Problem #1

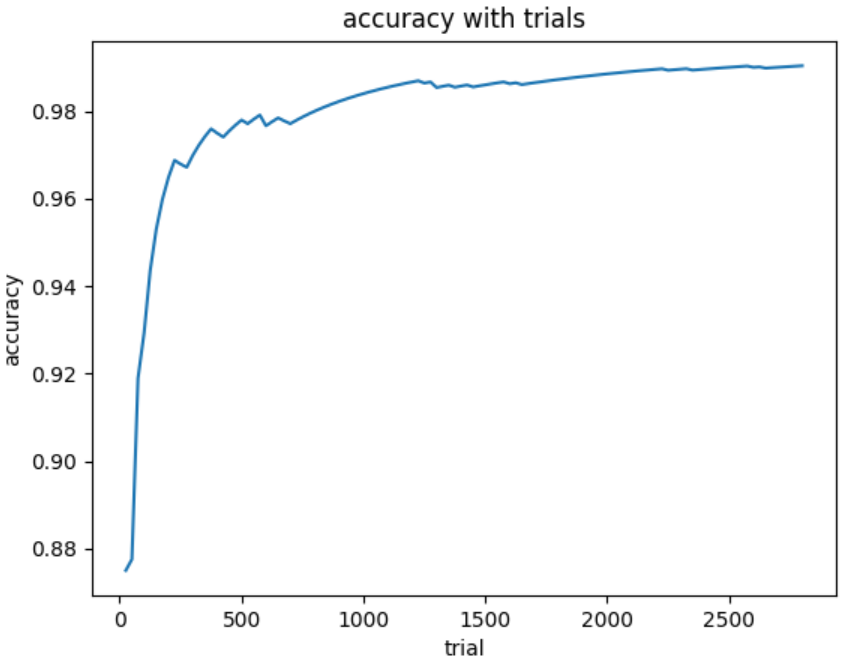
Code

```
def Perceptron(d1, d2, big_list, weights):

    # we will keep track of how many the perceptron gets correct, and
    # also record our accuracy level every 25 images
    correct = 0
    accuracy_values = []


    for i in range(0, len(big_list)):

        # if our accuracy level has crossed 99%, break out
        if (i > 25) and (correct / i) > 0.99:
            stop = i
            break

        image = big_list[i]
        s = numpy.dot(image[0], weights) # dot the inputs with the weights
        compare = image[1]

        # y is the result of our step function
        y = 0
        if s < 0:
            y = 0
        else:
            y = 1

        # fix the weights depending on if y matches compare (the real digit)
        # add 1 to correct if it got it right
        if (y == 0) and (compare == 1):
            w = numpy.array(weights)
            x = numpy.array(image[0])
            weights = w + x
        elif (y == 1) and (compare == 0):
            w = numpy.array(weights)
            x = numpy.array(image[0])
            weights = w - x
        else:
            correct += 1

        # if we are on a 25th trial, record the accuracy level
        if (i != 0) and ((i + 1) % 25 == 0):
            accuracy_values.append(correct / i)

    # plot the accuracy level per 25 images
    x = numpy.arange(1, len(accuracy_values) + 1)
    x = [i * 25 for i in x]
    plt.xlabel("trial")
    plt.ylabel("accuracy")
    plt.title("accuracy with trials")
    plt.plot(x, accuracy_values)
    plt.show()
    return (weights, stop)
```

```
def runPerceptron(d1, d2):
    # get images for d1 and d2, and create lists of tuples that group each image
    # with the digit it is representing
    A = load_image_files(d1)
    A_list = [(x, d1) for x in A]
    B = load_image_files(d2)
    B_list = [(x, d2) for x in B]
    big_list = A_list + B_list

    # shuffle the list
    numpy.random.shuffle(big_list)

    # set up some random initial weights
    weights = numpy.random.normal(0,1,size=len(A[0]))

    return Perceptron(d1, d2, big_list, weights)
#3
new_weights = runPerceptron(0, 1)[0]
```

