Jack Barry

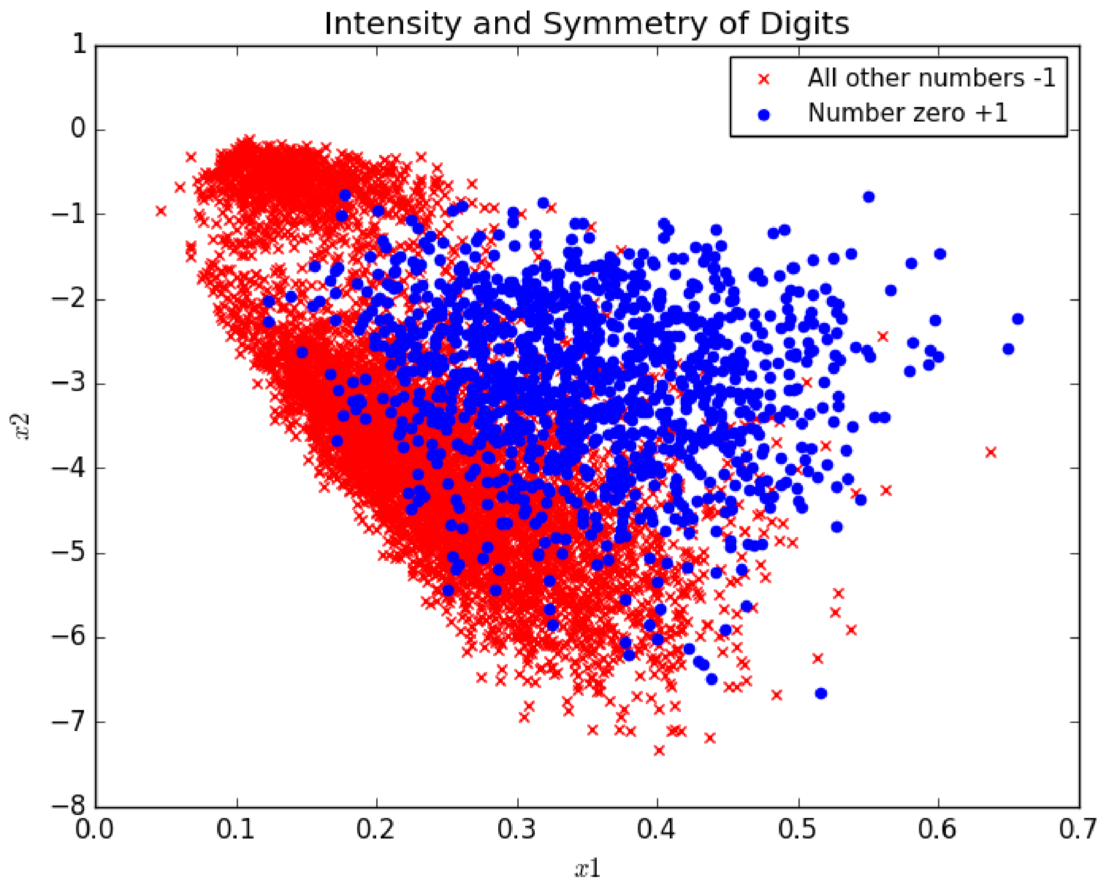
10/13/2016

CMPT 404

Prof. Rivas

Midterm

1- Implementing the code provided was very similar to homework three. This process went relatively smoothly. Getting the data from features.csv to stream through the algorithm was giving errors at first. After looking at the errors that usually mentioned the dimensions of the input not matching the function I realized that N had to be equal to the number of records in features.csv. So after opening features.csv and seeing there was 7291 records I ran the perceptron class with input size N=7291, this worked. Loading the data created the following data plot.



In the future I would change the input size, N to be calculated from the source file with some form of .length function. Seeing the data in this way helped me to understand some of the results I was going to get. The data is not clustered or seperable so therefore the pocket algorithm will not likely not converge and would continue to run if I let it. To stop this from crashing anything I made sure that the function would stop running after w did not change for 1000 iterations. When I run the function with y[y<>0] and y[y==0] and w equal to the solution found by the linear regression I get the following results I get the following results.

Linear regression solution : [-1.27626061 5.5602845   0.23832255]

Best w: [-7.27626061 37.32835076   1.65107255]

Iterations: [1002.]

The pocket error for this case did not change as it found the best solution from using the linear regression solution. The pla error bounced back and forth between 0.1086 and 0.8362.

