

**“Your Handwriting is Worse Than the Doctors!”**

**Cursive Handwriting Segmentation  
and Character Recognition**

**Fall 2007**

Group 3

Jon Taylor, [jmtaylor@andrew.cmu.edu](mailto:jmtaylor@andrew.cmu.edu)

Siddarth Kumar, [siddarth@andrew.cmu.edu](mailto:siddarth@andrew.cmu.edu)

Irina Khaimovich, [ikhaimov@andrew.cmu.edu](mailto:ikhaimov@andrew.cmu.edu)

## Table of Contents

1. Introduction – The Project .....	3
2. Our solution .....	3
3. Prior Work .....	4
4. Quick Introduction to ICR .....	5
5. Data Flow .....	6
6. Algorithms .....	7
6.1. Preprocessing .....	7
6.2. Handwriting Normalization.....	8
6.3. Thinning.....	9
6.4. Word Segmentation .....	11
6.5. Character Segmentation .....	11
6.6. SVM.....	13
6.6.1. RBF (Radial Basis Function) Kernel .....	15
6.6.2. Parameters .....	15
6.7. Features.....	16
6.7.1. Zoning Density Algorithm .....	16
6.7.2. Width/Height Ratio .....	16
7. Training, Testing and Validation Sets .....	17
8. Results .....	18
9. Available Software & Our Work .....	24
10. Optimization, Paging and Parallelism .....	25
11. The Demo .....	26
12. Division of Labor .....	27
13. Schedule .....	28
14. Conclusion - Future Improvements .....	29
15. References .....	30

## **1. Introduction – The Project**

The problem occurs while studying and reading the professor's notes that were uploaded onto the internet or looking over friend's notes that have been lent to you, and you find yourself taking more time deciphering the handwriting than learning what is on the page.

Automatically deciphering handwriting has huge applications, which includes currently implemented ICR technology that the Postal Service uses to help sort mail. ICR technology applies to a wide scope of any data that is handwritten (i.e. forms) that needs to be processed. Imagine if all forms could be run through a system, and automatically be entered into a database. Automatic processing would drastically increase time efficiency and cut costs.

## **2. Our Solution**

The solution to unreadable notes is putting them through a scanner, and converting the messy cursive into machine print (ASCII). Our project will deal with the cursive segmentation and recognition of these notes. This project will take a scanned image of words and segment the cursive letters. To obtain correct segmentation of cursive letters:

- 1) Document will be scanned in
- 2) Scanned document will be turned into a binary image for further processing
- 3) Normalization of handwriting to take out slant
- 4) Thinning algorithm is applied to facilitate segmentation
- 5) Segmentation algorithm to split each character for recognition
- 6) Support Vector Machine (SVM) implementation for recognition

### **3. Prior Work**

In the past there have been three projects in 18-551 that have attempted handwriting recognition. In spring 2002, Group 8 implemented an offline hand printed text recognition system using Fourier Descriptors. The second project was done in spring 2003 by Group 5, which furthered the previous year's project by improving the segmentation algorithm, detected capital letters, and including context recognition. The third project was done in spring 2005 by Group 9 which consisted of a PDA that was the input to an online handwriting recognition system.

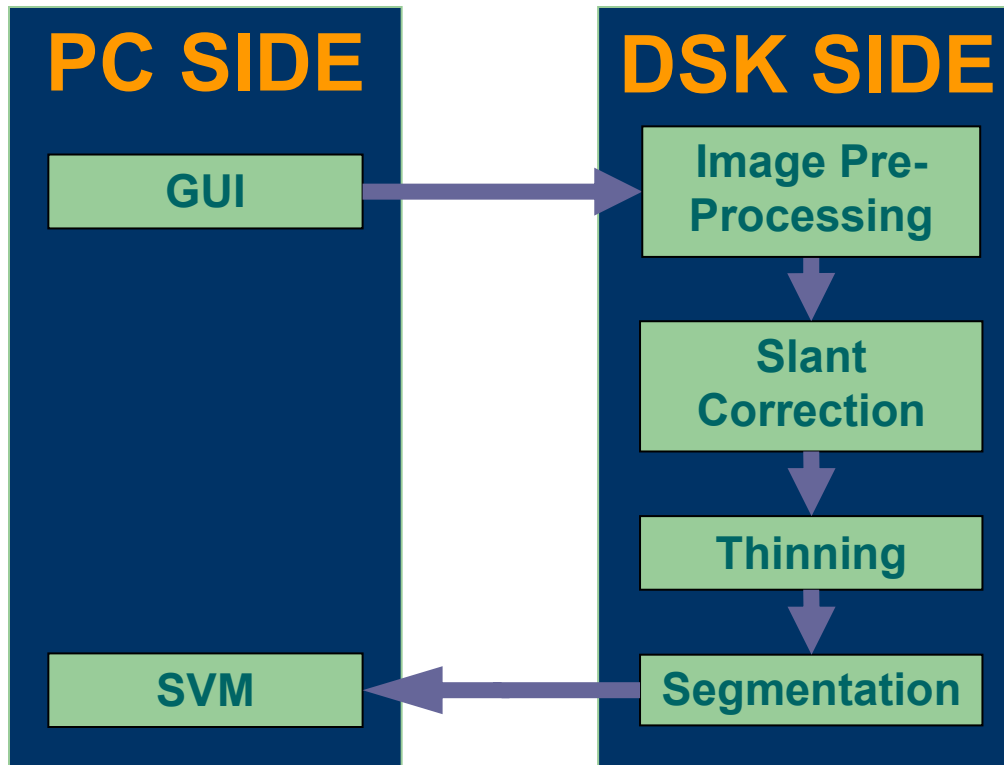
Thus far there has been no group to attempt cursive recognition. Our project differs vastly from previous projects because we implement general projections for slant correction, implement a thinning algorithm in order to trace the words, our segmentation algorithm is much more complicated because the characters within a word are all connected in different ways, and we implement an SVM for recognition.

