Classifying Images of Numbers Using First Nearest Neighbor and Principal Component Analysis

Chase Perry

October 19, 2019

1 Problem Definition

Automating various processes has recently become a point of focus for many industries, as there are countless jobs and tasks that could be replaced by a machine that frees up a persons time to focus on more high level tasks. Sorting items, data retrieval, and organization are all such activities that while important, don't necessarily require a great deal of thought to manage. What they do require however is time, and when a person is left to deal with these tasks they are being taken away from focusing on more intensive and beneficial tasks that benefit their business or company more.

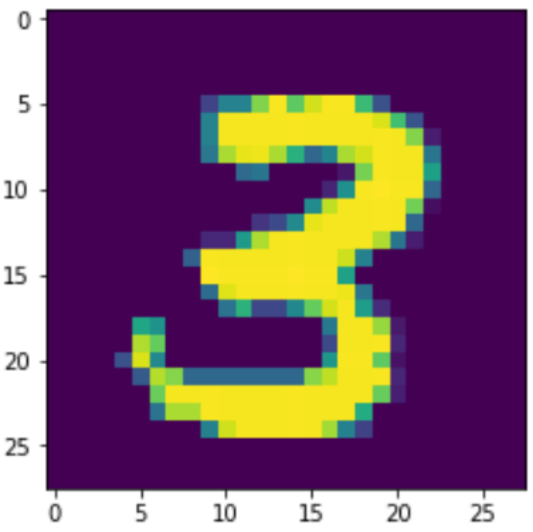
Image classification, say for the purposes of sorting mail, has become a hot topic for these reasons. Taking a picture of a number and correctly identifying what number is in that picture, and doing so efficiently, is a stepping off point for this entire process. While identifying each image with 100% accuracy is outstanding, if we have to spend 20 doing so for each image then the accuracy means nothing, as a business won't consider that a viable solution. In this project I will use First Nearest Neighbor to decide the class of the image, and Principal Component Analysis to minimize the comparisons needed to accurately make this prediction. This will hopefully yield a potential solution for businesses to consider to minimize wasted manpower and cost.

2 Data Set Source

The data set used for this project is the Modified National Institute of Standards and Technology database of handwritten figures, using sixty thousand training images and ten thousand testing images. This database was built by 'remixing' data from the National Institute of Standards and Technology, which was done in an effort to improve the variations and patterns seen in the images. The original NIST database was built using a training set of images written by U.S. Census Bureau employees while the testing set consisted of images written by U.S. high school students, as such there were likely to be some key differences between the figures written in training versus those tested against.

While the training and testing sets were in no particular order, as in all images of zero were first and all images of nine were last, they were shuffled during each run to ensure that there was minimal chance of creating a bias towards a specific class of digit. The most frequently occurring number in the training set was the number '1', and the least occurring figure was '5' though they were still relatively close in overall samples. Because of this difference however, there could be a potential bias to the figures though this remains to be seen after actually attempting to classify each image.

The original dataset obtained from MNIST also contains each image as a 784 value vector, which when reshaped into a 28 x 28 matrix yields each number similar to that in Figure 1. Since K-Nearest Neighbors isn't dependent on the data being in a specific shape, other than having the same number of attributes, each image was left as the 784 value vector for classification purposes and only reshaped when it was necessary to visualize the image stored.



Example Image Of The Number '3' As Stored By MNIST

3 Proposed Solution

The general strategy by which each test image will be classified includes the following steps: producing the co-variance matrix between the testing set and the mean image derived from the training set, getting the eigenvectors from this co-variance matrix, and then use this vectors to project each image from 784 dimensions to some lower dimension. Once the entire training and testing sets have been projected from their original 784 dimensions to some lesser dimensional space each testing image will be compared to the projected training set, with the purpose of finding its nearest neighbor in this new space. This nearest neighbor will be our estimated class of the testing image, and thus used for identifying the accuracy of our prediction.

Our attempts to classify each test image will be based on two main variables which we will control, these being the number of training images used to test against and the number of dimensions each image is projected onto, and by what metric we will determine the nearest neighbor. First, the reasoning for changing the number of training images is that the larger the pool is from which we can 1. produce the co-variance matrix and 2. compare against to decide the nearest neighbor the better. A smaller training pool will likely yield less diversity between images of the same class, meaning we're less likely to correctly identify images that look different from the 'typical' member of that class. Just how much this changes our accuracy however is what remains to be seen.

We will also be changing the number of dimensions each image is projected onto, as the fewer dimensions we move them to the less data we'll be throwing away and vice versa. As each image originally exists in 784 dimensions projecting down to 1, 2, or 5 dimensions means we'll be throwing away a good deal of information. However, how much this information means to the overall class prediction is something that we'll be testing against as minimizing the number of dimensions considered means minimizing the number of comparisons that are needed before making a class prediction.

