Benny Tsai
600.425
Declarative Methods
Homework 3

HOW-TO:

First, run the "build" script to construct the extended data sets and the standard drawings library (discussed below).

The individual "classify" scripts will each produce files "training_error" and "test_error" for the corresponding classifier.

Alternatively, use the "run_all" script to run all classifiers sequentially; the script will produce a set of training and test error files for each classifier to be viewed at the end.

1) a. k = 1 training error = 0 (0%)
test error = 0.0351515 (3.5151%)
k = 3 training error = 0.0206667 (2.0667%)
test error = 0.0387879 (3.8788%)
k = 25 training error = 0.0626667 (6.2667%)
test error = 0.070303 (7.0303%)

b. training error = 0.000166667 (0.0167%)
test error = 0.0375758 (3.7576%)

c. Class 0 training error = 0.00984454 (0.9844%)
Class 1 training error = 0.00950992 (0.9510%)
Class 2 training error = 0.00983232 (0.9832%)
Class 3 training error = 0.00999570 (0.9996%)
Class 4 training error = 0.00999159 (0.9991%)
Class 5 training error = 0.00996470 (0.9965%)
Class 6 training error = 0.00961185 (0.9612%)
Class 7 training error = 0.00978935 (0.9789%)
Class 8 training error = 0.00996923 (0.9969%)
Class 9 training error = 0.00987840 (0.9878%)
test error = 1.57576% (26 errors)

2) 1-nearest neighbor training error = 0 (0%)
test error = 0.0884848 (8.8485%)
3-nearest neighbors training error = 0.0465 (4.6500%)
test error = 0.0860606 (8.6061%)
25-nearest neighbors training error = 0.128167 (12.8167%)
test error = 0.124848 (12.4848%)

Neural Net training error = 0.000166667 (0.0167%)
test error = 0.0339394 (3.3939%)

Support Vector Machine	Class 0 training error = 0.00999355 (0.9993%) Class 1 training error = 0.00998050 (0.9980%) Class 2 training error = 0.00992560 (0.9926%) Class 3 training error = 0.00999713 (0.9997%) Class 4 training error = 0.00994736 (0.9947%) Class 5 training error = 0.00998938 (0.9989%) Class 6 training error = 0.00997084 (0.9971%) Class 7 training error = 0.00999075 (0.9991%) Class 8 training error = 0.00998747 (0.9987%) Class 9 training error = 0.00996780 (0.9968%) test error = 5.81818% (96 errors)

I chose to use 3 different techniques to augment the original features with 2 additional feature sets. First, the original greyscale values (from –1.0 to 1.0) are re-scaled to range from 0.0 to 1.0. Next, a blurring filter is applied such that each pixel takes on the average value of its original value and its neighbors' original values. The blurred, scaled value of each pixel is written out as 256 new features. Finally, I measure the "difference" between an image and standard drawings of each of the 10 numbers (built up from training examples) by summing the difference in greyscale values over all pixels in the image and each standard drawing. The "difference" scores are written out as 10 new features.

The intuition I followed here is that given standard drawings of each digit, categorizing new digits by comparing them with the standard drawings should yield good results. The standard drawings are simply the average composite of all blurred, scaled versions of training examples for a given digit. For example, the standard drawing for 0 is built up by averaging the sum of all blurred, scaled versions of drawings of 0 in the training data. The reason I apply the blurring filter is so that the classifiers will be more forgiving of minor differences in images.

Unfortunately, the blurring may have done too good a job. As the data demonstrates, in most cases the new features result in a decrease in classification accuracy! My hypothesis is that by building the standard drawings from blurred images, some of them may have ended up too similar to be useful. The standard drawings of 8 and 0, for example, are exceeding similar. Fortunately not all is lost; I was relieved to see that in the case of neural nets, my new features actually improved performance, though not by a large margin.

In the brief research that I did prior to beginning the project, I read that most real-time systems attain high accuracy in hand-written digit recognition by noting the direction of strokes as the digits are written. This is a great idea, but it doesn't seem to be applicable to something like scanning of documents, where we cannot know reliably the direction of strokes. Hand-written documents are probably the most troublesome. In this case, the idea of building up standard drawings may be useful, given that a person's handwriting does not change significantly between instances. I'm imagining something like the following scenario:

Suppose we wish to scan a certain handwritten document, but the handwriting is particularly atrocious. We can build up a library of standard drawings for a person by scanning in an image which value we know from another source. For instance, if we have the image of a date on a check, and we know the date the check was signed, we can then infer what digits are represented in the image and use this information to build up standard drawings of the digits involved.