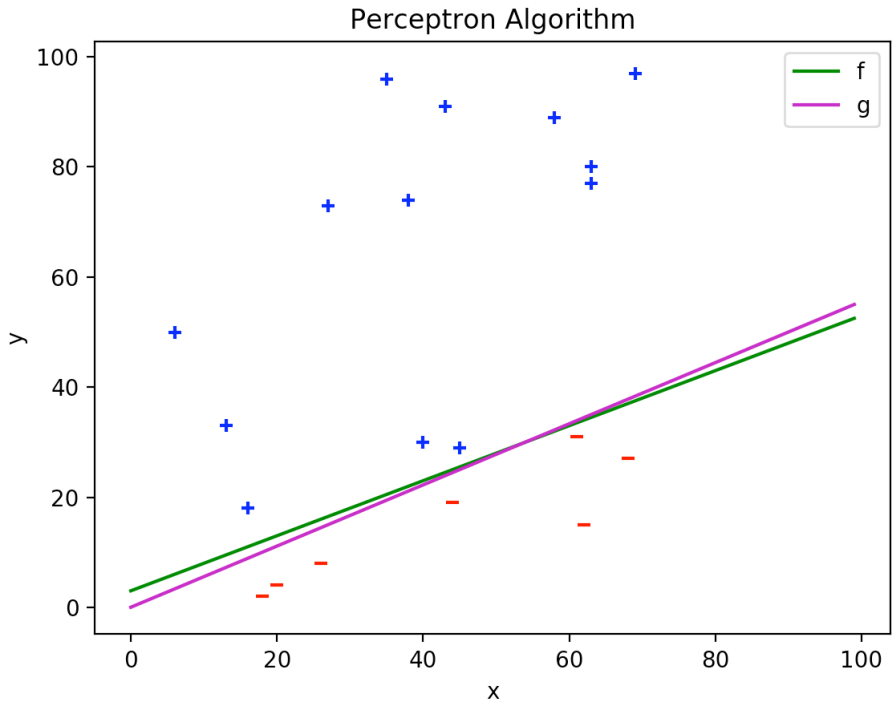
Samantha Rothman

Problem 1.4

a) The target function is f = 0.5x+3. Below is the code to make the target function and generate a random data set with labels (1 or -1) based on the point's relation to the target function. The result is plotted in part b.

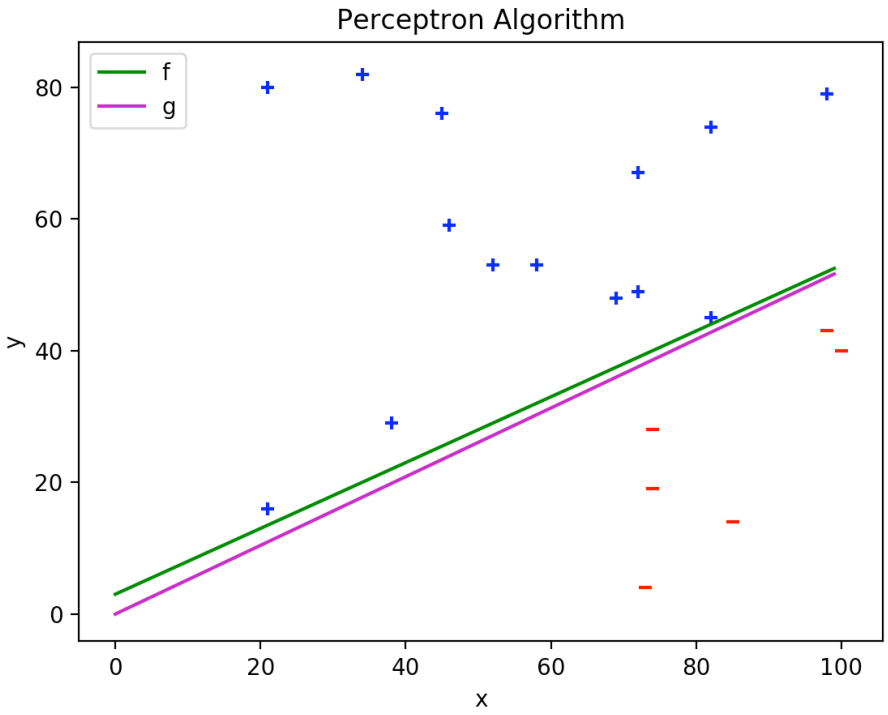
```
target = 0.5 * t + 3
data = numpy.random.randint(low = 0, high = 101, size =(20, 3))

for i in data:
    x = i[0]
    if i[1] >= (0.5*x+3):
        i[2] = 1
    else:
        i[2] = -1
```

b) When running the perceptron algorithm on the randomly generated data, the program gives the following results: final weights: [-0.04 -1.6 2.88] and number of updates: 18. It also creates the graph below.



