SPS – Letter Recognition Report (CW2)

* approach to analysis in the Fourier domain
* features you chose to separate the character data
* results of the Fourier domain analysis
* analysis of the classifier
* decision region plot(s) and their analysis for the classifier

Now **here's how the k-nearest algorithm works**. When given a vector, it checks all neighbors around it up till a distance of 'k'. Within these neighbors, someone one label will have the maximum number of vectors.

For example, consider the '0's dashed vector above. Every neighbor till a distance of 'k' has the label '0'. So the given vector must be a zero too. Simple as that.

Similarly for 5's dashed vector above. Neighbors till a distance of 'k' are all labelled as '5'. So the given vector is a 5.

Deciding k in K-nearest neighbors

Deciding a "good" k for your data is very important. If you choose a big K, you might end up including unwanted vectors in the neighborhood. If your K is very small, you might not have enough vector to correctly "identify" a label.

So how do you find a good k? Well, you use heuristics and find, by hit and trial, a decent value.

Disadvantages of K-Nearest neighbors

K-Nearest is the most simplest algorithm you can use for classifying things. It isn't optimized for speed or space.

You need lots of training samples to ensure lots of vectors are withing the 'k' sphere. And you need to have ALL training vectors in memory at all times. Then you compare a single test vector against LOTS of a training samples (to ensure they're within the 'k' sphere)/.

In short, its really slow and consumes a lot of memory.

|  |  |
| --- | --- |
| down voteaccepted | General idea  Option 1: Load both images as arrays (scipy.misc.imread) and calculate an element-wise (pixel-by-pixel) difference. Calculate the norm of the difference.  Option 2: Load both images. Calculate some feature vector for each of them (like a histogram). Calculate distance between feature vectors rather than images.  However, there are some decisions to make first.  Questions  You should answer these questions first:   * Are images of the same shape and dimension?   If not, you may need to resize or crop them. PIL library will help to do it in Python.  If they are taken with the same settings and the same device, they are probably the same.   * Are images well-aligned?   If not, you may want to run cross-correlation first, to find the best alignment first. SciPy has functions to do it.  If the camera and the scene are still, the images are likely to be well-aligned.   * Is exposure of the images always the same? (Is lightness/contrast the same?)   If not, you may want [to normalize](http://en.wikipedia.org/wiki/Normalization_(image_processing)) images.  But be careful, in some situations this may do more wrong than good. For example, a single bright pixel on a dark background will make the normalized image very different.   * Is color information important?   If you want to notice color changes, you will have a vector of color values per point, rather than a scalar value as in gray-scale image. You need more attention when writing such code.   * Are there distinct edges in the image? Are they likely to move?   If yes, you can apply edge detection algorithm first (e.g. calculate gradient with Sobel or Prewitt transform, apply some threshold), then compare edges on the first image to edges on the second.   * Is there noise in the image?   All sensors pollute the image with some amount of noise. Low-cost sensors have more noise. You may wish to apply some noise reduction before you compare images. Blur is the most simple (but not the best) approach here.   * What kind of changes do you want to notice?   This may affect the choice of norm to use for the difference between images.  Consider using Manhattan norm (the sum of the absolute values) or zero norm (the number of elements not equal to zero) to measure how much the image has changed. The former will tell you how much the image is off, the latter will tell only how many pixels differ. |

**HOG**

histograms of oriented gradients [1], or HOG, are a very popular image feature in computer vision. the recipe is pretty straight forward: the image is divided into (usually 8x8) cells, for each cell you compute a (usually 9 bin) gradient orientation histogram. then there’s a funky normalization step where you group cells into blocks (typically a block is 2x2 cells or 3x3 cells), and your descriptor consists of going through each block and normalizing the histogram of each cell in that block by the block’s magnitude (i.e. each cell is represented multiple times in the final descriptor; the paper contains a much better explanation).

https://www.youtube.com/watch?v=0Zib1YEE4LU

This report will discuss how I used the Fourier Transform and feature classification to recognise different alphabetical letters given a set of images. I began by viewing the test set images, the magnitude spectrum and the phase spectrum, but they didn’t reveal much information. I used a High Pass Filter to remove low frequencies by masking with a central window sized according to the input image (30% approx.). This revealed more information in the magnitude spectrum, especially for the letters T,V.

After experimenting with different convolution formulas, it appeared that this would be the most appropriate method for extracting the edges. To isolate the vertical data, I applied a convolution function with the weighting function g(x) = [[1,-1]], and g(x) = [[1],[-1]] to isolate horizontal data. I combined these two results by finding a common size between the images and summing their normalised values. This produced a very clear image only containing the edges of each letter.

Process so far:

1. Convolution on image – reconsider because change not made whilst ‘in the fourier space’
2. Fourier Transform
3. Magnitude spectrum and Phase spectrum using np.angle
4. Isolate parts of mag spec using Spectral regions
5. Calculate power value of image
6. Configure feature matrix
7. Compare using cw1