Face and Digit Classification

CS440 - Intro to Artificial Intelligence

Alexander Rozenblit

May 6, 2019

Features

For every algorithm for both digits and face, I used each pixel as a feature. So, for every digit the feature array was a 2 dimensional 28 by 28 array. For each face the feature array was a 2 dimensional 60 by 70 array. I computed the feature arrays by iterating through the data and putting a 1 in the array if the pixel is a '+' or '#' and a 0 if the pixel is white space.

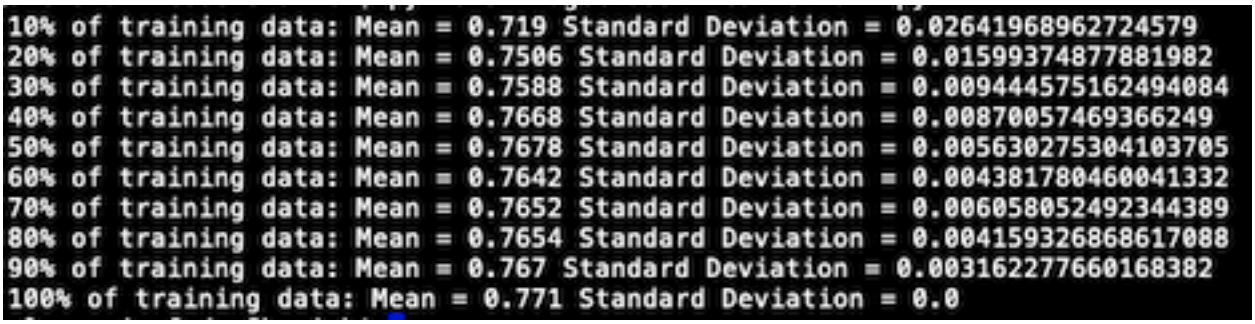
Naive Bayes Classifier

Digits

For the Naive Bayes Classifier for digits I first went through all the training data and added up the features. So I made a 3 dimensional array that was made up of 10, 28 by 28 arrays. Let's say I was traversing a digit that was labeled an "8", I would go to the 8th index of the 3D array, and increment each element if the image had a pixel there. After going through all the training data, and incrementing each element like that, I divided each element of each digit by the frequency of that digit. So, if there were 500 "8"'s I would divide each element in the 8th array by 500. After this process I ended up with a 3D array that showed the probability that there was a pixel in that spot given it was that specific digit . For example if array[0][3][5] had 0.15 in it at that spot. That means that given the digit is a 0 there is a 15% chance that the spot 3,5 has a pixel in it.

Using this array I could now apply Bayes rule and start making predictions on the test data. For each test digit, I computed the probability that it is a 0 or 1 or 2 or ... or 9. To compute the probability that a given test digit is 0 for example, you go through each pixel of the test image. If there is a pixel in that spot you multiply by the probability in the array we computed earlier, and if there is no pixel in that spot you multiply by 1 subtracted by that probability. After this process I ended up with an array with the probability that the test data is a 0 or 1 or 2 or ... or 9. Then you select the biggest probability and the digit that corresponds to that probability is the prediction.

Analysis of Naive Bayes Classifier on Digits:



Faces

The Naive Bayes Classifier for faces works very similarly to the Naive Bayes Classifier for digits. The only difference is the number of classes. Digits has 10 (one for each digit). Face is binary so only has two classes. So when computing the probabilities only two need to be computed. The two are the probability that the picture is a face and the probability the picture is not a face. The greater probability is picked out of the two and the picture is

labeled based on that.

Analysis of Naive Bayes Classifier on Faces:

```
10% of training data: Mean = 0.7213333333333334 Standard Deviation = 0.09444575162494076
20% of training data: Mean = 0.8146666666666667 Standard Deviation = 0.01849924923401548
30% of training data: Mean = 0.8453333333333334 Standard Deviation = 0.018499249234015507
40% of training data: Mean = 0.864 Standard Deviation = 0.024312776705080627
50% of training data: Mean = 0.864 Standard Deviation = 0.021395742047841636
60% of training data: Mean = 0.8893333333333333 Standard Deviation = 0.010110500592068741
70% of training data: Mean = 0.8853333333333333 Standard Deviation = 0.010954451150103305
80% of training data: Mean = 0.8933333333333333 Standard Deviation = 0.010540925533894577
90% of training data: Mean = 0.908 Standard Deviation = 0.005577733510227167
100% of training data: Mean = 0.9066666666666666 Standard Deviation = 0.0
```

```
Time per training picture: 0.0042242168587221535
```