In this case, f is close to g, and they both separate the data well. The slopes of the two lines look different, though.

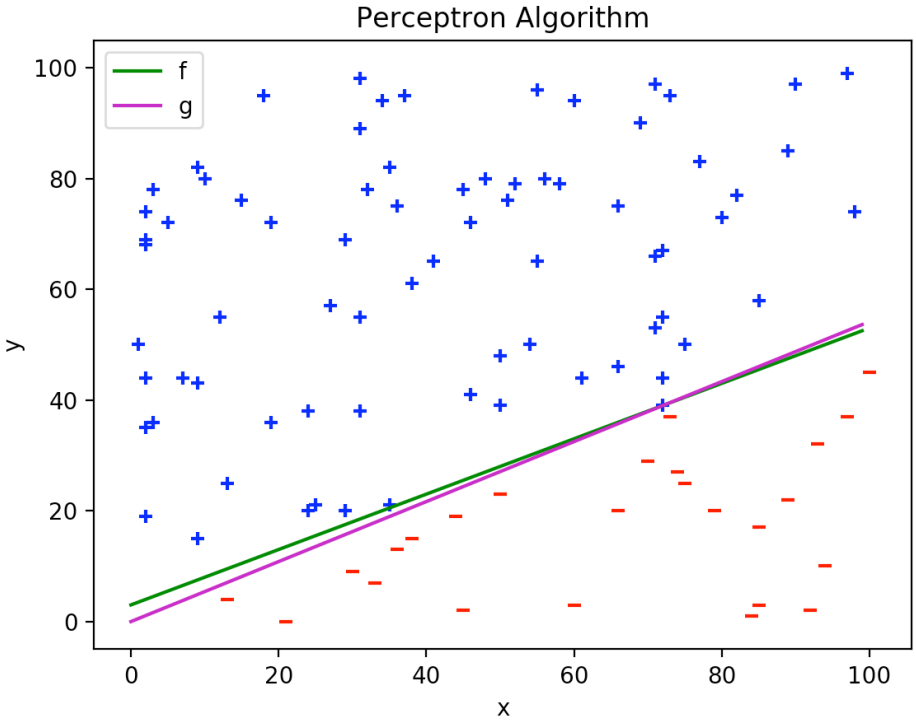
c) Repeating this with a new randomly generated data set of size 20, the program yields the following results for the graph below: final weights: [0.02 -0.96 1.84] and number of updates: 7.



Similarly to part b, the two functions separate the data well and are close to each other. This time, however, the slopes of the lines seem to be closer. Also, the algorithm converged quicker.

d) To generate 100 random data points, I changed the code from part a to:

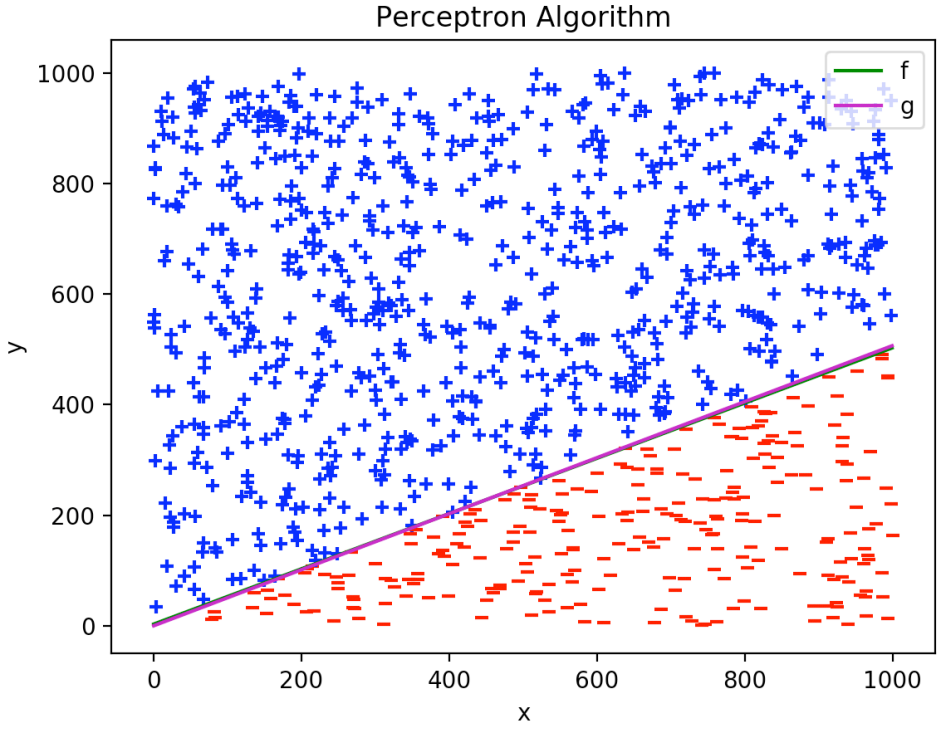
```
data = numpy.random.randint(low = 0, high = 101, size =(100, 3))
```