## **4. Quick Introduction to ICR**

Handwriting recognition is an ongoing task to solve. Optical character recognition, OCR, is the attempt to turn handwritten print into ASCII, whereas intelligent character recognition, ICR, is the attempt to turn different fonts and styles, like cursive, into ASCII. ICR is a considerably harder task to accomplish due to variability and segmentation issues. Even though ICR has been around for some time, it has not been universally implemented yet.

There are commercial products that implement ICR solutions. ABBYY has a product called ABBYY FormReader 6.5 Desktop Edition which can take structured or semi-structured forms and capture key data that needs to be processed.

## 5. Data Flow



Our system starts with the GUI where the user chooses an image of cursive text that should be converted to ASCII text. The user puts in a threshold and then hits run. The GUI starts executing the DSK code, which includes the slant correction, thinning, and segmentation. The processed images are shown to the user in the GUI and then the user hits run recognition to run the SVM. The results produced by the SVM are shown in the GUI.

## 6. Algorithms

We will talk about the different algorithms that we used in each section as shown by the Data Flow chart.

### 6.1 Preprocessing

The first thing that is required to be done is to read in a bitmap file and convert it to an array. This operation is performed on the PC side, and then the array is sent to the DSK to be processed further. A 'bmp' file is simply an array of values, with an extra file and info header in order for the image to be read correctly. We simply read in these pieces of information, and then send the size of the image and the array to the DSK. We assumed all images being read in were already in the form of 256 grayscale. In order to properly process these images further, we need to convert them to binary images. A binary image is one in which there are only two colors, normally with a value of zero corresponding to a black pixel and a value of one corresponding to a white pixel.

The most common way to convert from a gray image to a binary image is to use "thresholding". Thresholding is when you set a value to decide whether or not a pixel will be black or white. A common threshold for text images is 200. This means, in a 256 grayscale image, that all pixel values below 200 become black, and all values above 200 become white. However, the intensity of a writing sample can vary greatly between authors, thus it is imperative that we are able to correct for this.

In order to achieve maximum results in the other sections, we desire an image of the writing sample that has both very little background noise (random black pixels in the background that are not part of the text) and has almost no "holes" in the image. A hole can be defined as either a white pixel in the middle of a character (where there should normally be a black pixel) or any gap between characters that is suppose to be connected. Since we found it very difficult to implement an algorithm that could properly threshold the image every time, we chose to have the user input the threshold to use for each image. This is much like if a user were scanning an image and they saw the result came out far to light, the scanner would allow them options to darken the image. This is exactly what we are doing. If the user sees that the thresholded image is to light or dark, they can adjust the threshold accordingly to achieve maximum results. If the user does not input a threshold however, it will automatically default to a value of 200.

After we threshold the image, we will also need to "zero pad" the image. This step simply involves adding a border of white pixels around the entire image. Although this step is not necessary, it is very important for improving efficiency in later steps. In other steps later on, we will often be checking the neighboring pixels for an individual pixel. Thus, if we find a black pixel on the border of an image, we will need to check each time to make sure that we don't try and access a pixel outside the image (and thus our program would crash). Thus if we add this border of white pixels, and we only perform

processing to the pixels from our original image, this will insure that when we are looking at an edge pixel, we will never accidentally call an out of bounds pixel, we will now be accessing the newly added border pixel instead.

## 6.2 Handwriting Normalization

Handwriting fluctuates greatly between people. Although humans can correct for this variability, it is a very complex task to have a computer understand off-line written text. Not only do people vary in their style for writing letters, but they also vary in geometric features. Hence, it is a typical step to correct for such geometric factors by normalizing handwriting before segmentation. Specifically, eliminating and/or taking into account slant angle and skew angle using generalized projections is an important pre-processing step. Figure 1 is an example of the definition of skew and slant.

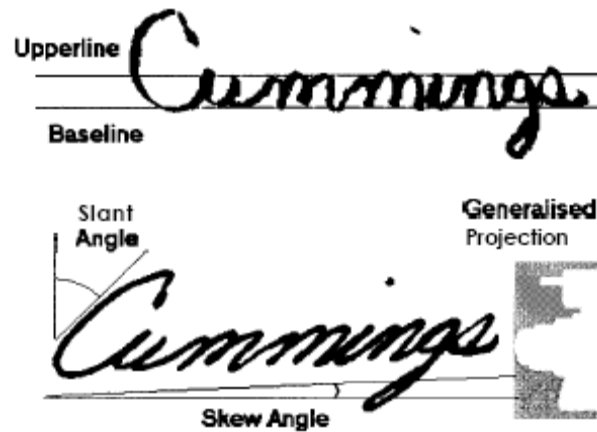


Figure 1 [1]

General projections (GP) are an extension of projection profiles (PP). To deal with these projections our image needs to be represented in binary black and white. Black will represent text. The PP will count up the black pixels located in every row. The rows with the greatest amount of black pixels represent highest likelihood of that pixel being part of text. Although, if there are many black pixels around the word (such as noise), then the PP method will not be very reliable. GP's are used to give greater weighting to black pixels that are adjacent to one another. Thus the more adjacent pixels, the more likely it is part of a word, and the more weighting it is given. Figure 2 shows the difference between PP weight and GP weighting.



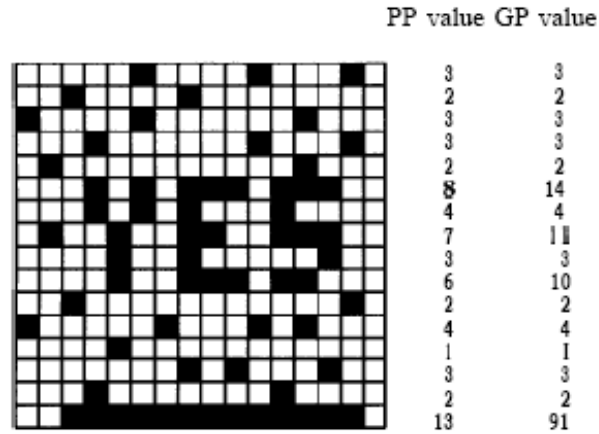


Figure 2 [1]

Slant estimation provides a slight rotation of the word in order to make the letters vertical. Many algorithms try to detect long quasi-vertical strokes and use their features in order to estimate the slant angle. GP uses all the image data in order to avoid the pitfall of quasi-vertical strokes if only a few are present. Thus if a word doesn't have many l's f's or such other long characters that have long strokes, GP's will be able to define the slant angle.