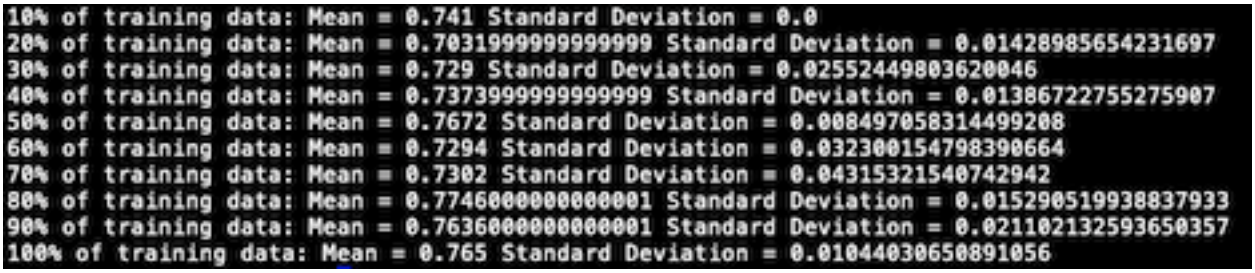
Perceptron Classifier

Digits

For the Perceptron Classifier for digits the algorithm works by using a weight vector. Each digit starts with a weight vector that contains 28 * 28 = 784 elements (one for each pixel). The element in the weight vector starts at 0. The algorithm then trains for 10 seconds. It trains by traversing every single image as many times as possible in the given 10 seconds. For each image the algorithm multiplies each pixel by all 10 weight vectors. So for example if a picture has a pixel in spot 0,0, but does not have a pixel in spot 0,1 the program would do $1 * w[0][0] + 0 * w[0][1] + ...$ This process would be repeated for ever spot in the picture and for every weight vector. At the end all the sums are compared and the digit with the greatest sum becomes the prediction. If the prediction is true no weight vectors are changed. However, if the prediction is wrong we do the following: The weight vector of the predicted digit is subtracted by the features of the digit. The weight vector of the true labeled digit is added with the features of the digit. This increases the chances, that if this same image was tested again the program would label it correctly.

After the 10 seconds of training the weight vectors are used to make predictions on the test images. Each pixel in the image is multiplied by all the weight vectors. The sums are added up just like in the training phase, and the greatest value is picked. The digit corresponding to that value is the prediction.

Analysis of Perceptron Classifier on Digits:



Faces

The Perceptron Classifier for faces works very similarly to the Perceptron Classifier for digits. The only difference is that the digits algorithm had 10 weight vectors, while the faces one only has two. That is because there are only two classes in the face algorithm: either the picture is a face or it is not. Besides that fact, the algorithms work exactly the same.

Analysis of Perceptron Classifier on Faces:

```
10% of training data: Mean = 0.6066666666666667 Standard Deviation = 0.015289834542697
20% of training data: Mean = 0.7733333333333333 Standard Deviation = 0.00932023411
30% of training data: Mean = 0.8333333333333334 Standard Deviation = 0.0252432834978
40% of training data: Mean = 0.8466666666666667 Standard Deviation = 0.03528231421
50% of training data: Mean = 0.8333333333333334 Standard Deviation = 0.015289834542697
60% of training data: Mean = 0.86 Standard Deviation = 0.02110245426913357
70% of training data: Mean = 0.8466666666666667 Standard Deviation = 0.059365037
80% of training data: Mean = 0.82 Standard Deviation = 0.021102132593650357
90% of training data: Mean = 0.86 Standard Deviation = 0.010490232891056
100% of training data: Mean = 0.8466666666666667 Standard Deviation = 0.0103065542697
```

```
Time per training picture: 0.02301116846088824
```