Running the algorithm on the 100 points, the program gave the final results: final weights: [0.04 -2.08 3.84] and number of updates: 20. In the graph above, the target function and final hypothesis are extremely close in this and even overlap. The slopes are a little different, but f and g look better than part b. However, part b and c both converged faster than this run.

e) To make the graph look nicer with 1000 points, I changed the scale of the graph so x and y both go to 1000 instead of 100. The code to generate 1000 random data points looked like this:

```
data = numpy.random.randint(low = 0, high = 1001, size =(1000, 3))
```



The perceptron algorithm gave the following results: final weights: [4.0000e-02 -7.3260e+01 1.4452e+02] and number of updates: 220. In the graph above, it looks as if f and g are almost exactly the same. Since both functions do a good job at separating the data, this extremely accurate result seems better than the runs above. Nevertheless, it did take many more updates to converge.

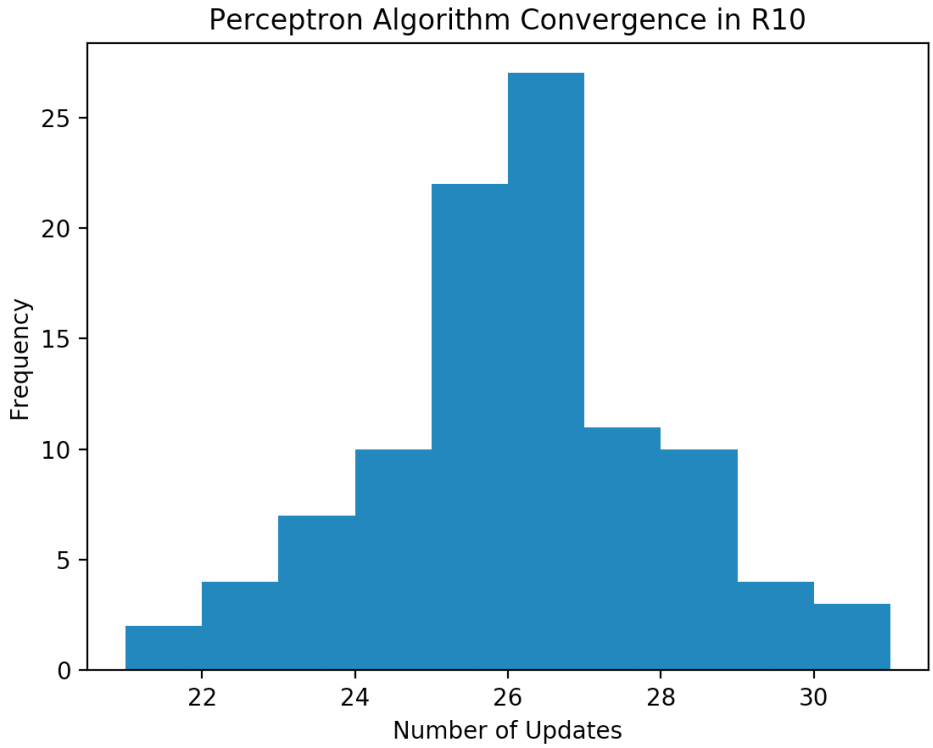
f) Using the code below, I randomly generated two multivariate normal distributions of 500 random points with 10 dimensions and then combined them into a set of 1000 linearly separable data points, with labels 1 and -1 for each distribution.

```
ten = numpy.random.multivariate_normal(10*numpy.ones(11), 2*numpy.eye(11), size=500)
two = numpy.random.multivariate_normal(numpy.zeros(11), numpy.eye(11), size=500)
inputs = numpy.concatenate((ten, two), axis = 0)

count = 0
for i in inputs:
    if count <500:
        i[-1] =  1
    else:
        i[-1] = -1
    count += 1
```

The results of the algorithm were final weights:
[0.56 0.02571458 0.04233727 0.0415056 0.07154859 0.01432669 0.05008635 0.066495 08 -0.01932723 0.05756056 0.00532484] and number of updates: 30.

g) Using the code below, I picked x(t) randomly for 100 experiments on the same data as part f. I then graphed the number of updates that the algorithm recorded until convergence each of the 100 times in the histogram below.

```
used = []
train = []
updates = []
for j in range(100):
    while len(used) != 1000:
        index = numpy.random.randint(low = 0, high = 1000
        if index not in used:
            used.append(index)
            point = inputs[index]
            train.append(point)

    perceptron =  Perceptron(10)
    perceptron.training(train)
    updates.append(perceptron.count)
```



h) As a result of the problems above, it is clear that the perceptron learning algorithm improves accuracy as N increases, but running time increases. When d increases, the accuracy also improves while run time increases.