The metric by which we will be judging how close each point is to each other in it's new dimension will be the Euclidean Distance. This metric was used due to its overall simplicity and because it had already been prepared for another project on a similar topic. The equation itself can be found in 1, where P is the test image we are currently trying to classify, X$_n$ is one such training image, 'i' represents one dimension or feature each image contains, and 'D' is the total number of dimensions each image has.

$$d(P, X_n) = \sqrt{\sum_{i=1}^{D} (P_i - X_{n_i})^2} \tag{1}$$

4 Solution

To begin, several functions were defined to manage the better part of the project so that a couple of standard for loops could be used changing the variable described above. First, was the overall definitions of the function which were used to get the Euclidean Distance between the test image and the training set, which can be seen in listing 4. While this function is relatively simple, it is vital to the rest of the project as if it functions incorrectly then the incorrect image could be chosen as the nearest neighbor to the test image.

```python
def euclidean(train, test):
  return np.sqrt(np.sum(np.square(train - test), axis=1))
```

After this function was defined another 'master' function was defined to handle the rest of the PCA and 1^{st} Nearest Neighbor logic, this can be found in listing 4. This function a number of inputs in order to function correctly, this includes (in order) the training dataset (already shortened to only include a certain number of images), the average image across the entire training set, the co-variance matrix, and the collection of eigenvectors sorted by impact on the data. It also requires the training classes, testing dataset and size, the function used to determine the nearest neighbor, and the number of dimensions to project both the training and testing datasets onto. First, it trims the testing set to the proper size based on what was passed in, creates an accuracy matrix, and gets the subspace projection matrix. Next, it projects the entire training dataset and shortened testing dataset into the new dimensional space and uses these projections to determine the nearest neighbor and class of each testing image.

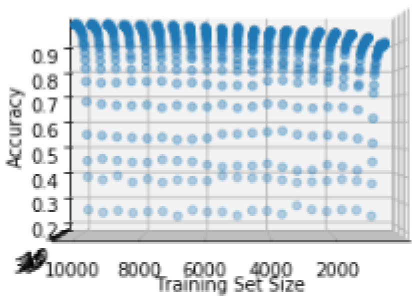
```
def test_classification(trainX, mean, cov_mat, eig_pairs, trainY, ...
    testX, testY, test_size, funct, num_dim):
  testX = testX[:test_size, :]
  acc = np.zeros((10, 10))
  subspace = np.array([eig_pairs[i][1] for i in range(num_dim)]).T

  trainX = trainX + mean
  trainX_pca = np.dot(trainX, subspace)

  testX_pca = np.dot(testX, subspace)

  for i in range(0, test_size):
    distances = funct(trainX_pca, testX_pca[i])
    loc = np.argmin(distances)
    pred = int(trainY[loc, 0])
    actual = int(testY[i, 0])
    acc[actual, pred] += 1
  return acc
```

The return of the function in 4 was then used to define the accuracy of every test. As the return was the accuracy matrix the sum of the main diagonal divided by the overall number of tests yielded the percentage of correct classifications per training size and dimensions. This value was used for analysis of benefit/loss based on the changes of the variables, and will be discussed later. There was one final function defined which obtained a number of the variables needed for the function defined in listing 4 however as this was primarily used as a helper function it is not necessary to share it here.

5 Evaluation

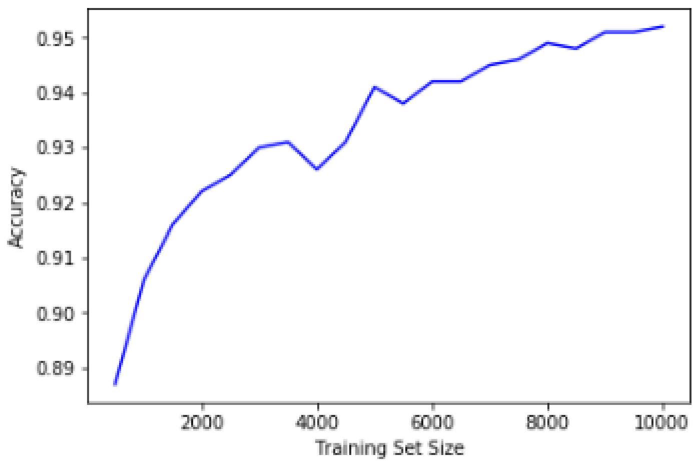
5.1 Changing The Training Size

To begin, we'll examine how the accuracy changed with regards to the size of the training set while the number of dimensions are held constant. As can be seen in Figure 2, while the accuracy across training set sizes is close for each selected number of dimensions those with the greater size usually performed better overall. This is especially evident towards the top, where the data was projected onto 30 dimensions, as the data shows a steady decline as the training set size was decreased.