In order to find the slant angle we calculate the GP's along angles from  $-15^\circ$  to  $45^\circ$  using eight equally spaced projections. The angle with the greatest GP is considered to be the region where the greatest slant angle will be present. Thus our new outside angles will become the GP's that were one less and one greater than the maximum GP we obtained. We iteratively do this until we are testing GP's within one degree of each other. Thus, instead of testing 61 angles, we are testing 23 angles to decrease the processing time.

After the slant angle is obtained we do a transformation of the image in order to fix the slant of the image. This is done by "shifting" the image. We start by moving along the bottom row of the image. For each pixel, we will look at the line on the image that corresponds to the slant angle that we found. For example, if we had a square image, and the slant angle was  $45^\circ$ , then our first line would run from the bottom left pixel to the top right pixel in the image. Each pixel on these lines should be then shifted over, such that they are now in a straight line. If you picture the square image again, the  $45^\circ$  line should now run straight up and down, and should be the left border. Thus we are simply shifting each pixel in the image to correct for the slant angle that we detected.

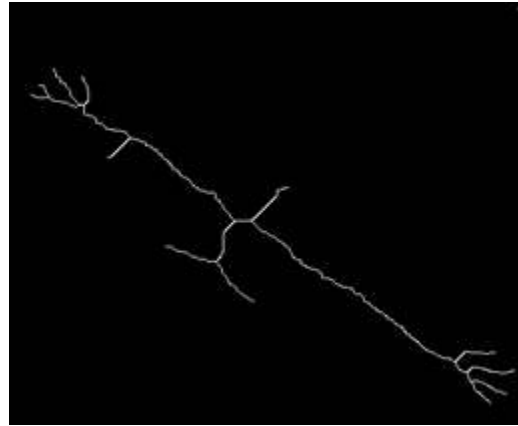
### 6.3 Thinning

In order to properly implement our segmentation algorithm, it is a requirement that we be able to trace through our image. In order to accomplish this, we need every black line in our writing image to be only 1 pixel in width. We will implement a thinning algorithm to make this happen.

There are several different types of thinning algorithms that can be used. The largest division of thinning algorithms comes between iterative and non-iterative algorithms. Iterative algorithms will keep performing on the image until it can no longer find pixels to change. Conversely, a non-iterative algorithm will perform all the operations once on the image. Since we require the line width of characters to be exactly one pixel wide, we will use an iterative algorithm, since this will give us a thinner image.

Since we decided to use an iterative thinning algorithm, we must now decide whether to use a parallel or sequential algorithm. A sequential algorithm will pre-compute the border pixels, and then apply a set of rules to each. A parallel algorithm will look at one pixel at a time, and the decision for deletion of a pixel will be based on the results of the previous iteration. Although sequential algorithms are often faster (since each pixel does not always need to be examined), we decided to use a parallel algorithm because of the ease of implementation and the more effective thinning of an image.

Each time we process a pixel, we follow a set of rules to determine if a pixel should be deleted. We start by performing two different iterations. Our goal throughout these operations is to determine if the pixel is an edge pixel, and thus could be effectively thinned. For each pixel in the first iteration, we will check the surrounding 8 pixels (the 3x3 grid, with the pixel being the middle) and store their values. If there are between 2 and 6 black pixels around this pixel, we will want to consider this pixel for deletion. Second, we will want to determine if there are only two sides to the edge (a white side and a black side). We determine this by seeing how many times in our 3x3 grid that neighboring pixels change color. Thus if we move around the outside of our 3x3 grid in a single direction (clockwise or counter-clockwise), there should only be one change going from a black pixel to a white pixel. This must be true to still be considered for deletion. Thirdly, we will want to make sure that our pixel is not part of a flat edge. For example, we do not want there to be a black pixel with the only 3 white pixels to be on one side. Although these pixels may eventually be deleted, we will first want to delete the corners of this line, and work our way in. This section is where our 2 iterations differ. In our first iteration, we check the left and top of the neighboring pixels to see if there are 3 white pixels. The second iteration will check for this border case on the bottom and right sides. This makes sure we are not thinning twice in a row, and thus over thinning, and actually losing part of the image. We will repeat the first iteration until there is no longer a pixel we can change. We do the same thing for the second iteration, performing pixel deletions until there are no longer any pixels to delete. Below is an example of what a thinned picture looks like. Images of a thinned cursive writing sample will appear in the results section.



#### 6.4 Word Segmentation

Before we are able to segment the characters in a word, we must first separate a line of writing into individual words. This task is fairly easy, as we assume there should be a white space between words. Our algorithm searches for this line of white pixels that is separating different words. In order to ignore any noise, our algorithm tests to make sure any black pixel is connected to at least 7 other black pixels. This indicates to us that the pixel is indeed part of a word, and not just noise. In order to increase accuracy, this number could always be increased; however, there was generally not much noise and limiting it to 7 pixels increased efficiency. Once we found the location of the first and last pixel of a word (viewing from left to right), we are now ready to use these values for the character segmentation.

#### 6.5 Character Segmentation

The segmentation part of this project is the true meat and bones. In order to have any chance at character recognition, you will need to segment the characters in a word properly. With printed text, you will always have a gap between characters. However, with cursive text, each word is one long string of black pixels; thus the problem of segmentation becomes exponentially harder. In researching algorithms, we found that many were either too simple or too complex. Many of the algorithms compensated for their short comings at segmentation by using a sort of lexicography. They would often use a weighted system to decide, as they built a word, what the next character should be. For example, if there algorithm recognized the first letter in a word to be a 'c', and the next character was decided to either be an 'h' or an 'n', there "dictionary" would tell them that it was much more likely that the letter was an 'h' rather than an 'n'. Although this would greatly improve accuracy of our segmentation and recognition, doing something such as this would have required too much time, and thus we had to bound our project. With that said, we required an algorithm that would effectively segment without any sort of dictionary help.