Running the program with initial w=0 I receive the following results.

The final pocket error was 0.10986147304896447

Best w: [-7.   40.01272639   1.8729375]

Iterations: [1016.]

This run took more iterations to converge and ended with a higher pocket error than the previous example with w = linear regression solution.

I then ran the code with y[y<>1] and y[y==1] under the same conditions and received the following results.

Linear regression solution: [0.54592846 -0.96745805 0.30085587]

Best w: [9.54592846 -11.05825887   5.21535587]

Final pocket error: 0.012069674941708956

Iterations: [1019.]

This run took longer to converge than y=0 but ended at a smaller pocket error. Interesting enough it did change from the initial error even though it was using the linear regression solution as initial w.

When I set the initial w=0 I received the following results.

Best w: [9.   -9.72136344 5.0625   ]

Iterations: [1027]

The final pocket error: 0.012343985735838706

Once again this took longer to converge and had a higher error than the run using w=linear regression solution.

In conclusion it is beneficial to use the solution from linear regression for the pocket algorithm but in reality it just implies that linear regression is a better model than the perceptron and pocket algorithm in this case. I imagine that there is some data where the pla is the best model as I am coming to understand that every analysis depends on the nature of the data being analyzed

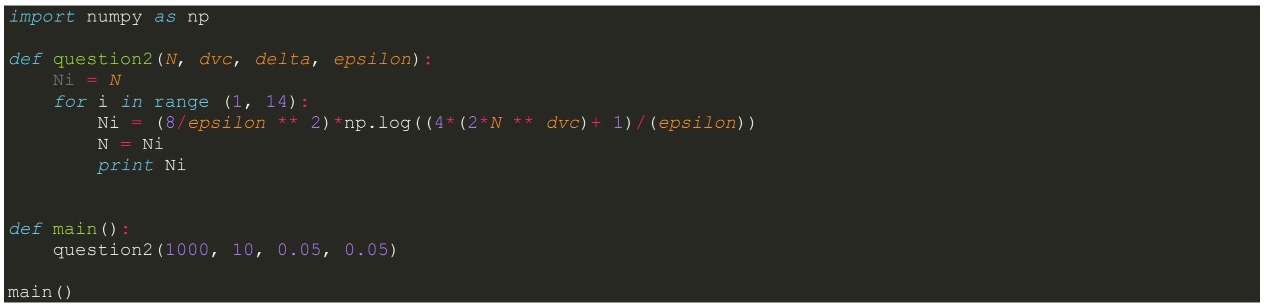
The code I used to solve this problem is as follows.

```
import random
import matplotlib.pyplot as plt
import numpy as np
import copy
from numpy import genfromtxt

class Perceptron:
  def __init__(self, N):
    self.X = self.generate_points(N)

  def generate_points(self, N):
    X, y = self.make_data(N)
    bX = []
    for k in range(0, N):
      bX.append((np.concatenate(([1], X[k, :])), y[k]))
    # this will calculate linear regression at this point
    X = np.concatenate((np.ones((N, 1)), X), axis=1);  # adds 1 as a constant
    self.linRegW = np.linalg.pinv(X.T.dot(X)).dot(X.T).dot(y)
    print self.linRegW
    return bX

  def make_data(self, N):
    dataset = genfromtxt('features.csv', delimiter=' ')
    y = dataset[0:N, 0]
    X = dataset[0:N, 1:]
    y[y<>1] = -1
    y[y==1] = +1
    c0 = plt.scatter(X[y == -1, 0], X[y == -1, 1], s=20, color='r', marker='x')
    c1 = plt.scatter(X[y == 1, 0], X[y == 1, 1], s=20, color='b', marker='o')
    plt.legend((c0,c1), ('All other numbers -1', 'Number zero +1'),
          loc='upper right', scatterpoints=1, fontsize=11)
    plt.xlabel(r'$x1$')
    plt.ylabel(r'$x2$')
    plt.title(r'Intensity and Symmetry of Digits')
    plt.savefig('midterm.plot.png', bbox_inches='tight')
    plt.show
    return X, y
```

```
def plot(self, mispts=None, vec=None, save=False):
    fig = plt.figure(figsize=(5, 5))
    plt.xlim(-10, 10)
    plt.ylim(-10, 10)
    l = np.linspace(-1.5, 2.5)
    V = self.linRegW
    a, b = -V[1] / V[2], -V[0] / V[2]
    plt.plot(l, a * l + b, 'k-')
    cols = {1: 'r', -1: 'b'}
    for x, s in self.X:
        plt.plot(x[1], x[2], cols[s] + '.')
    if mispts:
        for x, s in mispts:
            plt.plot(x[1], x[2], cols[s] + 'x')
    if vec.size:
        aa, bb = -vec[1] / vec[2], -vec[0] / vec[2]
        plt.plot(l, aa * l + bb, 'g-', lw=2)
    if save:
        if not mispts:
            plt.title('N = %s' % (str(len(self.X))))
        else:
            plt.title('N = %s with %s test points' \
                      % (str(len(self.X)), str(len(mispts))))
        plt.savefig('p_N%s' % (str(len(self.X))), \
                    dpi=200, bbox_inches='tight')

def classification_error(self, vec, pts=None):
    if not pts:
