1, 0.886,
0.926, 0.9, 0.912, 0.93, 0.888, 0.922], [0.936, 0.94
8, 0.88, 1, 0.946, 0.806, 0.952, 0.918, 0.848, 0.906
], [0.962, 0.962, 0.93, 0.958, 1, 0.912, 0.93, 0.918
, 0.918, 0.826], [0.898, 0.938, 0.91, 0.854, 0.908,
1, 0.908, 0.958, 0.836, 0.89], [0.942, 0.968, 0.896,
0.964, 0.944, 0.908, 1, 0.974, 0.95, 0.96], [0.958,
0.96, 0.94, 0.922, 0.912, 0.938, 0.972, 1, 0.922, 0.
832], [0.93, 0.89, 0.872, 0.846, 0.916, 0.816, 0.944
, 0.93, 1, 0.898], [0.958, 0.96, 0.932, 0.932, 0.832
, 0.92, 0.968, 0.844, 0.908, 1]]
```



The table above indicates that the hardest pairs to distinguish between were numbers that can appear to be the or extremeley similar (especially when hand-written). For example, 3 and 5, 4 and 9, 5 and 8 and 8 and 3. This matches my intuition, since I would think that numbers that are more starkly contrasting in how they appear on paper would be easier for the weights to attune to, making the perceptron algorithm guess more accurately. In contrast, it would be more difficult to recognize numbers that have a lot of ink overlap when on paper, which is what my results seem to suggest.

```
In [ ]:
```