1 Image Preprocessing and Basic Operations

1.1 Edge Detection

The Sobel operator works by attempting to find areas of an image in which the rate of change is high. If the rate of change around a pixel is above a threshold it is marked as an edge. This rate of change is calculated by using the below kernels (figure 1.1.1) and finding the approximation of the gradient and then this gradient's magnitude.

dx:

-1	0	1
-2	0	2
-1	0	1

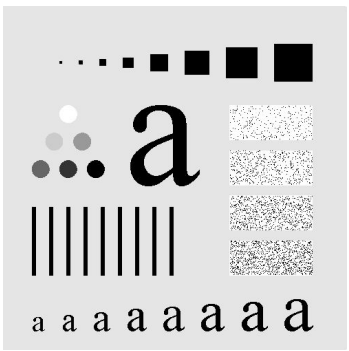
dy:

-1	-2	-1
0	0	0
1	2	1

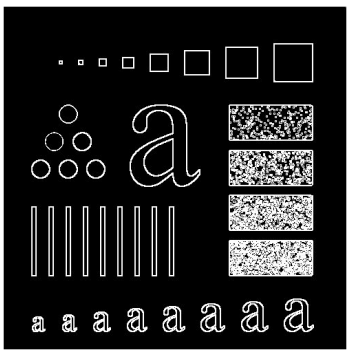
FIGURE 1.1.1

This process is done for all pixels in the image (sans the outermost edge of pixels). This is the convolution process.

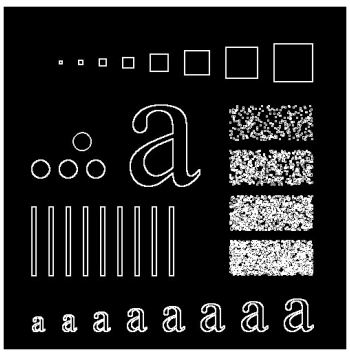
I have run the edge detection convolution process using the kernels in figure 1.1.1 and two different thresholds, one at 80 and one at 120. As we can see in figure 1.1.2, the lower threshold seems to have identified all edges, even those with a small amount of contrast, in the original image. The higher threshold pass missed two of the lighter circles and did not identify the edges of the ‘noise’ boxes as edges.



ORIGINAL



THRESHOLD AT 80



THRESHOLD AT 120

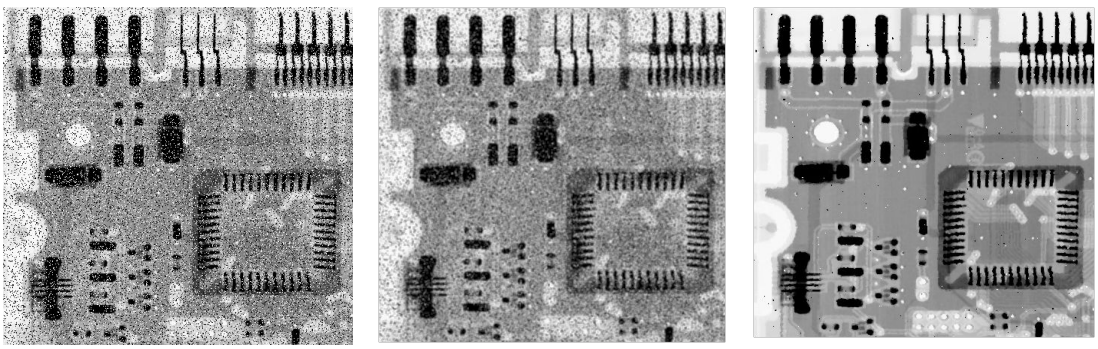
FIGURE 1.1.2

When attempting to identify edges around smaller objects, say an object of one pixel, the Sobel operator identifies all the pixels around that one pixel as an edge. This works out fine for objects which are more than one two pixels apart but when many objects are within this distance we could end up with mostly edges. This can be seen in the close up of the lowermost noise box (see figure 1.1.3). Here edges are denoted in white and non-edges are in black.