Throughout our research, we did come across one major principle of segmentation that appeared to be crucial for properly segmenting cursive text. This principle is the idea of “over-segmentation”. In order to properly segment all characters, it is ideal that for some characters you actually segment more times than actual characters. This becomes very apparent if you think about the letter ‘w’. The character ‘w’ is virtually the same as a ‘u’ and an ‘i’ put together. Thus, it would not be desirable to segment the entire letter ‘w’, but rather segment it such that it would appear there are 3 “i’s” in a row. Then during our recognition, we can try to reform the letter ‘w’ by putting some of the segmentations back together. It is very easy to put words back together by adding two segmentations together, but it would not be possible to split two characters apart without the proper segmentation. Thus, it is imperative that we use over-segmentation.

Since ICR of cursive writing is still a topic of heavy research, we decided to develop our own segmentation algorithm. Our algorithm is based on tracing both the upper and lower contours of the image to find when connectors between characters are used. Usually, connectors between characters are the only lines connecting two characters. This means that if you tried tracing the upper and lower contours, these connectors should be traced by both the upper and lower tracing algorithm. Although we could simply just try vertical line segmentation, our algorithm will be much more effective in figuring out segmentation points, as there will be more rules we can implement with tracing to determine the optimal point for segmentation.

First, we will trace the upper contour of the word. To do this, we follow a simple set of rules. We follow the direction set as shown below.

1	2	3
8	X	4
7	6	5

We will try and move to direction 1 first. If that pixel is white, then we will try and move to direction 2 and so forth. In order to ensure full tracing though, if two consecutive pixels are black, then we will first move in the cardinal direction. For example, if 3 and 4 are both black, we will move to 4 first, and then we will move to 3 the following move. This will ensure we visit every pixel in the trace. We will also keep track of every pixel visited to ensure we never revisit a pixel twice to avoid an infinite loop. At each pixel, we also check to see if we are very at an intersection point (a pixel such that we can choose between 2 directions to travel, not including pixel we just moved from). If we ever get to a dead-end (i.e. the top of a ‘t’), then we simply revert back to the last intersection point, and try a different direction from there. We will keep performing these operations until we get to the end of the word pixel.

Now that we have the upper contour traced (and saved in a separate array), we will want to trace the lower contour. In order to increase efficiency, we will perform the segmentation of characters as we trace the word. We start by tracing the lower contour of the word. We will trace the lower contour exactly as we did with the upper, except we will use a new hierarchy for the direction to travel, as shown below.

7	6	5
8	X	4
1	2	3

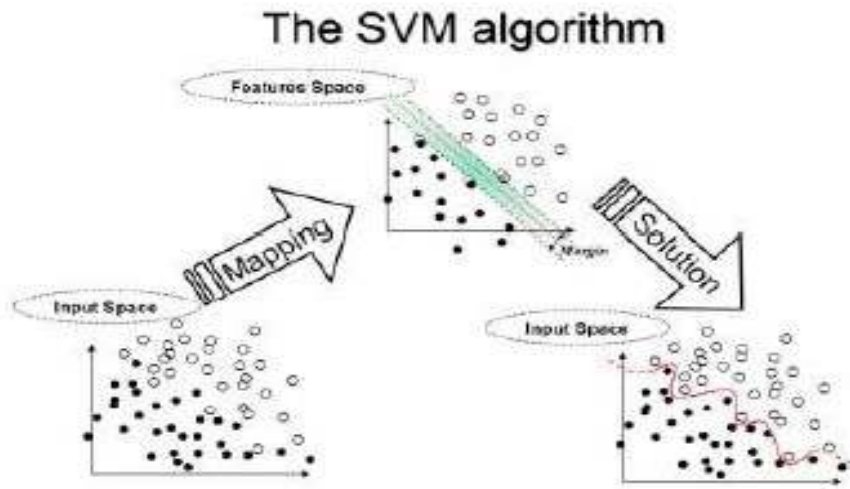
The first thing we are checking for is when the upper and lower contours trace is on the same line for at least 3 pixels without splitting apart. If this occurs, we mark that point as a segmentation point, with a few exceptions.

1. If the point is within 2 pixels of another segmentation point, we ignore it and move on
2. If the point is within 5 pixels of the start or end of a word
3. The pixel must be moving in an upward direction, or have just moved in an upward direction. All connectors between words should at some point move in an upward direction. This prevents us from segmenting too early in a character.
4. If the upper and lower contours travel along the same line for many pixels (more than 12 for our algorithm). This will often occur in a letter such as an 'r'. To deal with this, if the pixel count of travelling together is over 12, we will segment at the next intersection point (When the upper and lower contours split). Note that we look for intersection points, and not when the lower contour is without the upper contour. For such letters as 'i', the upper contour does trace elsewhere, but the lower contour trace is still always part of the upper trace. This is why we look for intersection points instead.