K Nearest Neighbors

Digits

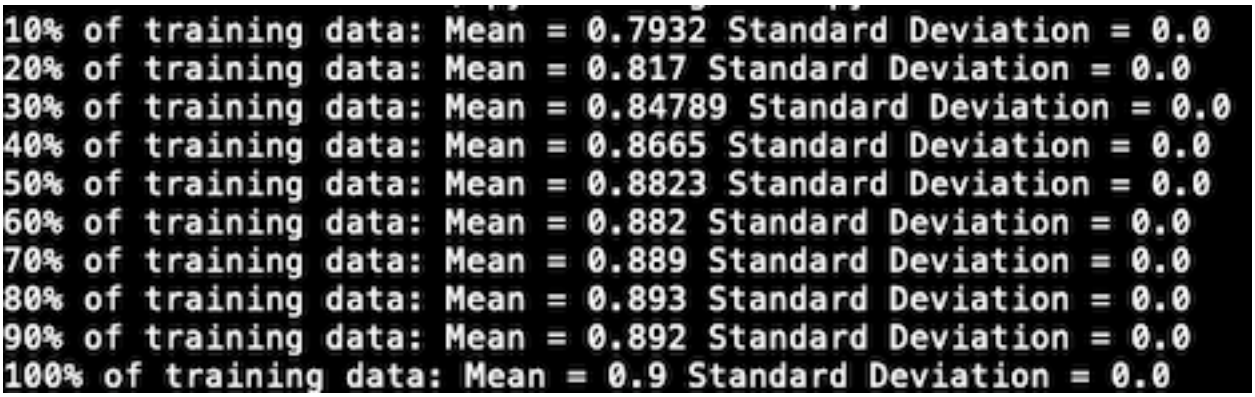
The last algorithm I chose to implement was K Nearest Neighbors. This algorithm works by taking each test image and comparing it with each training image. So for every single test image I computed the euclidean distance between the feature vector of the test image and every single training image. I computed the euclidean distance using the formula for finding the distance between two vectors:

$$= \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}.$$

After computing all the euclidean distances, I wanted to find the k smallest distances. I set my k as 5, so for every test image I found the 5 training images that were closest to the test image. Then I looked at the labels for each training image, and the one that occurred most frequently was my prediction for that test image. This process was repeated for every single test image.

Analysis of K Nearest Neighbors on Digits (Note: Since this algorithm takes

multiple hours to run I was only able to run one iteration of it, so that is why the Standard Deviation is 0 throughout).



Faces

The K Nearest Neighbors algorithm for faces is almost the exact same as the K Nearest Neighbors algorithm for digits. The only difference is that instead of 10 options for a prediction (one for each digit) there are only two.

Analysis of K Nearest Neighbors (Note: Since this algorithm takes multiple hours to run I was only able to run one iteration of it, so that is why the Standard Deviation is 0 throughout).

```
10% of training data: Mean = 0.4866666666666667 Standard Deviation = 0.0
20% of training data: Mean = 0.49333333333333335 Standard Deviation = 0.0
30% of training data: Mean = 0.5733333333333334 Standard Deviation = 0.0
40% of training data: Mean = 0.7533333333333333 Standard Deviation = 0.0
50% of training data: Mean = 0.7266666666666667 Standard Deviation = 0.0
60% of training data: Mean = 0.7733333333333333 Standard Deviation = 0.0
70% of training data: Mean = 0.7066666666666667 Standard Deviation = 0.0
80% of training data: Mean = 0.76 Standard Deviation = 0.0
90% of training data: Mean = 0.7466666666666667 Standard Deviation = 0.0
100% of training data: Mean = 0.76 Standard Deviation = 0.0
```

```
Time per training picture: 0.42033042400216314
```

Analysis

Digits vs. Faces

For both Naive Bayes Classifier and Perceptron Classifier the algorithms yielded higher accuracy for predicting faces. This makes sense, because in digit prediction the algorithm has to choose between 10 digits, while in face prediction the algorithm only has two choices, this means it is more likely that the algorithms will be able to correctly guess face or not face than a specific digit.

Time

The most time efficient algorithm was the Naive Bayes Classifier. For the Perceptron Classifier, each training phase was given 10 seconds to run. For this algorithm the time per training picture is irrelevant, because time is independent of the amount of training data the algorithm learns from. The K Nearest Neighbor algorithm was by far the slowest. For digits it took 3 hours and 48 minutes to run one iteration. In comparison with Naive Bayes, K Nearest Neighbors is about 675 time slower.

Accuracy

Even though the algorithms took varying times to run, the accuracy was generally similar. This is because the same features were used for every single algorithm. For Naive Bayes and Perceptron the accuracy for predicting digits was about 0.77, while the accuracy for predicting faces was near 0.9. This was reversed for the K Nearest Neighbors algorithm where the accuracy for predicting digits was 0.9, and the accuracy for predicting faces was 0.76.

Conclusion

In conclusion, all these algorithms were able to give good accuracy for both predicting digits and faces. The Naive Bayes Classifier and the Perceptron Classifier make more sense to use, because they are significantly faster, than K Nearest Neighbors and give similar accuracy.