Accuracy Using Euclidean Distance Projected onto 30 Dimensions

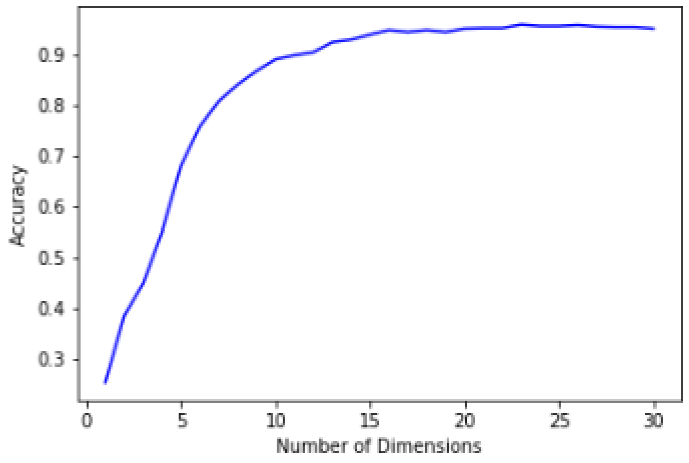
This can also be seen in Figure 3, where as the size of the training set increases the accuracy does as well. For this figure specifically, the number of dimensions projected onto was held at 30 so there was no cause of error from that variable. It is also evident here that around a size of 3500-5000 the accuracy varied a bit questionably with the quick drop and then sharp increase, this is somewhat visible in Figure 2 as well but only when the dimension is 1.



Accuracy After Projecting Onto 30 Dimensions, Varying Training Size

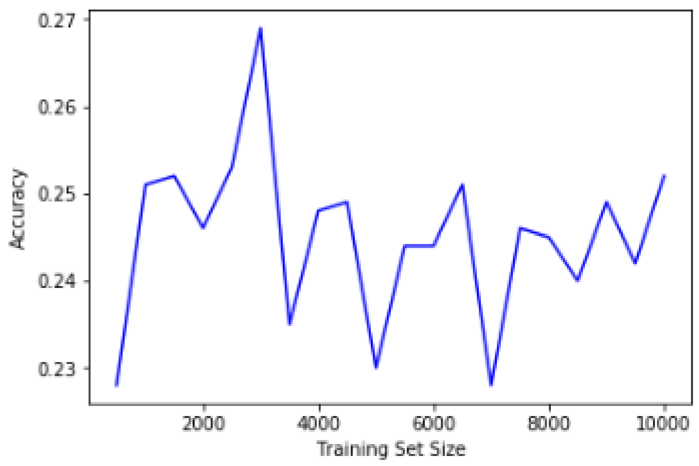
5.2 Changing The Number of Dimensions Projected Onto

The other main variable tested in this project was how the number of dimensions projected onto effected the overall accuracy of the class predictions. For any figures in this section, unless otherwise stated, the training set size was held at 10000 as this was the most accurate size based on the data in Figure 3. To begin, as the overall number of dimensions increased the accuracy increased however it faced a more rapid increase and drop off in the benefit of the number of dimensions. As can be seen in Figure 4 there was a sharp increase in accuracy when the number of dimensions went from 1 to around 10 or 15 but from that point onward to 30 dimensions there was next to no change in accuracy. This lack of change indicated that there is likely no benefit to the accuracy of classifications and yet we'd increase the number of comparisons needed to make to get the Euclidean Distance.



Accuracy Using 10000 Training Images

Likewise, as the number of dimensions decrease we become increasingly more unstable with our predicted classes. Specifically, as can be seen in Figure 5, we have next to no consistency in the accuracy when the number of dimensions are too low. This is likely due to just how much information we throw away when projecting down to so few dimensions, especially given the fact the each image originally exists in 784 dimensions. As discussed above, as the training set size increases we would expect to see a general increase in the accuracy but here this was not the case. While the change in accuracy was minor between the change in training size their was still a noticeable lack of standardization among the change.



Accuracy After Projecting To One Dimension

6 Discussion

Based on the data collected among the different tests, I believe that while an increase in the size of the training set provides a solid, steady increase in the accuracy the same cannot be said of the number of dimensions. There is a noticeable increase in the accuracy of our classifications but this is only up to a certain point, after which the benefit from increase the number of dimensions (and thus the number of comparisons) ceases. This indicates that there is likely a sort of 'Goldilocks' zone where the size of our training set and the number of dimensions projected onto yield a maximum accuracy while also providing the minimum number of calculations to finding the principal components and minimizing the comparisons between a new test image and the overall training set.

7 Future Work

In my opinion, the next step to take with this kind of project would be to test out different metrics for calculating the nearest neighbor to use for classification. In this experiment the Euclidean Distance was used to test accuracy so that the other variables could be held constant without too much mixup, however other metrics could potentially exist that could yield a better accuracy. Another potential area for further study could be to examine the images that were misclassified and see if there are any noticeable features that could cause this. As there were 1000 images that were classified it wasn't very possible to search through each one for potential causes for error. But, if those that were misclassified could be identified as having key features (such as a stray marking or such) then the images could potentially be 'cleaned' up to exclude stray marking that aren't like any other images found in the training set.

8 What I've Learned

Through this project I've learned a bit about optimizing code to work better when the same operations are being performed across multiple items. For instance, in an earlier version of this project I was calculating the projected test image one per image, rather than the entire testing set projected all at once. This alone was causing extremely poor performance when attempting to gather data for this project, with upwards of 30 minutes of run-time in attempt to get the data for the Euclidean Distance. After correcting this and doing all the projections at once the overall run-time was shortened to close to 7-10 minutes speeding up efficiency enormously.