William Bidle

Final Question 3

Given the simplistic nature of the problem, I thought this would be a good opportunity to compare the performances of a linear and logistic model on a classification problem. It is uncommon to see linear models used in classification problems, mainly because the output of a linear model is non-probabilistic, and therefore makes it difficult to say which category the data fits into. A logistic model on the other hand, takes the same structure of the linear model, and utilizes an activation function that turns our nonsense output into one that is a probability of being in a certain class. For this problem I took data from Class A to be represented by a 0, and data from Class B to be represented as a 1. Given how little data we were provided, I wasn't expecting my networks to learn a whole lot, or even work well at all. Indeed, when I first tried this problem, the small dataset led to some nonsense results. To overcome this, I tried my hand at some data augmentation, and the results were much more satisfying.

Data Augmentation

The original data that was given only contained letters drawn on a 5x5 pixel grid, leaving 25 total pixels that I could give my networks (see Figure 1a). To improve this data set, I expanded the size of each image to 11x11[1], 121 total pixels, and added some random noise around the outside of the letter (see Figure 1b). I did this 100 times for each letter, so now my 10 5x5 images turned into 1000 11x11 images. Obviously, there would still be a risk to overfitting if the generated noise wasn't sufficient enough, which is the main reason I didn't use even more copies. If I generated too many copies, given the limited number of combinations for random noise, some samples may start to repeat. I also made sure to shuffle the data around each time the model was trained to make sure there wasn't any accidental overfitting I could easily prevent.

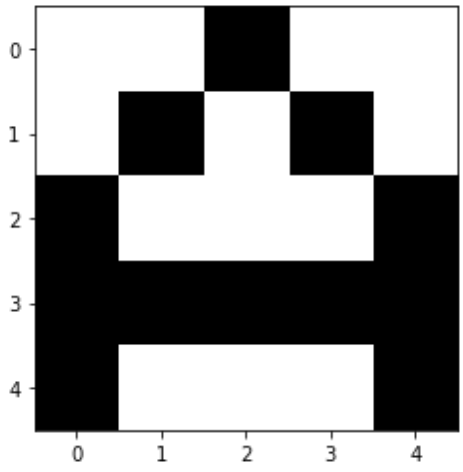


Figure 1a. Original Data

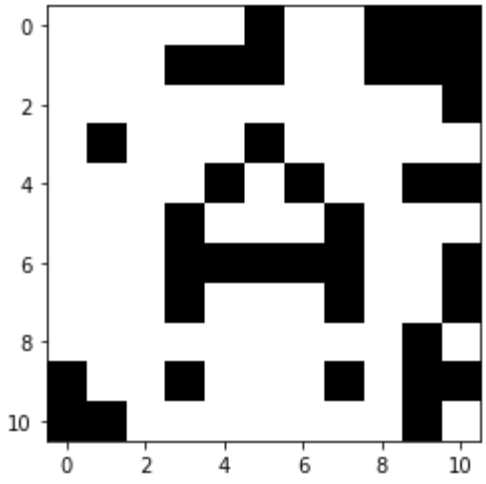


Figure 1b. Augmented Data

[1] No real reason I chose this size other than it allowed me to use the outer two layers for noise and keep the area around the letter intact.

Linear Regression

The first model I constructed was a linear regression model. Given our input size of 121 (121 pixels in an 11x11 image), I needed to generate 121 random weights (each close to zero) associated to each input. The output was calculated by simply taking a linear combination all the weights and corresponding inputs multiplied together:

$$Output = \underline{w} \cdot \underline{input}$$

Where $\underline{w} \cdot \underline{input}$ is a dot product of the weight and input vectors. The cost is taken to be the difference squared between the output and actual class value (either 0 or 1):

$$Cost = (Output - Actual)^2$$

From these two equations, it is fairly easy to calculate the gradient:

$$\nabla C = \frac{\partial C}{\partial \underline{w}} = 2 * (Output - Actual) * \underline{input}$$

And I can apply stochastic gradient descent to our weights:

$$\underline{w}_{\text{next}} = \underline{w}_{\text{current}} - \alpha * \nabla C$$

