198:440 Project 2
Face and Digit Classification

DeCaro, Drew

Lin, Zhengyuan

Sheikh, Saydain

August 9, 2020

In this project, we classified digits and faces by extracting their features, and then applying those features to the naive bayes and perceptron algorithms.

To extract the digit training features, I created a list that would hold 10 Digit objects that I created. The Digit object had a field for the label (whether it was 0, 1, 2, etc.), a field for the "count" which held the amount of times each digit appeared in the training data, and a list of matrices. I first iterated through "traininglabels" and created an array of those values, and I then set each Digit object's "count" to the amount of times I found its respective number. Then I scanned each character of "trainingimages", and added those characters into 2D array. At the end of each image, I would add this 2D array to its respective Digit object's matrix list. By the end, the matrix list for each digit would have a length equal to the "count" field mentioned earlier. So in short, the features for digits were simply the value at each matrix index (or pixels), which would be either a '+', a '#', or a space.

I followed the same procedure to extract the training features for faces, but I only needed a list of 2 Face objects, to hold values for images that were either faces or not faces. I once again created a 2D array for each image in "facedatatrain", but this time rather than use each pixel as a feature, I divided the 2D array into 10x10 submatrices, and I counted the number of #'s I saw in each submatrix and used the counter of each submatrix as the features. Initially, I tried using individual pixels as features, but I got low accuracy in the test cases. I attribute this to a few reasons. The face images are far larger than digits, and also contain a higher ratio of spaces. Also, the images in the training set that were labeled as faces, varied much more significantly than, for example, all of the digits that were labeled as a "one". In addition, there were much fewer training images. By switching the features to the frequency of #'s found in a 10x10 submatrix, I yielded much higher accuracy.

Once again, I used the same procedure to extract the features for test cases, but for these, I only added each matrix to a list, since I am unaware at that point what classification the images fall under.

To implement the naive bayes algorithm I first needed to calculate $P(D \mid C)$ for each different classification (C). To do this, I went through each pixel of the testing image, and compared it to the same pixel in each image of the first classification, zero. Every time I found a matching pixel, I would increment a counter that represented the probability of that pixel being a match with that classification. After I have gone through all of the pixels, in all of the images for a single classification, I would then multiply together all of the counters I created. This multiplied sum is my $P(D \mid C)$, and I would then multiply this by P(C), which is the amount of times the classification appeared in the training set to get $P(C \mid D)$. After doing this for every classification, I would select the highest $P(C \mid D)$ to make my prediction. I used the same procedure for face images, but as mentioned earlier, I used the number of 's found in a 10x10 grid as my features. To add, I experimented with the smoothing technique a bit. For some reason, I found lower accuracy when I implemented a smoothing counter for digits, but by adding a smoothing counter for faces, my accuracy improved. This is likely because images in the same digit classification are very similar to each other, so if there is a single pixel that does not have a match to a classification, it likely does not belong. This is not the case for faces though, as all of the images in the face classification are significantly

different from each other.

To implement the perceptron algorithm, I initialized a set of weight vectors for each classification to zeroes. For digits I needed 10 weight vectors, but for faces I only needed 1. For every image in the training set, I would extract its features in the same manner mentioned earlier. Then I created a feature vector for each image by representing a space as a 0, a as a 1, and a + as a 2. Then, I checked the image's label and iterated through all of the possible labels (0, 1, 2, 3, etc). When I found the match for the image's label, I would compute the F value by multiplying the weight and feature vectors and then add the zero weight (the first weight in the equation that does not get multiplied by any feature). Being that we have a match in this scenario, if the F value is not greater than 0, then I needed to increment all of the weights by their respective feature, and increment the zero weight by 1. I do this until the F value is larger than 0. In the 8 other scenarios where the image's label does not match a certain classification, I would again calculate the F value, and if it was not smaller than 0, I would decrement the weights until the F value is below zero. Being that the feature and weight vectors are both large and that there are many test cases, it was unlikely that I was able to find the perfect weight vectors in just one iteration of this. So I put this procedure inside of a loop that would only break if it went through all of the training images without needing to adjust any of the weight vectors. Especially as the training set grew closer to 100%, this became a very unlikely scenario, so I ended up breaking the loop after 300 iterations for digits, and 100,000 iterations for faces. I used many more iterations for faces because I used 10x10 grids as features, so it had much smaller weight and feature vectors, so each iteration took significantly less time. I found that with the higher number of iterations for this algorithm, I had much higher accuracy, but I needed to set a limit as it was becoming way too time consuming.

