Alex Shung

Netid: ashung2

Credits: 3

Assignment 4

# Part 1:

## Implementation:

### Training:

The data read from the provided files, and the lines were grouped by 28 lines and then flattened. Essentially taking the 28 x 28 feature and extending it into a 768 x 1 array to represent each number. The labels were read at the same time, matching the place to correctly identify the label of each image. A Number class was defined as the 1d vector of features, or data of the number, and the matching expected label. Perceptrons, represented by a weight vector and a label that they were attempting to identify, were then initialized to 0’s. There was on perceptron initialized for every unique label seen in the training labels data, meaning there was a total of 10. The following pseudocode was then used:

While curEpoch is less than epochLimit:

For every number in trainingData:

For every perceptron in classPerceptrons:

Weights = perceptron.weights

expectedLabel = perceptron.label

data = number.data

y = 1 if number.label == expectedLabel else -1

classifiedLabel = np.sign(weights.dot(data) + bias)

# If classified incorrectly update weights

if y != classifiedLabel:

weights = weights + learningRate(curEpoch) \* (y - classifiedLabel) \* data

perceptron.weights = weights

curEpoch ++

To generate the training curve, after each epoch was finished, the perceptrons were tested against the training data’s hold out set which was a size of 1000. Once the epoch limit was reached, the perceptrons were returned and ready for testing. This testing was done the same as described below, with the difference being that the data being tested was from the hold out set instead of the testing set.

### Testing:

The testing data and labels were read in and processed to numbers in the same way as above. Then for every number in the test data, it’s 1d feature vector was multiplied against every perceptron, and the perceptron which produced the max value was used as that number’s classification. These results were then compared against the real expected results, and the result is the confusion matrix presented below.

### Parameter tuning:

The epoch was decided after looking at the generation of the training curve. The training curve was generated by taking the perceptrons after that epoch, and seeing how many of the hold out set that they correctly identified at that time. From that, it appeared that beyond about 20 epochs the result began to oscillate, so 20 was picked. It may be possible that 20 already over fit much of the data as the resulting classification on the testing data did not hold up to that amount. Other parameters were tuned using the function findBestConfiguration. This function first dealt with the static bias and learning rate, and would train perceptrons on the training data. For bias, a bias of 0 and 1 were tested. For learning rates, a regular (1000 / 1000 + t), a penalized (1000 / 3 \* (1000 + t)), and a lenient (1000 /(1/2) \* (1000 + t)) were tested. Every combination of learning rate and bias was tested. Once the perceptrons were trained, the function then tested the perceptrons against the hold out data. The hold out set was a set of 1000 as before. The combination which produced the best accuracy on the hold out set was then selected to continue testing, which in this case was with bias = 1, and a learning rate function of penalized. With these parameters set, the random parameters (weight vector random initialization and order of data) were tested. For the weight vector, it was initialized to a random weight, trained on the training data, and tested against the hold out set 25 times. The average accuracy across those 25 times was then compared to the value achieved from the non-random value generated from the best static configuration. Since the average was lower than the static method, the weight vector was not randomized. To identify if we should randomize the number order, a similar process was used. The numbers would be randomized every epoch. Once again the perceptrons were trained against the training data, and tested against the hold out set. The average accuracy on the hold out set over 25 times was identified to be lower than the static method. As such, randomized numbers were not used. It is somewhat important to note that the initial ordering already appeared random. Running some simple tests, I could produce particularly awful results with specific orderings, however if the ordering was randomized at least once, the results tended to be fine. The results of the configuration testing are below.

perceptrons.20.0.False.basicLearningRate.isWeightRand.False.txt

Training information

Correctly classified overall: 93.84%

perceptrons.20.0.False.penalizedLearningRate.isWeightRand.False.txt

Training information

Correctly classified overall: 93.86%

perceptrons.20.0.False.lenientLearningRate.isWeightRand.False.txt

Training information

Correctly classified overall: 93.84%

perceptrons.20.1.False.basicLearningRate.isWeightRand.False.txt

Training information

Correctly classified overall: 93.56%

perceptrons.20.1.False.penalizedLearningRate.isWeightRand.False.txt

Training information

Correctly classified overall: 94.66%

perceptrons.20.1.False.lenientLearningRate.isWeightRand.False.txt

Training information

Correctly classified overall: 94.54%

Randomized Weight: Average is 94.014 compared to static 94.66

Randomized Data: Average is 92.88000000000002 compared to static 94.66

bias: 1, randomized data: False, learningRate: penalizedLearningRate, weightRand: False