Where α is the learning rate of the model, a parameter I had to mess around with to get results close-ish to 0 and 1 (again, not expecting this to work perfectly, but I tried to at least have it make some sense). Using this method, I updated the weights after every image to get my final results. As expected, the results themselves are a little difficult to analyze. Figures 2a and 2b show two results for two different training sessions, with the reminder that we initialized Class A with 0, and Class B with 1. Not many of the results are very close to 0, and in both scenarios, L was found to have an output greater than 1. At first this makes me think the model is indeed just useless, however I decided to run a bunch of training and testing sessions, then average the outputs to see what the expected value of each test data point is. The results can be seen in Figure 3.

```
0 = Class A, 1 = Class B

K : [0.4142958]
L : [1.16529256]
M : [0.53973273]
N : [0.90748985]
O : [0.9430821]
```

Figure 2a. Testing Results 1

```
0 = Class A, 1 = Class B

K : [0.97657424]
L : [1.32555704]
M : [0.78417439]
N : [0.5578786]
O : [0.33492971]
```

Figure 2a. Testing Results 2

The results below give me an idea of what the network gives as an output on average for a given training session. Given this information, one of the conclusions I think to draw is that the network is saying K is mostly different than all of the other letters. Whatever class it belongs to (potentially Class A since it is closer to 0?), it is definitely not grouped with L. To me this makes some deal of sense, as K has 2 long diagonals, while L has only two straight lines. It may be recognizing these diagonal components in the other letters M, N, and O as well (even if it is only a little), and these signatures draw the result closer to 0. It is worth noticing that only Class A has diagonal lines in its letters, as Class B only has straight line components. It is hard to say what exactly the network is classifying each letter as, but if I had to make a guess, I would say it is calling K and M Class As, and L, N, and O Class Bs. This may or may not be what the network is picking up on, and it would be worth trying to improve our analysis by applying logistic regression.

```
K: 0.4097842057755175
L: 1.198522431222037
M: 0.6004060049963317
N: 0.9855945800926099
O: 0.9986540619957612
```

Figure 3. Average Testing Results

Logistic Regression

The second model I constructed was a logistic regression model in the hopes that it would do better than the linear model. Given the same input size of 121 (121 pixels in an 11x11 image), I again needed to generate 121 random weights (each close to zero) associated to each input. This time, the linear combination of weights and corresponding inputs was put into a sigmoid activation function:

$$Output = \sigma\left(\underline{w} \cdot \underline{input}\right)$$

Where $\underline{w} \cdot \underline{input}$ is again the dot product of the weight and input vectors, and σ is the sigmoid function. This now squishes the output between 0 and 1, reminiscent of probabilities, something much easier to interpret than the output of the linear model. This time the cost is taken to be:

$$Cost = -y * \ln[output] - (1 - y) * \ln[1 - output]$$

Where y is the true value of the input data, and the nice thing about this equation is that much like the linear model, there is a simple formula for the gradient:

$$\nabla C = \frac{\partial C}{\partial \underline{w}} = (Output - Actual) * \underline{input}$$

And I can apply stochastic gradient descent to our weights:

$$\underline{w}_{\text{next}} = \underline{w}_{\text{current}} - \alpha * \nabla C$$

Where α is again the learning rate of the model (I used the same one I found for the linear model to keep the comparison fair). Trained on the same data I used for the linear model, Figures 4a and 4b show two results for two different training sessions. Immediately I can see some differences and even some similarities between these results and the linear model's results, but first I again ran an average over many training sessions (see Figure 5).

```
0 = Class A, 1 = Class B

K: 0.880257498617049
L: 0.07891872383152632
M: 0.9940476334476436
N: 0.6341639055481836
O: 0.5422886622835373
```

Figure 4a. Testing Results 1

```
0 = Class A, 1 = Class B

K: 0.9534740041307067
L: 0.07770175879716823
M: 0.9722844341940813
N: 0.3446048268613188
O: 0.291273903045164
```