Problem #1

Graph

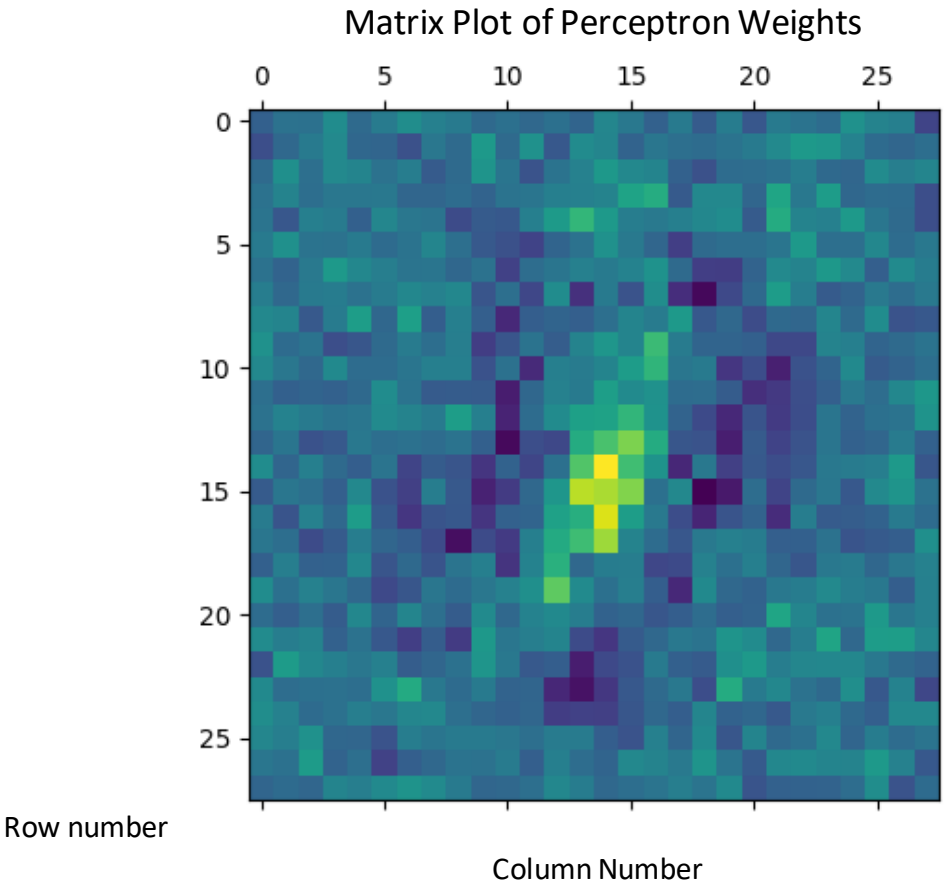


Problem #2

From the graph in Problem 1, we can see very clearly that the accuracy of the perceptron in differentiating between 1's and 0's does seem to converge on 100%. It will never be exactly 100% (since our accuracy is dependent on the total number of trials, and we are bound to have a few incorrect guesses during the training in the first few trials), but it gets very close. In my code, I have a loop that iterates through a shuffled list of the 12,665 images (of both 1's and 0's); however, I have a check at the beginning of each iteration of the loop that calls for the loop to break if the accuracy passes 99%. In the graph, we can see that the numbers along to x axis (the trial/image number) only reach ~2,500, meaning it only took ~2,500 images out of the 12,665 possible in training to reach that 99% accuracy mark. With more trials, we could probably get closer to the fill 100%; however, doing so and taking up extra runtime would be unnecessary at this point, I believe.

The fact that the perceptron was so clearly able to converge on 100% accuracy tells us that 1's and 0's are very much linearly separable on this feature space. This is to be expected; the digit 0 and the digit 1 are perhaps two of the most simple looking digits, as well as the most unique looking from each other. It would be difficult to hand-write either of these digits in such an obscure way that it would look like the other (i.e. it is very easy to judge at a hand-written digit and make a binary decision as to whether it is a 0 or a 1, which is basically the definition of linear separability). This may not pertain to other digits that have more overlapping features, such as 0 and 8 or 6 and 9.

Problem #3



In the matrix plot above, our weights are plotted as 28x28 matrix. Lighter colors (such as yellow, green, red) indicate larger numbers/weights, and darker colors (such as purple, blue) indicate lower numbers/weights. Those that are the neutral blue color are the weights closest to zero (the weights that don't have too much impact on classifying the digit as a 1 or 0). In our matrix, we see that most weight fall into the 'blue' color, with a somewhat low/middle tier weight. However, we can very clearly see a bright yellow center, and purple half circles surrounding this center. Considering the layout of our algorithm, this is exactly what we'd expect.

In my particular algorithm, I tell a weight to increase when the perceptron guesses it is a 0, but it is actually a 1. So, it would make sense that the weights corresponding to the center pixels of the images (around where the one would be written) would need to be increased much more than the surrounding pixels in order to detect a 1. Similarly, I have a similar procedure for when the digit is a 0 but the perceptron guesses a 1, except in that case I decrease the weight. This is what causes the stark purple circle around the center, in approximately the pixels that a 0 digit would take up and need to have detected in order to be correctly identified.