FIGURE 1.1.3

1.2 Noise Cancellation

There was a very noticeable difference found when removing noise with a mean convolution mask versus removing noise with a median filter. As can be seen in figure 1.2.1 the median filter performs significantly better than the mean filter, even so far as to bring out details not even noticeable in the original.



ORIGINAL 3X3 MEAN FILTER 3X3 MEDIAN FILTER

FIGURE 1.2.1

The mean mask does a poor job at removing noise (at least this type of salt and pepper noise) because it incorporates the noise into the image. The noise pixels are included

when determining the average and therefore contribute to the final rendered image. In addition to this, mean filters have the effect of “smoothing” an image which may not be the desired outcome.

In contrast to this, the median filter does a better job at removing the noise because it has the ability to actually remove the noisy pixels. It works on the assumption that in a 3x3 area (at least when using a median filter with a 3x3 neighborhood) the noisy pixels will be the extrema of the pixels in the neighborhood. Therefore these pixels are removed and not incorporated into the final image.

1.3 Image Enhancement

In order to enhance (sharpen) an image one wants to add more “contrast” and enhance edges and places of difference between one pixel and the next. This is the opposite of the smoothing which happens when applying a mean mask.

sharpen:

0	-1	0
-1	5	-1
0	-1	0

over sharpen:

-1	-1	-1
-1	9	-1
-1	-1	1

FIGURE 1.3.1

I attempted to use two different convolution kernels, both 3x3 masks (seen in figure 1.3.1). When applying the ‘over sharpen’ kernel the image became too sharp and ended up looking over processed (see figure 1.3.2). I do, however, believe that this kernel may have its uses in image processing, perhaps as a preprocessing step for edge detection to assist in highlighting the edges.

ORIGINAL SHARPENED IMAGE OVER SHARPENED IMAGE

FIGURE 1.3.2

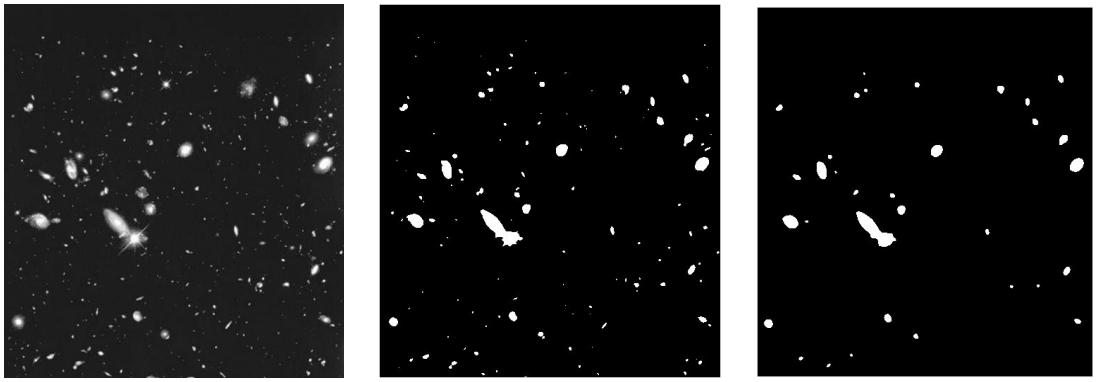
It would be very interesting to play with different kernels here to see what achieved better results. Perhaps rotating the 'sharpen' kernel above 45 degrees so that the zeros where in the north, south, east and west positions and the -1 at the angles. Or even seeing what effect a 1x3 or a 3x1 mask would have on different blurry images.

2 Mining Image Data

2.1 Mining Space Images

In order to process the hubble image I firstly smoothed the image. This was done by iterating over all the pixels (sans the outermost n pixels, n here depended on the size of the area involved in the averaging) and smoothing/averaging each. This averaging was achieved, not by using a mask but, by getting the average of all the pixels within the area around the pixel in question. Two loops from -1 to 1 (for a 3x3 area), one to move the inspected pixel north and south and one to move the inspected pixel east and west. This is the same technique used for the mean noise reduction above in question 1.2.