Figure 4a. Testing Results 1

The biggest thing to notice is how close to 0 L is in this model. In the linear model, the output for L was higher than any of the other letters in the test set, which almost seemed to indicate that it was being marked as Class A. The results below seem to indicate the opposite conclusion, as its value is extremely close to 0. If we draw a cutoff at 0.5, then the model is telling us that K and M are Class B, and L, N, and O are Class A. This is exactly the opposite conclusion I drew for the linear model's results, however this time I feel more confident with my analysis. The results this time seem to make sense for pretty much every letter. For example, I could see structural similarities in the shapes of K[2], H, and F, and O with D and C[3]. The main reason being what the logistic model's outputs actually represent. The output of the linear model is not constraint to lie in a certain range of values, which makes it really difficult to say with certainty what the results actually mean. In the case of the logistic model, the output is being fed into a function that forces the value to be between 0 and 1, synonymous with probability. Given our only two classes are Class A = 0 and Class B = 1, if the output is close to 0, it is predicting Class A, and if the output is close to 1, it is predicting Class B. The closer the output is to these values the more certain the network is. In the case of N and O, the network is classifying them both as Class A, however it is far more uncertain of these than it is of K, L or M.

```
K: 0.9185886759575014
L: 0.05616921664531114
M: 0.9773810916134421
N: 0.4119140368982378
O: 0.3504726338766631
```

Figure 5. Average Testing Results

[2] Perhaps the linear model didn't actually recognize diagonal features that K shared with Class A in the first place.
[3] This is one of the main reasons I am more comfortable with the logistic model's analysis. If anything was to belong to Class A it would be O.

I am sure there are some flaws with both of the networks, however given the scale of the problem and the lack of knowledge where the test data actually belongs, there were bound to be errors. I would conclude, however, that the use of a logistic regression model is much more useful for a classification problem than a linear regression model. There may be ways to interpret the linear model's results in a meaningful way but given how simple a logistic model makes the problem, it is the superior option.

In [1020]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
import time
```

Initialize Data

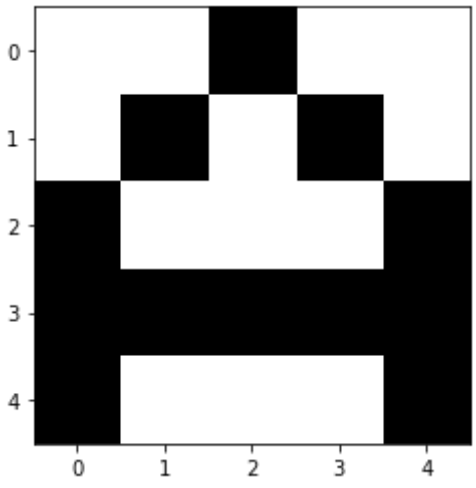
In [1021]:

```
### Class A ###

A = np.array([[0,0,1,0,0],
     [0,1,0,1,0],
     [1,0,0,0,1],
     [1,1,1,1,1],
     [1,0,0,0,1]])

B = np.array([[1,1,1,0,0],
     [1,0,0,1,0],
     [1,1,1,0,0],
     [1,0,0,1,0],
     [1,1,1,0,0]])

C = np.array([[0,1,1,0,0],
     [1,0,0,1,0],
     [1,0,0,0,0],
     [1,0,0,1,0],
     [0,1,1,0,0]])

D = np.array([[1,1,1,0,0],
     [1,0,0,1,0],
     [1,0,0,1,0],
     [1,0,0,1,0],
     [1,1,1,0,0]])

E = np.array([[1,1,1,1,0],
     [1,0,0,0,0],
     [1,1,1,1,0],
     [1,0,0,0,0],
     [1,1,1,1,0]])

#### Class B ###

F = np.array([[1,1,1,1,0],
     [1,0,0,0,0],
     [1,1,1,0,0],
     [1,0,0,0,0],
     [1,0,0,0,0]])

G = np.array([[1,1,1,1,0],
     [1,0,0,1,0],
     [1,0,0,0,0],
     [1,0,1,1,1],
     [1,1,1,1,0]])

H = np.array([[1,0,0,0,1],
     [1,0,0,0,1],
     [1,1,1,1,1],
     [1,0,0,0,1],
     [1,0,0,0,1]])

I = np.array([[1,1,1,1,1],
     [0,0,1,0,0],
     [0,0,1,0,0],
     [0,0,1,0,0],
```

```
     [1,1,1,1,1]])

J = np.array([[1,1,1,1,0],
     [0,1,0,0,0],
     [0,1,0,0,0],
     [0,1,0,1,0],
     [0,1,1,1,0]])