Problem #4

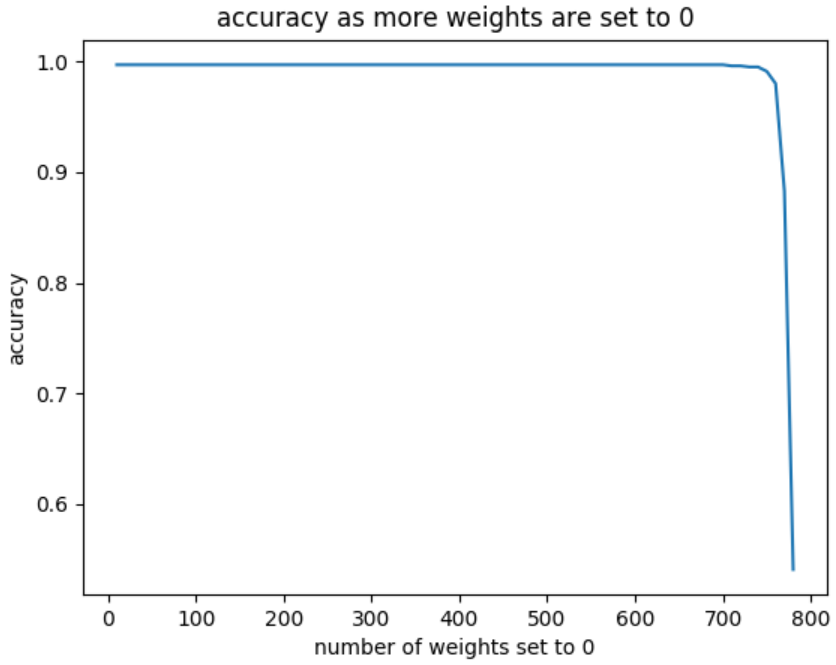
Code

```
def runPerceptronAdjust(d1, d2):

    # get images for d1 and d2, and create lists of tuples that group each image
    # with the digit it is representing
    A = load_image_files(d1)
    A_list = [(x, d1) for x in A]
    B = load_image_files(d2)
    B_list = [(x, d2) for x in B]
    big_list = A_list + B_list

    # shuffle the list
    numpy.random.shuffle(big_list)

    # set up some random initial weights
    weights = numpy.random.normal(0,1,size=len(A[0]))

    run = Perceptron(d1, d2, big_list, weights)
    # derive our adjusted weights and image index where we stopped our perceptron
    new_weights = run[0]
    index = run[1]
    trials = big_list[index : index + 1000]

    # creating some reference lists to help us change weights to 0
    weights_ref = []
    for i in range(0, len(new_weights)):
        weights_ref.append((abs(new_weights[i]), i))
    weights_ref = sorted(weights_ref, key = lambda x : x[0])[:780]

    # the y values for our plot
    accuracy_values = []

    for i in range(0, 78):

        # getting to values to set to 0 and removing them from reference
        weights_change = weights_ref[:10]
        weights_ref = weights_ref[10:]

        # changing the values in our actual weights
        for j in range(0, 10):
            change_index = weights_change[j][1]
            new_weights[change_index] = 0
```

```
        # test these weights over our 1000 new trials
        correct = 0
        for trial in trials:
            compare = trial[1]
            image = trial[0]
            dot = numpy.dot(image, new_weights)
            # y is the result of our step function
            y = 0
            if dot < 0:
                y = 0
            else:
                y = 1
            # see if our perceptron guessed correctly
            if (y == compare):
                correct += 1
        # record the accuracy
        accuracy_values.append(correct / 1000)

    x = numpy.arange(1, len(accuracy_values) + 1)
    x = [i * 10 for i in x]
    plt.xlabel("number of weights set to 0")
    plt.ylabel("accuracy")
    plt.title("accuracy as more weights are set to 0")
    plt.plot(x, accuracy_values)
    plt.show()

runPerceptronAdjust(0, 1)
```