After each pixel average was found a threshold was applied to this averaged value. I played around with a couple of different thresholds (80, 100, 110, 127, 200, etc.). To my eye the setting the threshold to 127 (about half) seemed to perform the best.



ORIGINAL 3X3 MASK 127 THRESHOLD 7X7 MASK 127 THRESHOLD

FIGURE 2.1.1

Now in order to eliminate, or at least minimise the amount of, “far away” galaxies (this being defined by a smaller area of brightness) I tested out different sizes of the area involved in the averaging of pixels (another way to look at this would be to say that I was trying different size masks). Upon comparing a 3x3 smoothing with a 7x7 smoothing it was found that the 7x7 smoothing (plus the thresholding) eliminated mostly all of the smaller galaxies. We can see in the above figure 2.1.1 that the 7x7 mask at the same threshold removes most of the smaller galaxies and leaves us with an image representing the close galaxies.

I then use a small bash script to convert the resulting image from PNG (what I used for all my image processing) to a P4 pbm file then set the max grey level to 1 and all pixels to either 1 or 0 using the command line utility sed (see the question2 readme.md file).

2.2 Face Detection

During feature extraction I firstly worked on some standard pixel statistics features. I choose mean, standard deviation and entropy. Then thinking about some domain specific features I though about the symmetry in faces. Faces are mostly symmetric from the side to side when viewed from the front. The process here is to divide the image in half horizontally and then compare pixel statistics between the two. Here we do not include the mean in these statistics as it would be greatly affected by the lighting of the image. Therefore the squared difference for both the standard deviation and entropy were chosen as features. Next the image was divided into four equal quadrants and for each of these the first order moment was found with an origin of the center of the image. Meaning that the starting point for the moment calculation for the top left quadrant was the bottom left of that quadrant and the starting point for the bottom right quadrant was the top left of that quadrant.

These features were written to a CSV file which was used during the classification process.

Since this feature data was in the continuous space I choose to represent it as a normal Gaussian distribution. Then each class's (face/non-face) prior was found from the training set by finding the percent representation of each class in the training data. This prior was then multiplied with the likelihood of each feature, give the class. The ratio of these (face to non-face) was then recorded. Each feature's likelihood was found using a probability density function for that feature's distribution (assumed to be normal).

• Mean • Standard deviation • Entropy • Squared difference standard deviation • Squared difference entropy • Moment top left • Moment top right • Moment bottom left • Moment bottom right

FIGURE 2.2.1 FEATURE SUMMARY

The resulting ratios from both the face and non-face test sets were captured and used to determine the ROC and the area under its' curve (AUC). This was done using the methods in https://ccrma.stanford.edu/workshops/mir2009/references/ROCintro.pdf.

After running the classification using a variety of different feature combination I found the smaller feature set of only the two squared difference features and the entropy feature yielded the best ROC curve (see figure 2.2.2).

FEATURE SET	AUC
• Squared difference standard deviation • Squared difference entropy • Entropy	0.74958
• All extracted features	0.73221

FIGURE 2.2.2

I would have liked to have more time to investigate some more features. For example I would be interested in what smaller pixel box features would do (2x3 boxes). Or, a squared difference between the top left and top right quadrants. But one which I did spend some time looking into was a way to learn how to identify an eye, mouth or nose.

This question would become a bit recursive if I were to do that. Meaning we would start looking for features to help find features. Whereas I am sure that this is a common task in image processing and computer vision I choose to concentrate on the more general features.