## Results

### Training Curve:

|  |  |
| --- | --- |
| Epoch | % Correct |
| 0 | 86.08 |
| 1 | 88.84 |
| 2 | 90.76 |
| 3 | 87.38 |
| 4 | 89.28 |
| 5 | 91.8 |
| 6 | 92.3 |
| 7 | 92.9 |
| 8 | 92.16 |
| 9 | 92.92 |
| 10 | 92.38 |
| 11 | 93.2 |
| 12 | 92.8 |
| 13 | 92.88 |
| 14 | 93.88 |
| 15 | 91.38 |
| 16 | 93.48 |
| 17 | 95.02 |
| 18 | 94.62 |
| 19 | 94.66 |
| 20 | 94.22 |
| 21 | 92.76 |
| 22 | 94.06 |
| 23 | 94.84 |
| 24 | 94.6 |
| 25 | 94.2 |
| 26 | 91.7 |
| 27 | 94.26 |
| 28 | 95.28 |
| 29 | 95.16 |
| 30 | 93.16 |
| 31 | 95.2 |
| 32 | 94.08 |
| 33 | 94.74 |
| 34 | 94.84 |
| 35 | 94.74 |
| 36 | 93.2 |
| 37 | 93.56 |
| 38 | 94 |
| 39 | 94.52 |
| 40 | 94.14 |
| 41 | 94.26 |
| 42 | 95.24 |
| 43 | 95.38 |
| 44 | 95.24 |
| 45 | 96.12 |
| 46 | 95.68 |
| 47 | 95.42 |
| 48 | 96.12 |
| 49 | 94.9 |
| 50 | 93.88 |
| 51 | 95.86 |
| 52 | 96.42 |
| 53 | 95.96 |
| 54 | 94.06 |
| 55 | 95 |
| 56 | 95.36 |
| 57 | 95.18 |
| 58 | 96.08 |
| 59 | 93.84 |
| 60 | 94.7 |
| 61 | 95.08 |
| 62 | 95.54 |
| 63 | 95.36 |
| 64 | 96.02 |
| 65 | 96.2 |
| 66 | 95.32 |
| 67 | 94.8 |
| 68 | 95.12 |
| 69 | 95.58 |
| 70 | 96.26 |
| 71 | 95.36 |
| 72 | 96.6 |
| 73 | 95.56 |
| 74 | 96.56 |
| 75 | 94.56 |
| 76 | 95.5 |
| 77 | 96.62 |
| 78 | 95.52 |
| 79 | 96.5 |

### Confusion Matrx

0 1 2 3 4 5 6 7 8 9

0 [ 85 0 1 0 0 0 1 1 2 0]

1 [ 0 102 0 0 1 0 1 0 4 0]

2 [ 0 3 78 1 4 1 4 3 9 0]

3 [ 0 3 0 68 0 20 0 5 4 0]

4 [ 0 0 1 1 87 0 6 2 6 4]

5 [ 3 0 1 2 0 76 1 3 6 0]

6 [ 1 1 2 0 2 3 79 1 2 0]

7 [ 1 4 3 1 3 0 0 81 6 7]

8 [ 2 2 2 4 4 12 2 1 73 1]

9 [ 0 1 0 2 11 4 0 8 3 71]

Correctly classified overall: 80.0%

Rows represent expected values

Columns represent predicted values

## Comparison to Naïve Bayes

My Naïve Bayes implementation of this same data set produced about 77% accuracy, and this method produced about 80%. The Naïve Bayes method tended to misidentify a large chunk of 8’s as 3’s. For the perceptron method, both of the classes seemed to average around the same, 68% for 3’s and 70% for 8. Classifying 0’s were done 10% better (from 84% -> 94%). Class 1 and 2 did a few percetages worse, however class 4, 5, 6 and 7 did significantly better. Class 9’s were done 10% worse as well. As a result, for most classes the perceptron method was better, and as a result the overall classification accuracy increased.