### Test Data ###

K = np.array([[1,0,0,1,0],
     [1,0,1,0,0],
     [1,1,0,0,0],
     [1,0,1,0,0],
     [1,0,0,1,0]])
L = np.array([[1,0,0,0,0],
     [1,0,0,0,0],
     [1,0,0,0,0],
     [1,0,0,0,0],
     [1,1,1,1,0]])
M = np.array([[1,1,0,1,1],
     [1,0,1,0,1],
     [1,0,1,0,1],
     [1,0,0,0,1],
     [1,0,0,0,1]])
N = np.array([[1,1,0,0,1],
     [1,1,1,0,1],
     [1,0,1,1,1],
     [1,0,0,1,1],
     [1,0,0,0,1]])
O = np.array([[0,1,1,1,0],
     [1,0,0,0,1],
     [1,0,0,0,1],
     [1,0,0,0,1],
     [0,1,1,1,0]])

Data = []
# stored as (image, label)
Data.append((A,0))
Data.append((B,0))
Data.append((C,0))
Data.append((D,0))
Data.append((E,0))

Data.append((F,1))
Data.append((G,1))
Data.append((H,1))
Data.append((I,1))
Data.append((J,1))

Data = np.array(Data)

test_data = []
test_data.append((K, 'K'))
test_data.append((L, 'L'))
test_data.append((M, 'M'))
test_data.append((N, 'N'))
```

```
test_data.append((O, 'O'))

test_data = np.array(test_data)
```

In [1022]:
```
plt.imshow(Data[0][0], cmap=plt.cm.binary)
plt.show()
```



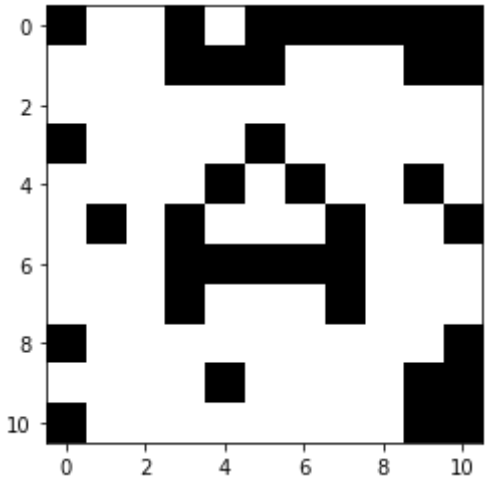
Data Augmentation:

In [1023]:

```
# Expand the size of the images and add some 'noise'

NewData = []
for item in Data:
    letter = item[0]
    tag = item[1]

    duplicates = 0

    #create a number of duplicates
    while duplicates < 100:
        x = np.zeros((11,11))

        #put the letter in the middle of the bigger array
        x[3:8,3:8] = letter

        step = 0

        #create random noise
        while step < 15:
            i = np.random.choice([0,1,9,10])
            j = np.random.choice([0,1,2,3,4,5,6,7,8,9,10])

            if x[i][j] == 1:
                continue
            else:
                x[i][j] = 1
                step += 1

        step = 0
        while step < 10:
            i = np.random.choice([0,1,2,3,4,5,6,7,8,9,10])
            j = np.random.choice([0,1,9,10])

            if x[i][j] == 1:
                continue
            else:
                x[i][j] = 1
                step += 1

        NewData.append((x, tag))
        duplicates += 1

#put the test data in the same bigger image format
NewTestData = []
for item in test_data:
    x = np.zeros((11,11))
    x[3:8,3:8] = item[0]
    NewTestData.append((x, item[1]))
```

In [1024]:
```
print('Now we have', len(NewData), 'images for our networks to use')
plt.imshow(NewData[0][0], cmap=plt.cm.binary)
plt.show()
```

```
Now we have 1000 images for our networks to use
```



Linear Regression

In [1025]:
```
def initialize_weights():
    weights = np.random.rand(1,121)
    return weights

def linear_cost(output_layer, label):
    cost = np.square(output_layer - label)
    return cost

def linear_derivs(input_layer, output_layer, label, weights):
    weight_derivs = (output_layer - label)*input_layer
    return weight_derivs

def linear_output(input_layer, weights):
    output_layer = np.dot(weights,input_layer)
    return output_layer

def train_model(Data):
    weights = initialize_weights()

    np.random.shuffle(Data)

    for item in Data:
        #Preprocess
        input_data = item[0]
        input_layer = input_data.flatten()
        label = item[1]