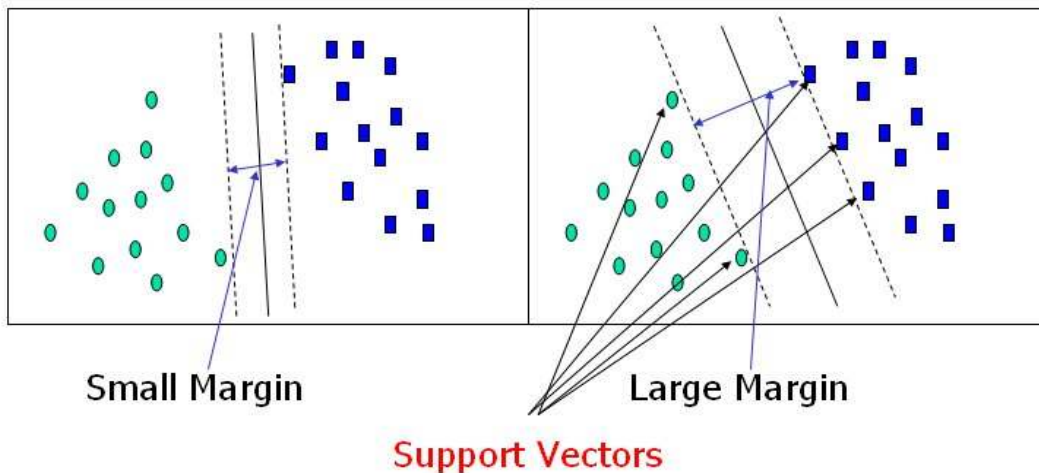
If none of these exceptions occur, we mark that point as a segmentation point.

### **6.6 SVM – Support Vector Machine (Character Recognition)**

We decided to use Support Vector Machines to recognize our segmented characters due to the fact it gives in the best results in pattern recognition and is easy to implement. SVM is a set of methods used in machine learning for classification and regression from a set of learning data. Each of these samples in the learning data is mapped to a value in a higher dimensional space. The data is classified by coming up with an optimum N-dimensional hyper plane which separates data into positive and negative examples.



The plane will be chosen to have the largest distance from the hyper plane to the nearest of the positive and negative examples. This is known as finding the maximum margin and the greater the margin, higher are the rates of classification. The vectors that define the width of the margin are known as support vectors. Here is the representation for a simplified two-class problem where the squares are the positive samples and the ovals are negative



For the purposes of our project, we used a multi-class SVM since our problem was to classify into one of 52 classes (26 lowercase & 26 uppercase) instead of just classifying into positive and negative samples.

### 6.6.1 RBF (Radial Basis Function) Kernel<sup>1</sup>

Using a kernel function, SVM's are an alternative training method for polynomial, radial basis function and multi-layer classifiers in which the weights of the network are found by solving a quadratic programming problem with linear constraints. In this case, since there is a non-linear relationship between the classifiers and number of features is not too large, we opted to use the RBF or radial basis function kernel which is of the form.

$$K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2), \gamma > 0.$$

, where gamma is an input and  $\mathbf{x}_i$  is the training vector and K is the kernel function

We used the libsvm package<sup>2</sup> to implement the SVM part of our project. A classification task usually involves training and testing data which consists of data instances. Each instance in the training set contains one "target value" (class labels) and several "attributes" (features). The goal of SVM is to produce a model which predicts target value of data instances in the testing set which are given only the attributes.

The training data file is of the format:

```
1 1:0.5 2:0.5 3:0.5.....10:0.5
.....
.....
.....
N 1:0.5 2:0.5 3:0.5.....10:0.5
```

where the class (1 to N) is the target class and the attributes or features are represented in increasing order with its value.

The input file is the set of features we extract from our segmented characters with 0 as the target class indicating unknown and in the same format as our training data mentioned above. The working of the SVM is in two stages. There is a one-time training process which outputs a file (model) which is the learning of the SVM based on the training data. In the second stage, the SVM predicts a target class applying this model on the testing data.

### 6.6.2 Parameters<sup>2</sup>

The libsvm software that we used accepts two parameters for the RBF kernel: C and gamma.

C refers to the penalty parameter which imposes a cost on misclassification and this plays a role in the size of the margin discussed earlier. It has been observed that when C is greater, gamma should be lower and vice versa so an optimal trade-off is required. We

---

<sup>1</sup> [www.dtrek.com/svm.htm](http://www.dtrek.com/svm.htm) Support Vector Machines

<sup>2</sup> <http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf> A guide to using the libsvm

utilized a Python script grid.py included in the libsvm package to give us the optimal values<sup>2</sup>. Hence, we used  $C=128$  and  $g=2$  for our RBF Kernel. Gamma is used in the computation of the kernel

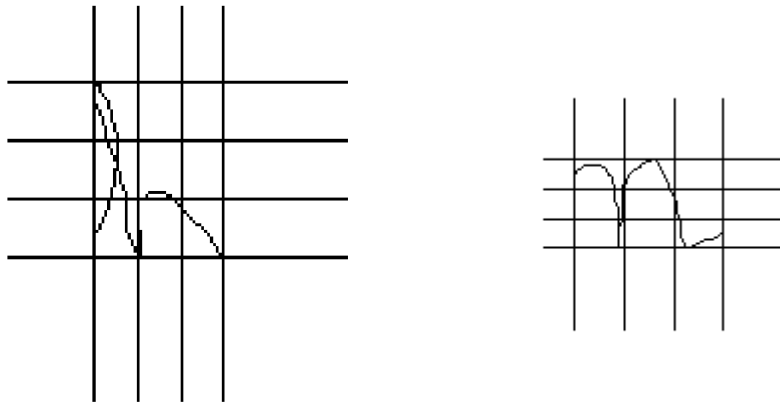
$$K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma \|\mathbf{x}_i - \mathbf{x}_j\|^2), \gamma > 0.$$

## 6.7 Features

To uniquely identify each character, it is necessary to identify and extract certain unique features from the character. This is our input into our recognition algorithm or the SVM discussed later. We used two primary methods: zoning density and width/height ratio to give us 10 total features (9 from the zoning density and the width/height ratio)

### 6.7.1 Zoning Density Algorithm

The character is divided into 9 equal boxes (3 x 3 matrix) like shown in the figure below. The number of black pixels in each of these cells in proportion to the total number of black pixels in the characters is used as a feature. So, the percentage of black pixels in each cell contributed to 9 features.



Using this characteristic, it is possible to distinguish an 'h' from an 'n' etc.<sup>3</sup>

### 6.7.2 Width/Height Ratio

The second feature we used was width to height ratio. Some letters are defined by their length while others are characteristically wide.

For example, an 'f' has a width to height ratio of 0.4957 while an 'm' has a width to height ratio of 2.3529.