Stats:

Digits using Naive Bayes:

10% of training: average accuracy = 41.9%; standard deviation = 8.9
20% of training: average accuracy = 46.59%; standard deviation = 1.13
30% of training: average accuracy = 48.85%; standard deviation = 4.73
40% of training: average accuracy = 52.95%; standard deviation = 4.03
50% of training: average accuracy = 59.69%; standard deviation = 8.34
60% of training: average accuracy = 58.45%; standard deviation = 1.63
70% of training: average accuracy = 59.12%; standard deviation = 3.04
80% of training: average accuracy = 60.12%; standard deviation = 3.81
90% of training: average accuracy = 59.93%; standard deviation = 2.73
100% of training: average accuracy = 62.10%; standard deviation = 0

Calculating all probabilities with 100% training data took an average of 1920.54 seconds

Digits using Perceptron:

10% of training: average accuracy = 70.80%; standard deviation = 5.42
20% of training: average accuracy = 74.45%; standard deviation = 2.02
30% of training: average accuracy = 73.65%; standard deviation = 2.19
40% of training: average accuracy = 71.00%; standard deviation = 3.87
50% of training: average accuracy = 68.40%; standard deviation = 1.98
60% of training: average accuracy = 67.4%; standard deviation = 3.67
70% of training: average accuracy = 68.05%; standard deviation = 4.94
80% of training: average accuracy = 66.80%; standard deviation = 3.21
90% of training: average accuracy = 66.70%; standard deviation = 2.82
100% of training: average accuracy = 72.90%; standard deviation = 0

Training phase with 100% training data took an average of 1383.93 seconds

Faces using Naive Bayes:

10% of training: average accuracy = 53.33%; standard deviation = 3.73
20% of training: average accuracy = 54.67%; standard deviation = 2.74
30% of training: average accuracy = 56.00%; standard deviation = 1.30
40% of training: average accuracy = 49.67%; standard deviation = 2.51
50% of training: average accuracy = 43.67%; standard deviation = 3.95
60% of training: average accuracy = 58.33%; standard deviation = 2.62
70% of training: average accuracy = 62.33%; standard deviation = 1.63
80% of training: average accuracy = 64.67%; standard deviation = 2.39
90% of training: average accuracy = 68.33%; standard deviation = 2.54
100% of training: average accuracy = 72.00%; standard deviation = 0

Calculating all probabilities with 100% training data took an average of 3393.47 seconds

Faces using Perceptron:

10% of training: average accuracy = 58.67%; standard deviation = 4.48
20% of training: average accuracy = 62.3%; standard deviation = 3.92
30% of training: average accuracy = 72.33%; standard deviation = 5.42
40% of training: average accuracy = 80.00%; standard deviation = 4.34
50% of training: average accuracy = 74.00%; standard deviation = 2.11
60% of training: average accuracy = 68.67%; standard deviation = 1.41
70% of training: average accuracy = 77.67%; standard deviation = 2.12
80% of training: average accuracy = 68.00%; standard deviation = 4.94

90% of training: average accuracy = 71.54%; standard deviation = 3.84
100% of training: average accuracy = 76.00%; standard deviation = 0

Training phase with 100% of data took an average if 388.29 seconds.

The statistics show that the perceptron algorithm executed with higher overall accuracy than the naive bayes algorithm. It is also notable that perceptron performs relatively well even with less training data. One reason to justify why the testing accuracy would ever be higher for a lower percentage of training, is simply the possibility that the randomly selected training data selected highly favorable training examples, or in other words, training examples that are a better representation of the testing data. One reason why is that naive bayes assumes that features are conditionally independent. Something that I personally find illogical with naive bayes is how we multiply $P(D \mid C)$ by the frequency that classification C appears in the training data. For example, if the training data contained 1 zero, and 5000 ones, the algorithm will most likely classify a zero as a one, because it's frequency is far higher. In other words, this part of the algorithm depends heavily on the training data being an honest representation of what we are likely to see in the testing data, but this may not be the case in reality. Another reason why I view the perceptron as a better algorithm, is its emphasis on the training stage of machine learning process. It separates a portion of its work to adjusting the weights of each feature until they can properly classify every image as its proper label. In addition, I found that the longer it spends in this training procedure, the higher the testing accuracy will be. On the contrast, naive bayes does not adequately separate training and testing, because it depends on finding similar features in every training image for each testing image. As the training images increase, naive bayes becomes really inefficient, but since perceptron separates the two stages, it is much significantly more efficient. Overall, we have seen that is a better idea to create linear expression to classify data, than to search for similar features.

While we successfully passed the accuracy threshold when using 100% of the training data, improvements are still possible. One thing worth experimenting with are the different ways we can collect features. As mentioned earlier, I needed to switch the face features to 10x10 grids to get higher accuracy. If given more time, I would have tried a similar strategy for digits to see if I can increase the accuracy. The main thing to improve for perceptron would be to simply increase the training, being that it is separate from testing. As mentioned earlier, the accuracy increased as I performed more iterations of the training algorithm. I'm curious how accurate it could be if I were to allow the algorithm to run for an entire day.