Samantha Rothman

Code used for perceptron (in this case for 1000 points):

```
import matplotlib.pyplot as plt
import numpy

class Perceptron(object):
    """Class to run the perceptron learning algorithm"""

    def __init__(self, dimension,threshold = 1000, learning_rate = 0.01):
        """initialize variables for the algorithm"""
        self.dimension = dimension
        self.thresh = threshold
        self.learn = learning_rate
        self.ws = numpy.zeros(dimension + 1)
        self.count = 0


    def predictor(self, point):
        """Predict label for each data point
           input = data point (has d dimensions and a label
           returns predicted label (int) """
        w12 = self.ws[1:]
        xy = point[:self.dimension]
        net = self.ws[0] + numpy.dot(w12, xy)
        if net >= 0:
            output = 1
        else:
            output = -1
        return output



    def training(self, inputs):
        """perceptron learning algorithm - looks for a misclassified
           point and and updates the weights until all points are correct""
        misclassified = 1000
        while misclassified != 0:
            if self.count == self.thresh:
                break
            index = numpy.random.randint(low = 0, high = 1000)
            point = inputs[index]
            label = point[self.dimension]
            xy = point[:self.dimension]
            pred = self.predictor(xy)
            if pred == point[self.dimension]:
                misclassified -= 1
            else:
                self.ws[1:] += self.learn * (label - pred) * xy
                self.ws[0] += self.learn * (label - pred)
                self.count += 1
                misclassified = 1000

"""
    def training(self, inputs):
    #Another version of the algorithm that goes through every
    #point and stops when the weights do not change between runs
    #did not use this but it has similar outcomes
        before = []
        for i in range(self.dimension + 1):
            before.append(0)
        for i in range(self.thresh):
            for point in inputs:
                label = point[self.dimension]
                xy = point[:self.dimension]
                pred = self.predictor(xy)
                self.ws[1:] += self.learn * (label - pred) * xy
                self.ws[0] += self.learn * (label - pred)
            if numpy.array_equal(before,self.ws) == False:
                self.count += 1
                for i in range(self.dimension + 1):
                    before[i] = self.ws[i]
                continue
            else:
                 break
"""
```

Samantha Rothman

To graph 2d:

```
# x axis
t = numpy.arange(0,1000)

# create target and data
target = 0.5 * t + 3
data = numpy.random.randint(low = 0, high = 1001, size =(1000, 3))

# label data according to target
for i in data:
    x = i[0]
    if i[1] >= (0.5*x+3):
        i[2] = 1
    else:
        i[2] = -1

# run perceptron for 2 dimensions
perceptron =  Perceptron(2)
perceptron.training(data)
print("final weights:", perceptron.ws)
print("number of updates:", perceptron.count)

# equation of line created by perceptron
y = -(perceptron.ws[0]/ perceptron.ws[2]) - ((perceptron.ws[1]/perceptron.ws[2]) * t)

# graph points and lines
plt.plot(t,target, color = 'g', label = 'f')
plt.plot(t, y, color = 'm', label = 'g')
plt.title("Perceptron Algorithm")
plt.legend()
plt.xlabel('x')
plt.ylabel('y')
for i in data:
    if i[2] == -1:
        plt.scatter(i[0], i[1], marker='_', color = 'r')
    else:
        plt.scatter(i[0], i[1], marker='+', color = 'b')

plt.show()
```

Samantha Rothman

Perceptron with 10 dimensions and histogram:

```
# create random 10d data
ten = numpy.random.multivariate_normal(10*numpy.ones(11), 2*numpy.eye(11), size=500)
two = numpy.random.multivariate_normal(numpy.zeros(11), numpy.eye(11), size=500)
inputs = numpy.concatenate((ten, two), axis = 0)

# label data
count = 0
for i in inputs:
    if count <500:
        i[-1] =  1
    else:
        i[-1] = -1
    count += 1

# run perceptron with 10 dimensions
perceptron =  Perceptron(10)
perceptron.training(inputs)
print("final weights:", perceptron.ws)
print("number of updates:", perceptron.count)

# run perceptron 100 times with x(t) random
used = []
train = []
updates = []
for j in range(100):
    while len(used) != 1000:
        index = numpy.random.randint(low = 0, high = 1000)
        if index not in used:
            used.append(index)
            point = inputs[index]
            train.append(point)

    perceptron =  Perceptron(10)
    perceptron.training(train)
    updates.append(perceptron.count)

# create histogram with number of updates until convergence
plt.hist(updates)
plt.title("Perceptron Algorithm Convergence in R10")
plt.xlabel('Number of Updates')
plt.ylabel('Frequency')
plt.show()
```