<sup>2</sup> <http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf> A guide to using the libsvm



## 7. Training, Testing and Validation Sets

To train our SVM's, we got 11 different samples of each character. 1 set out of these 11 different samples was used for validation and 1 set not present in these 11 different samples in the training set was used for testing. They were in 256 color .bmp format. Some of the sample characters we used are:



There wasn't any noise or distortion in our samples that we had to account for.

## 8. Results

The results are broken up into 4 sections: Slant Correction, Thinning, Segmentation, and SVM. We will also show 3 examples of images we tested, showing the thresholded image, the slant corrected image, the thinned image, and then the final segmented image.

### Slant Correction

Although we have no numerical way of determining how our normalization worked, we can see from all of our test images that it seemed to work with extreme accuracy. We were able to account for any angle, positive or negative, and correct for that angle. The normalization we used was pretty straight forward and worked as expected.

### Thinning

Once again we have no numerical way to determine how our thinning worked. However, we do feel fairly comfortable in the implementation and results from this algorithm. The algorithm did do exactly what we wanted, although in certain cases it did seem to slightly over thin the image, which resulted in missed segmentation points.

### Segmentation

Overall, our group was extremely happy with the results from the segmentation algorithm. Due to the fact that we developed the algorithm ourselves, the accuracy rate of 86.5% was right near our goal. Although our final project goal was dependent upon how well the SVM worked, we were very pleased with the accuracy of segmentation, especially on very neatly written samples. Below are 3 different samples, all with varying results

The first example is the best sample handwriting sample we had. We were able to properly detect every segmentation point that we wanted. Although there are a few extra segmentation points, our character recognition should be able to weed those out. For example, if we slice a 'g' in half, then our SVM should be able to see that the combined segmentation of 'g' is much closer to a character than either half of a 'g'.

The quick brown fox jumps over the sleazy dog

The quick brown fox jumps over the sleazy dog

The quick brown fox jumps over the sleazy dog

The quick brown fox jumps over the sleazy dog

The second image was an average image, having a few missed points, but on the whole still segmented fairly well.

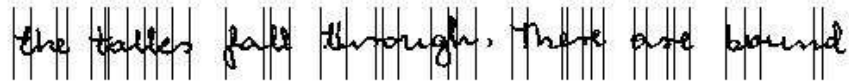
the taller fall through, there are bound

the taller fall through, there are bound

Cursive Handwriting Segmentation and Character Recognition  
18-551 Group 3 of Fall 2007

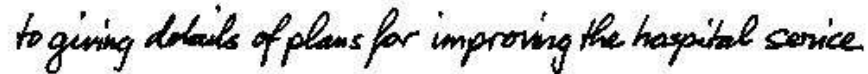


the taller fall through. There are sound

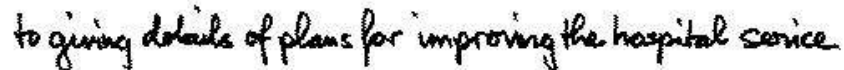


the taller fall through. There are sound

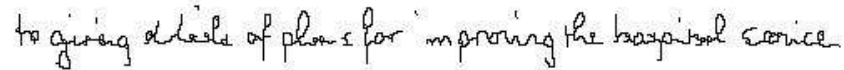
The third image was one of the worst ones. For various reasons, the image failed to detect about 30% of the desired segmentation points.



to giving details of plans for improving the hospital service



to giving details of plans for improving the hospital service



to giving details of plans for improving the hospital service



to giving details of plans for improving the hospital service

Cursive Handwriting Segmentation and Character Recognition  
18-551 Group 3 of Fall 2007

Below is a chart which summarizes our results from 19 images, which gave a total of 153 words. In total, we were able to detect 603 points out of a desired 697 points. This results in an **86.51%** accuracy for segmenting images.

<u>Words</u>	<u>Total Correct Segmentation Points Found</u>	<u>Total Segmentation Points Needed (Total Characters)</u>
9	30	32
7	30	33
9	25	28
10	42	47
7	33	36
7	31	34
6	24	29
8	30	33
9	37	37
8	32	33
5	17	21
9	33	40
8	30	36
8	41	49
10	35	50
8	33	37
7	33	43
8	34	40
10	33	39

After analyzing all 19 of our test images, we found two distinct reasons for why the segmentation failed. These two reasons account for a large majority of the missed points as well.

1. We failed to segment the letter 's' properly. Normally an 's' is suppose to look like the figure on the left, however, many of people's handwritten 's' actually look like the figure on the right. Our algorithm is waiting for that intersection point as circled below, before it thinks there is a segmentation point. However, with many of the 's' in our samples, the 's' looks like the one on the right, which doesn't have this intersection point. Thus our algorithm does not think there is suppose to be a segmentation point, and fails in this instance.



2. The second area that our algorithm often fails is with 'i' and 'u'. Normally our algorithm would see an intersection point as shown below on the left, and will look to segment as soon as the trace moves in an upward direction again. This would be right smack center through the 'u' (which is a good thing; we want to segment the 'u' in half). However, due to faults with both our thresholding and our thinning, we show below on the right what the thinned 'i' actually looks like in some examples. We can see there is no intersection point, and thus our algorithm thinks that we are still tracing the same character, and it doesn't know to segment as soon as the trace moves upward again.



### SVM

We managed to get 95.8333% accuracy with our validation using a RBF kernel using parameters  $C=128$  and  $g=2$ .

```
C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\siddarth\Desktop\libsvm-2.85\windows>svmpredict ourtes
t1.dat ourtraining3.dat.model test1.pred
Accuracy = 95.8333% (23/24) (classification)
C:\Documents and Settings\siddarth\Desktop\libsvm-2.85\windows>
```

But, our testing on untrained data did not give us anything above 60 %. This was mainly because we could not get enough distinguishable features for the SVM to make a good classification. Furthermore, our accuracy could have been increased if the number of samples were greater. In the context of the whole project, we could not integrate the SVM fully into our cursive handwriting segmenter and recognizer because there was no feedback mechanism from the SVM. Therefore, since we were using over-segmentation, we could not guarantee that our DSK was sending whole characters to the SVM and this led to the SVM sending back random outputs. We would have been successful either if we had found some way to combine over-segmented bits to make a character (hard) or got some feedback from the SVM saying that it does not think that the input is a character and then we could have recombined it based on that feedback. Overall, our SVM was unable to output the written sample within any degree of accuracy.