Problem #4

Graph



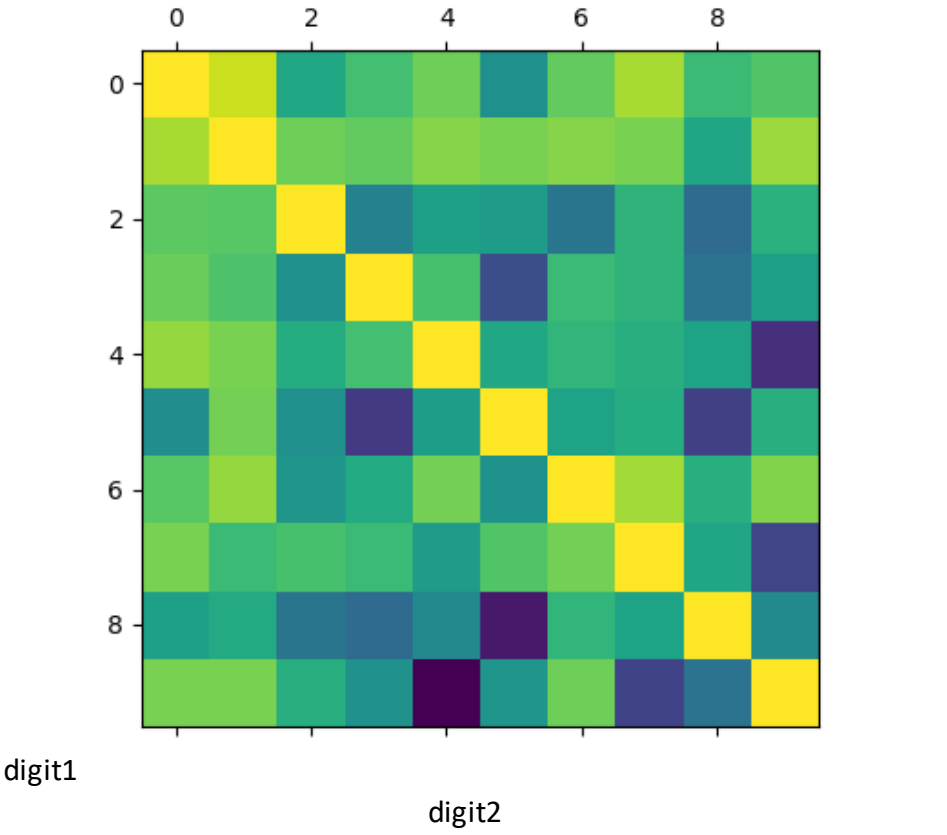
The graph shows a very sharp drop somewhere around 700 to 750. This tells us that for the perceptron's diagnostic in determining if a digit is a 0 or a 1, there are very few pixels (~30-80) that actually contribute to its decision and help it maintain the near 100% accuracy.

Problem #5
Code

```
def PerceptronAccuracy(d1, d2, big_list, weights):

    # like our previous perceptron, but it returns the accuracy after 2500 trials

    # we will keep track of how many the perceptron gets correct
    correct = 0

    for i in range(0, 2500):

        image = big_list[i]
        s = numpy.dot(image[0], weights) # dot the inputs with the weights
        compare = image[1]

        # y is the result of our step function
        y = 0
        if s < 0:
            y = 0
        else:
            y = 1

        # fix the weights depending on if y matches compare (the real digit)
        # add 1 to correct if it got it right
        if (y == 0) and (compare == d2):
            w = numpy.array(weights)
            x = numpy.array(image[0])
            weights = w + x
        elif (y == 1) and (compare == d1):
            w = numpy.array(weights)
            x = numpy.array(image[0])
            weights = w - x
        else:
            correct += 1

    return correct / 2500 # accuracy after 2500 training trials
```

```
def CompareAllDigs():

    # get images for all our digits, and create lists of tuples that group each image
    # with the digit it is representing
    all_images = {}
    for i in range(0, 10):
        images = load_image_files(i)
        all_images[i] = [(x, i) for x in images]

    # initalize our main matrix which will be 10x10
    big_matrix = []

    # find the accuracy between all pairs of digits
    for i in range(0, 10):
        i_compares = [] # the accuacies between our current i and all other digits
        for j in range(0, 10):
            if i == j:
                i_compares.append(1.0)
            else:
                # getting all our images
                big_list = all_images[i] + all_images[j]
                numpy.random.shuffle(big_list)
                weights = numpy.random.normal(0,1,size=len(big_list[0][0]))
                i_compares.append(PerceptronAccuracy(i, j, big_list, weights))
        # adding our list of 10 accuracies to our main matrix
        print(i, i_compares)
        big_matrix.append(i_compares)

    # showing the results
    plt.matshow(big_matrix)
    plt.show()

CompareAllDigs()
```

Problem #5
Graph



The graph above displays the perceptron's accuracy level of distinguishing between two digits 0 through 9. The first thing we notice is the diagonal line of yellow accuracies, indicating a high accuracy level. This is expected; the yellow squares are on the squares representing a comparison between the same number (such as 0 and 0) so the accuracy should be 1. We can also look at specific number comparisons to see if this matrix meets our expectations.

Overall, it does. One of the darkest spots (lowest accuracies) is between 9 and 4. These number share similar features, so that seems reasonable. I suspected 6 and 9 would have a low accuracy, but their accuracy is somewhat high (light green). I suppose this is because the perceptron looks at the location of features, not just the features themselves. Other dark spots (low accuracy) include 8 and 5, 3 and 5, and 7 and 9. All of these pairs share features, so this supports our expectations.