O and X Recognition

ECE102
Chris Brown
Due: 1 May 2001
April 2, 2001

1 The Problem

Your problem is to distinguish digitized O’s from X’s. Within that problem, you get to make up the assumptions and the methods. Your final product is your program (preferably in Matlab, but can be your choice), and a written report explaining your method and the assumptions underlying it, results in the form of statistics or graphs, suggestions for future enhancements, final conclusions, and references if any. Your data can be made up by you (text editor) or if you are interested there are datasets on the web. Include your code and readme as usual in what you hand in. Transcripts of runs are nice.

There is a lot written on character recognition, and if you want to cite some of the literature to compare your algorithm to other existing ideas, fine. However, this is not meant to be a library research exercise as much as a thinking exercise.

2 But Wait...!

OK this is in some sense a solved problem. Here are two more that are actually research issues at places like our own Eastman Kodak Co. Automatically distinguish indoor scenes from outdoor scenes, and likewise distinguish upside-down from rightside-up photos. I’ll gladly accept either of these topics instead of the O’s and X’s problem. The same issues, *mutatis mutandis*, appear in all these classification problems. You’ll want different operators (feature finders), but the classification algorithm can be the same.

3 The Assumption Space

Depending on your assumptions, this problem can be trivially easy or very difficult. Here are some axes in the assumption space:

Spatial resolution of digitized data. For a working program I’d say between $5 \times 7$ and $16 \times 16$ is a reasonable size; smaller or larger is fine too. I’ve seen a database of $32 \times 32$ digits.

Greyscale resolution of data: array elements can vary in darkness from from 0 to (say) 255 instead of being binary 0 and 1.

Input is from a single known type font, from one of multiple known fonts, from some unknown type font, or handwritten. For handwritten, it is either “raw” or “preprocessed” (see examples below).
Don’t forget people who make diagonal crosses through their O’s, put loops at the top, or leave a space at the top. Or draw an X as two kissing semicircles something like this (or make an X with one wavy line and one straight one, or put serips on the X. However, given the plethora of really wild fonts available (My fave is Carol’s Chunks), handwritten may actually be easier than an unknown random font.

Invariance to systematic geometric variation: Translation, Rotation, Size variation (zoom), Skew (say italics vs. roman or different handwriting slants).

Noise: high-frequency (speckle in the background), low-frequency (blotchy low-contrast background variation), dropout (holes where there should be letter).

Let’s all assume that an input character is presented in an integer array, with each element representing the darkness of a picture element (pixel).

So for some examples of binary input: If background is a 0 and ink is 1, a 7 × 7 X might look like this (similar idea for O’s of course):

\[
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
\]

A rotation-invariant recognizer would accept this:

\[
\begin{bmatrix}
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0
\end{bmatrix}
\]

Maybe all the X’s are identical and there are no problems of noise, varying size or rotation. If size can vary and some shape variation is allowed, then these can be 5 × 7 Xs:

\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1
\end{bmatrix}
\]
If we allow a little additive noise and dropout, the first X above could look like

```
1 0 0 0 1 0 1
0 1 0 0 0 1 0
0 0 1 0 0 0 1
0 0 0 1 0 0 0
0 0 1 0 1 0 0
1 1 0 0 0 1 0
1 0 0 0 0 0 0
```

A handwritten X might be (noisily) digitized as:

```
0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 1
```

But there are pre-processing programs that can clean up such data to yield output something like this, which is thinned, has noise removed, dropouts filled, and is normalized to fit the whole array.

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1
1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 0 0
0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1 1 1 1
0 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0 1 1 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
```

4 Hints

Ideally, I'd like to see a clustering approach. My next favorite would be a decision tree. A neural-net approach could be good but be aware you may need many many training samples.

Clustering isn't mysterious. If you have N features, each with a real-number output (e.g. density of 1's, or number of diagonal edges in the upper left quarter of the images, or the B values of a B-binned edge-orientation histogram, WHATEVER N features, they're just an N-vector. You may want to normalize the values so that the amount they contribute to the distance metric will be scaled right. Scaling by their individual standard deviations is a good start.

Anyway, during training you can just keep the average of these vectors for each class and that'll be the mean vector for that cluster. Then to test, you get a new vector and calculate its Euclidean distance from these class-representative mean vectors and the nearest one is your classification.

Alternately, remember all the training vectors (in an array say) along with their classification (in a parallel array). For a new test vector, calculate which of all the training vectors is closest, and its class is your answer. This is "nearest-neighbor". There are many enhancements you can use, like k-nearest neighbor, principle component analysis to throw out useless features, etc. etc.

If you think about decision trees you'll see that they ask a sequence of questions, each of which divides feature space into two halves. So they are another way of getting a decision surface in feature space. My guess is that it takes a fair amount of ingenuity to devise a tree that can cope with oddly-shaped or closely-intertwined clusters.

My recommendation is to start with a problem over in the easy hyper-corner of assumption space and gradually relax assumptions until time runs out. Here's a sample simple problem and solution. All X's are made with (digitized) straight lines from corner to corner, all O's are made with (digitized) straight lines that fill up the left and right columns and the top and bottom rows. No noise. The algorithm is to look around the center of the input array for pixels. Any there means X, none means 0. This isn't an inspiringly difficulty problem but starting here means your low-level infrastructure (reading in data, accessing pixels, etc.) works. Next you might try to obtain translational invariance by using smaller characters that can translate around in the array. Then go for noise and dropout, thick letters, different fonts, rotational invariance, etc. etc.

As for methods, I'd love to see a comparison. If you make some elementary feature-finders (like the central density one above) then you can easily do a decision tree, a clustering approach, AND a neural net (use the matlab neural net routines or write your own!).