## 9. Available Software

For our project we wrote most of the code. We used available open source code for the thinning and SVM algorithms.

### Online Source Code

1. Thinning Algorithm

This algorithm was found in a paper of survey's of thinning algorithms. The code was written in Matlab. After testing it in Matlab, it was converted it into C and placed it onto the DSK.

2. SVM Code

This code is in an executable format written in C.

### Our Code

1. Preprocessing

Read in bitmap image in C and then thresholded the image to convert it to a binary image. Implemented on the DSK.

2. Handwriting Normalization

Slant detection and correction algorithm was written in Matlab and tested. Afterwards converted into C and implemented on the DSK.

3. Segmentation Algorithm

Segmentation algorithm, which includes tracing upper and bottom contours in order to find segmentation points was written and tested in Matlab. Later it was converted to C to be implemented on the DSK.

4. Features for SVM

Features that include zoning were written in Matlab and then converted to C to be implemented on the DSK.

5. GUI

GUI was written in Java, and interfaced with DSK and SVM code.



## 10. Optimization, Paging and Parallelism

Images were read in on the PC side and the size and dimensions were passed to the DSK along the 2-D matrix of the values. The average size of the image of the sentence we were segmenting was 500 x 50 pixels (20-25 Kb, 1 byte per pixel, 256 color bmp). Periodic copies of this matrix are made for the sake of algorithms (thinFrame, padFrame, newFrame etc.) but were freed appropriately to avoid memory leaks. So, memory problems were avoided. Direct paging could not be done since subtle modifications had to be made to the new matrix.

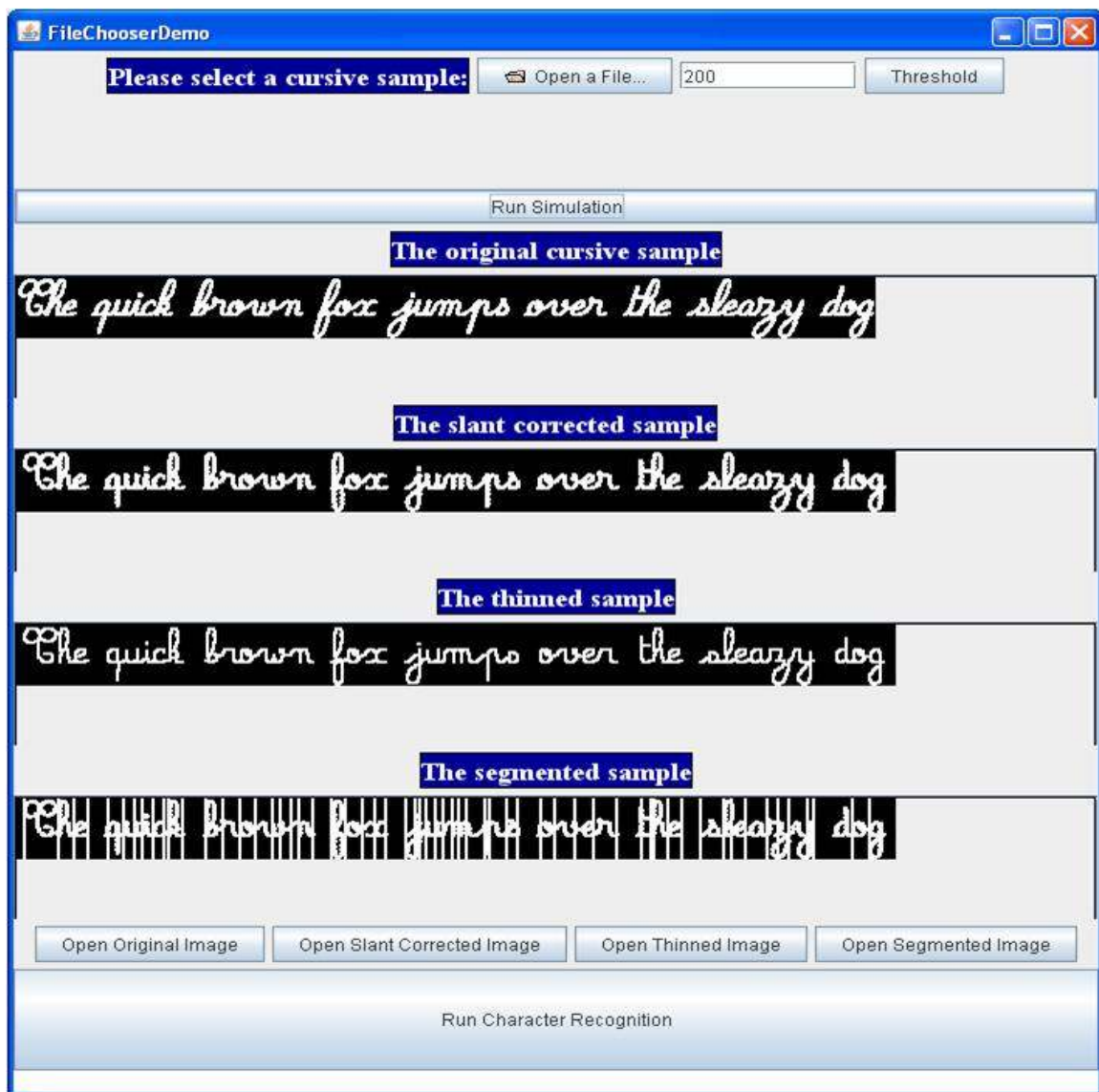
We optimized by using the third level of optimization (-o3) on Code Composer. The selected functions on the DSK and their associated timings on using Function Profiling:

preproc()	88,696 cycles or 0.00354784 secs
zeropad()	231,449 cycles or 0.00925796 secs
threshold()	64,263 cycles or 0.00257052 secs
segmentLines()	1,153,689 cycles or 0.04614756 secs
thin()	8,619,633 cycles or 0.34478532 secs
traceLower()	233,735 cycles or 0.00934940 secs
traceUpper()	222,985 cycles or 0.00891940 secs

It should be noted that all these functions vary wildly from image to image especially the segmentation algorithm (traceLower) because the outer loop is controlled by the number of words. As such it will vary with the length of the word (number of black pixels). traceLower and traceUpper showed good optimizations with up to 7 iterations done in parallel while the other functions showed up to 2 iterations done in parallel. The most time-intensive part was actually writing the intermediate files for purposes of displaying on the GUI but the actual algorithms were executed very quickly.