3 Data Mining using Extracted Features

3.1 Object Recognition

File parsing

The first step in the Object Recognition task was to amalgamate all the different feature files into one tabular data set. I achieved this via some simple text file processing using a combination of unix tools. Using a bash script I firstly created a header line by counting the number of columns in each file and concatenating the filename (for example *mfeat-fac*) with the index of the column This yielded a header line similar to figure 3.1.1 below for each file.

mfeat-fac-1 mfeat-fac-2 ... mfeat-fac-216

FIGURE 3.1.1

Then for each file I used *awk* to to add line numbers to the front of each line. These line numbers where then used by the *join* command line tool (a simple tool to join two files together based on a common column, see the man pages for more info) to join all the files together into one file. The class designation of each line (0-9) was added in a column named *num_class*. I utilised *awk* and modular division to determine the current lines class. Afterwards this file was split into two, one for testing (*testdata*) and one for training (*trainingdata*) and any temp files created during this process were removed.

The resulting files are space delimited files with 100 sets for each class, a header line, columns for each feature and a column (the last column) which defines the class.

Training

For the training of the decision tree classifier I chose to use Python with the *sklearn* and *pandas* libraries for classifier and file loading respectively. This tool chain was chosen because of the robust support for visualisation of the tree and confusion matrix. The script *tree.py* produces a file *letters_dt.png* (and a *letters_dt.dot* file)

Decision Tree

Upon investigation of the constructed decision tree not all of the features were used in its construction. Out of the 649 features only about 40 are used (depending on the actually run of the tree). Of the 6 different feature “sets” the pixel averages where the most unused. Of the 240 2 x 3 windows only four were used in the construction of the decision

tree. This implies that there were four areas in the original images which had mean values different enough to help in the classification. In a way this is understandable as the majority of the image would be white and most of the actual writing (dark spots on the image) would be in the same general areas but at the same time would have a large variation when dealing with 2x3 areas. These features are not translation or rotation (when considering the whole number) invariant and do not well when dealing with hand written numbers which could be written in different areas of the image and at different angles and slants.

	PREDICTED
TRUE	**0** [98 0 1 0 0 1 0 0 0 0] **1** [0 90 2 1 1 1 0 1 0 4] **2** [0 2 90 3 0 0 0 5 0 0] **3** [0 2 0 91 1 2 0 4 0 0] **4** [0 3 0 1 92 3 1 0 0 0] **5** [0 0 0 3 1 93 0 0 0 3] **6** [0 0 0 0 1 0 99 0 0 0] **7** [0 1 2 2 0 1 0 94 0 0] **8** [0 0 0 0 0 0 4 0 96 0] **9** [0 0 0 0 0 1 0 1 0 98]

FIGURE 3.1.2 CONFUSION MATRIX – ALL FEATURES

Confusion matrix (6 morphological features)

After training the Decision Tree with only the 6 morphological features I was able to produce the confusion matrix see in figure 3.1.3. As can be seen when using only the morphological features the classifier had the hardest time identifying numbers which are morphologically similar. For example we can see from the confusion matrix that 6's are mistaken for 9 about as much as they are identified correctly as 9's and the same is true when identifying 9's (they are often mistakenly identified as 6's). The same is true for the other numbers which have similar shapes, i.e. they can be rotated or flipped and look like another number. As can be seen with the 2's and 5's. Although the 6's and the 9's are the most difficult to distinguish from one another.

	PREDICTED

TRUE

	0	1	2	3	4	5	6	7	8	9
0	98	0	1	0	0	1	0	0	0	0
1	0	92	1	0	2	0	0	4	0	1
2	0	1	67	10	3	10	1	8	0	0
3	0	0	10	52	19	11	0	8	0	0
4	0	8	1	**20**	52	2	0	16	0	1
5	0	0	**23**	**23**	5	49	0	0	0	0
6	0	0	0	1	0	0	50	0	0	**49**
7	0	1	3	6	9	3	0	78	0	0
8	0	0	0	0	0	0	1	0	98	1
9	1	0	1	0	0	1	**67**	0	0	30

FIGURE 3.1.3 CONFUSION MATRIX - 6 MORPHOLOGICAL FEATURES

Comparison (all features vs. morphological only)

It is very clear from analysing the two confusion matrices in figures 3.1.2 and 3.1.3 that when using all the features the classifier performs much better. It would be interesting, time permitting, to further investigate which features provide the greatest influence on the classification.