        pts = self.X

    M = len(pts)
    n_mispts = 0

    for x, s in pts:
        if int(np.sign(vec.T.dot(x))) != s:
            n_mispts += 1
    error = n_mispts / float(M)
    return error

def choose_miscl_point(self, vec):
    pts = self.X
    mispts = []
    for x, s in pts:
        if int(np.sign(vec.T.dot(x))) != s:
            mispts.append((x, s))
        return mispts[random.randrange(0, len(mispts))]

    def pla(self, save=False, linear=False):
        #set w=0 to start if we are running pocket algorithm and set it equal to the solution found by
linear regression if we are running linear regression
        if linear != False:
            w=self.linRegW
        else:
            w = np.zeros(3)

        self.keepW = copy.deepcopy(w)
        self.plaError = []
        self.pocketError = []
        X, N = self.X, len(self.X)
        #set number of iterations = 0
        it = 0
        lastIT = 0
        self.plaError.append(self.classification_error(w))
        self.pocketError.append(self.plaError[it])
        #keep running until all points are correctly classified or we exceed 10 iterations of pocket
        while self.plaError[it] != 0 and lastIT < 1000:
            it += 1
            # Pick random misclassified point
            x, y = self.choose_miscl_point(w)
            # Update weights
            w += y * x
            # Update if new w is better
            self.plaError.append(self.classification_error(w))
            #increment the count to check if there is a change in w. this will allow us to stop running if w
is not changing
            if self.pocketError[it-1] < self.plaError[it]:
                lastIT+=1

            if (self.pocketError[it - 1] > self.plaError[it]):  # for Pocket
                self.pocketError.append(self.plaError[it])
                self.keepW = copy.deepcopy(w)
            else:
                self.pocketError.append(self.pocketError[it - 1])


        #    if (save==True) and it % 100 == 0 or it ==1:
         #      self.plot(vec=w)
          #     plt.title('N = %s, Iteration %s\n' \
            #            % (str(N), str(it)))
             #    plt.savefig('p_N%s_it%s' % (str(N), str(it)), \
              #            dpi=200, bbox_inches='tight')

        self.w = w
        print 'PLA Error:'
        print self.plaError
        print 'Pocket Error:'
        print self.pocketError
        print 'Best w:'
        print self.keepW
        print lastIT

        return it

    def check_error(self, M, vec):
        check_pts = self.generate_points(M)
        return self.classification_error(vec, pts=check_pts)

def main():
    it = np.zeros(1)
    for x in range(0, 1):
        p = Perceptron(7291)
        it[x] = p.pla(save=True)
        print it
main()
```

Question 2

For this question I referenced the book for equation 2.13. I then interpreted the logic of this equation into python code. My code is as follows.



```
import numpy as np

def question2(N, dvc, delta, epsilon):
    Ni = N
    for i in range (1, 14):
        Ni = (8/epsilon ** 2)*np.log((4*(2*N ** dvc)+ 1)/(epsilon))
        N = Ni
        print Ni


def main():
    question2(1000, 10, 0.05, 0.05)

main()
```

The output that is yielded is the following

```
237288.725136

412305.60991

429985.200208

431328.750415

431428.583099

431435.988764

431436.538053

431436.578795

431436.581817

431436.582041

431436.582057

431436.582059

431436.582059
```

I chose to stop calling the function in a recursive manner after 14 iterations because after playing with the program the output stopped changing after the 13th iteration, I included the 14th to illustrate this I extended to 1000 and the result at the 1000th iteration was the same as the 13th.