        #Calculate our layers
        output_layer = linear_output(input_layer, weights)

        #Calculate Cost
        cost = linear_cost(output_layer, label)

        #Calculate weight derivatives
        weight_derivs = linear_derivs(input_layer, output_layer, label,
weights)

        weights = weights - 0.05*weight_derivs

    return weights

def TestModel(test_data, weights):
    results = []
    for item in test_data:
        x = item[0].flatten()
        output_layer = abs(linear_output(x,weights))

        results.append(output_layer[0])
    return results
```

In [1026]:
```
weights = train_model(NewData)
```

In [1027]:
```
results = TestModel(NewTestData, weights)
print("0 = Class A, 1 = Class B")
print()
print("K:",results[0])
print("L:",results[1])
print("M:",results[2])
print("N:",results[3])
print("O:",results[4])
```

```
0 = Class A, 1 = Class B

K: 0.08809749094316827
L: 1.5949090742815057
M: 0.6739724781713333
N: 0.27380102299744635
O: 0.24739313387010187
```

In [1028]:
```
#get an average over a bunch of training sessions
results = np.zeros(5)
for i in range(50):
    weights = train_model(NewData)
    results += TestModel(NewTestData, weights)

results = (results/50)
print("K:",results[0])
print("L:",results[1])
print("M:",results[2])
print("N:",results[3])
print("O:",results[4])
```

```
K: 0.6000170648557712
L: 1.2182755054265357
M: 0.6081753399461132
N: 0.9486863011010143
O: 1.2549434538653776
```

Logistic Regression

In [1029]:
```
# Using initialize_weights from above

def sigmoid(x):
    result = 1/(1 + np.exp(-x))
    return result

def logistic_cost(output_layer, label):
    # using a different cost function here
    cost = -label*np.log(sigmoid(output_layer)) - (1-label)*np.log(1-sig
moid(output_layer))
    return cost

def logistic_derivs(input_layer, output_layer, label, weights):
    #the 2 gets absorbed into the learning rate
    weight_derivs = (output_layer - label)*input_layer #calculate da/dw

    return weight_derivs

def logistic_layers(input_layer, weights):
    output_layer = sigmoid(np.dot(weights,input_layer))

    return output_layer

def train_model(Data):
    weights = initialize_weights()

    np.random.shuffle(Data)

    for item in Data:
        #Preprocess
        input_data = item[0]
        input_layer = input_data.flatten()
        label = item[1]

        #Calculate our layers
        output_layer = logistic_layers(input_layer, weights)

        #Calculate Cost
        cost = logistic_cost(output_layer, label)

        #Calculate weight derivatives
        weight_derivs = logistic_derivs(input_layer, output_layer, label
, weights)

        #update weights
        weights = weights - 0.05*weight_derivs


    return weights

def TestModel(test_data, weights):
    results = []
    for item in test_data:
        x = item[0].flatten()
        output_layer = sigmoid(np.dot(weights,x))
```

```
            results.append(output_layer[0])
        return results
```

In [1030]:
```
weights2 = train_model(NewData)
```

In [1031]:
```
results = TestModel(NewTestData, weights2)
print("0 = Class A, 1 = Class B")
print()
print("K:",results[0])
print("L:",results[1])
print("M:",results[2])
print("N:",results[3])
print("O:",results[4])
```

```
0 = Class A, 1 = Class B

K: 0.9478577009862261
L: 0.06900699544926335
M: 0.9895743242348853
N: 0.6765903812712191
O: 0.3717894803790008
```

In [1032]:
```
#get an average over a bunch of training sessions
results = np.zeros(5)
for i in range(50):
    weights = train_model(NewData)
    results += TestModel(NewTestData, weights)

results = (results/50)
print("K:",results[0])
print("L:",results[1])
print("M:",results[2])
print("N:",results[3])
print("O:",results[4])
```

```
K: 0.9216579885567561
L: 0.05394185985276019
M: 0.9804092755926073
N: 0.4846045488064354
O: 0.34781232487495095
```

In [ ]:
```

```