## 11. The Demo

To demo our final project, we implemented a Java GUI that showed the processed image at different stages in each of the algorithms. In our demo, the user could load any bmp image file from any directory. Then, the user enters the threshold to use (default 200), and hits the threshold button. This copies the image file into the DSK working directory, and passes the threshold to be sent to the DSK. Once the user is running the DSK, they hit run simulation, and it will begin to display the images as they are processed through the DSK. Once the DSK is completely done, the user is free to run the SVM on the processed image. The output text will be displayed in the box at the bottom.



## 12. Division of Labor

TASK	RESPONSIBILIY OF
Preprocessing	Jon
Slant Correction	Irina
Thinning	Jon
Segmentation	Jon
SVM Implementation	Siddarth
SVM Features	Irina/Siddarth
Implementation on DSK	Siddarth
GUI	Irina

\*Note: Although we have split up the responsibility and leadership of each task, there was considerable overlap as we worked together on nearly every task.

## 13. Schedule

Week 1 (October 8):

- Continue to understand algorithms

Week 2 (October 15):

- Compile database
- Start Preprocessing Images

Week 3 (October 22):

- Further understand normalization algorithms
- Start implementing normalization algorithms (General Projections)

Week 4 (October 29):

- Continue/Finish implementing normalization algorithms
- Further understand segmentation algorithms
- Start implementing segmentation algorithms

Week 5 (November 5):

- Continue implementing segmentation algorithm
- Implement segmentation algorithm on DSK
- ***Prepare for oral updates (Everyone)***

Week 6 (November 12):

- Finish implementing segmentation algorithm
- Finish Thinning algorithm on DSK
- Implementation of SVM
- Start working on GUI output of system

Week 7 (November 19):

- Thanksgiving Break

Week 8 (November 26):

- Finish all loose ends (aka stay up all night, every night in lab)
- Optimize code
- ***Prepare for oral presentation***

Week 9 (December 3):

- ***Complete Final Write-Up***

## 14. Conclusion - Future Improvements

There is an enormous amount of work that could be done for future projects (or if we had another semester!).

1. Reading an entire document instead of just one line at a time
2. Dealing with both colored and grayscale images (very easy to do, just simply ran out of time)
3. Thresholding could have been done through an algorithm, not through user input
4. Noise cancelling algorithm in the preprocessing stage
5. A filling algorithm, to fix gaps between characters (currently a gap between characters will cause the word to finish segmenting early and miss a large portion of the word).
6. A thinning algorithm that would not lose intersection points. This was one of the biggest reasons for missed segmentation points.
7. If the other 4 steps are implemented, then the segmentation algorithm would improve greatly. However, it may be optimal to have a different set of rules based on different images to improve accuracy.
8. More training and more features for our SVM. Had we had more time, it would have been optimal to use around 50 training samples for each character, and around 20-30 "good" features for the SVM. We were nowhere close to either of these numbers, thus our poor results in the SVM section.

As a pioneering group with cursive recognition, we feel we have set a decent path for any groups who wish to follow. Since we had to research everything without the help of prior groups, breaking ground on individual sections took longer than planned. We were simply unable to successfully finish implementing the SVM, although, the rest of the project worked as we had hoped. Future groups should hopefully be able to replicate our project with a working character recognition algorithm. Overall, we were pleased with our project; however, we were slightly disappointed we were unable to successfully implement our SVM algorithm with a high rate of character recognition.

## 15. References

### **Paper describing algorithm for slant correction using Profile Projections**

[1] Nicchiotti G. and Scagliola C., Generalised Projections: a Tool for Cursive Handwriting Normalisation. *Proc. 5th ICDAR* Bangalore 1999, pp 729-733.

### **Paper describing segmentation algorithm using holes and minima points**

[2] Nicchiotti G. and Scagliola C., Simple and Effective Cursive Word Segmentation Method, 7 th IWFHR, Amsterdam, September 11-13, 2000

### **In depth paper describing segmentation using chain codes**

[3] Bozinovic R.M. and Shrihari S.N., *Off-line cursive script recognition. IEEE Trans. PAMI* 11 (1) 1989 pp. 68-83.

### **Source code and Paper on Thinning Algorithm that was used.**

[4] [A Survey of Image Thinning Algorithms](http://www.cs.ucf.edu/~hastings/index.php?content=papers): Website:  
<http://www.cs.ucf.edu/~hastings/index.php?content=papers>

### **Description of SVM for handwriting recognition**

[5] Abdul Rahim Ahmad; Khalia, M.; Viard-Gaudin, C.; Poisson, E., "Online handwriting recognition using support vector machine," *TENCON 2004. 2004 IEEE Region 10 Conference* , vol.A, no., pp. 311-314 Vol. 1, 21-24 Nov. 2004

### **Cursive Sample Writing Database**

[6] <http://www.iam.unibe.ch/~zimmerma/iamdb/iamdb.html>

### **Database of handwritten individual characters:**

[7] <http://mlearn.ics.uci.edu/databases/uji-penchars/>

### **Source code for SVM**

[8] <http://www.csie.ntu.edu.tw/~cjlin/libsvm/>