COMP 400 Project Report

Customized OCR solution for handwritten materials

Yuanhang Yang

Supervisor: Prof. Joseph Vybihal

December 2015

1. Introduction

Electronic documents are playing an increasingly important role when passing on information. At the same time, traditional handwritten documents still serve as an irreplaceable part in information transferring. This leads to a seemingly trivial yet frequent challenge for almost every computer user: converting handwritten documents to electronic ones.

Of course, keyboards are always there; they are the most traditional "converters" we can go to. Optical character recognition (OCR) programs have also been developed over the past years. However, if you don't have the handwriting of a printer, it can be difficult to for OCR tools to recognize what you write, even if they appear perfectly readable to humans. There are software available online that convert handwritten materials to electronic text documents, but they are not designed to recognize everyone's handwriting.

This project provides a solution to offline handwriting recognition problem with introductory level machine learning techniques. It comes with certain limitations, such as the scope of characters supported (digit recognition), and running time (with full training set the algorithm needs about 1.5 – 1.8 seconds to process each digit); however, the underlying programming framework provides an easily extensible solution with high accuracy, and the development of this solution served its purpose as a learning experience for the student.

The web application developed for this project is named ScribeX, and is available on http://scribex.azurewebsites.net/. A scribe is a person who reads some handwritten material and copies them. ScribeX is a computer program that does so. Source code of ScribeX is available on the student's GitHub page: https://github.com/aweeesome/scribex.

2. Related works

In the field of machine learning, handwriting recognition is a widely studied problem, with solutions varying in architecture and complexity. The foundation of solutions to this problem has been comprehensively studied in an article from 2000[i], where IEEE fellows Plamondon and Srihari discussed the basic steps to solving this problem: image pre-processing, image segmentation, character and word recognition, and applications of these techniques. This article also divided handwriting recognition problem into two sub-problems: online (real-time recognition where algorithm has information on the sequence and direction of pen strokes) and offline (where algorithm is only presented images of pre-written documents).

Online handwriting recognition benefits from the real-time pen movement information: for example, a written character can be easily segmented into strokes with the information on when and where the pen touches and leaves the writing pad, even when these strokes overlap. Another advantage of online handwriting recognition is the algorithm's ability to take real-time feedback from users, hence correct its output based on such information[ii]. Solutions to online handwriting recognition has been commercially available in many touch-screen devices, and they have shown high accuracy when the users aren't prescription-writing doctors.

Offline handwriting, on the other hand, is much less commercially available due to its difficulty, and the fact that it has less use cases. Still, many works have been done on this topic, with a large amount of them built on one of the two strategies: point matching and artificial neural networks. The former is implemented in this project.

A point matching algorithm attempts to match pixel points in input characters and training set characters, and defines the ones with most points matched as the best matches. It is by nature a computation-heavy algorithm because it iterates through every pixel point in all the input and training set characters[iii]. A neural network solution, however, is often less costly while providing the same level of accuracy, but it requires carefully designed neural networks with their number of input neurons, output neurons and hidden layers meticulously chosen. A genetic algorithm by Mathur, Aggarwal, Joshi and Ahlawat in 2008 provides a design on neural networks with high accuracy[iv].

3. Solution design

The solution developed for this project consists of three major parts: image pre-processing, character recognition and result presentation. Together they serve as a high-accuracy writer-dependent[1] solution for handwritten digit recognition. The outcome of this solution is presented in the form of an ASP.NET web application named ScribeX, with a simple user interface.

3.1 User flow

Upon opening the web app in a browser, the user is able to choose an image file from local device, and optionally specify the training set size to apply in digit recognition. By default, the training set size is set to 20, which means for each digit in out alphabet ({0,1,2,3,4,5,6,7,8,9}), there are 20 corresponding training images.

The user then waits for the result, formatted into words and lines as specified in original input image, to show up on the result page.

[1] An algorithm is writer-dependent if its scope changes for each user. In this project, the solution is writer-dependent because the training set solely comes from the student's handwriting.

3.2 Architecture design

In this solution, it takes four stages to complete a user request: user input, image pre-processing, digit recognition, and result presentation.

1. **User input**: User uses web UI to specify training set size and upload an image.
2. **Image pre-processing**: The image uploaded needs to be processed into monochrome, 1Bpp[2] bitmap images to allow the recognition algorithm to work on it. In this stage, images go through thresholding, noise removal, line segmentation, word segmentation and character segmentation. The output of this stage is a list of images, each representing one character to be recognized, while preserving formatting information. This algorithm will be discussed in more details later.
3. **Single character recognition**: In this step, the algorithm loops through each character image, and recognize them individually. Detailed discussion on this algorithm will be in a separate section.
4. **Formatting and presenting result**: With the help of formatting information from step 2, the program formats recognized digits into separate words and lines, and present them on the web application's result page.

The architecture design of this solution is illustrated in the following graph:

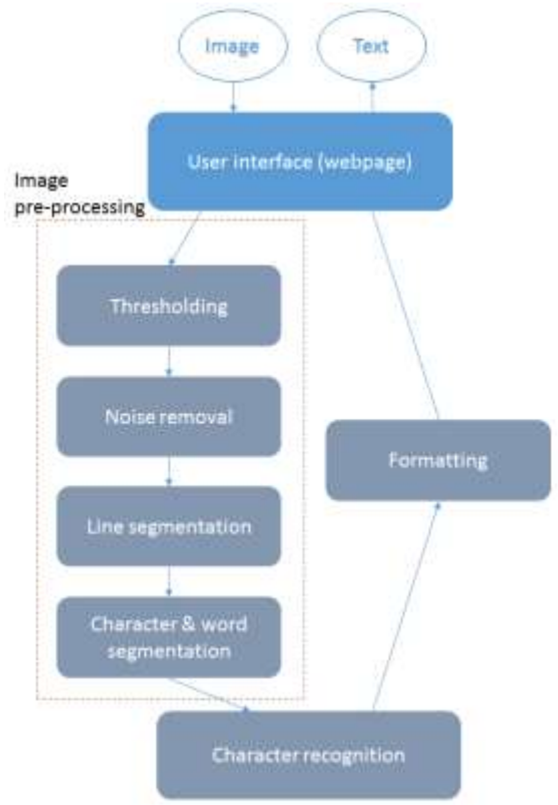

Figure 1 ScribeX architecture design

[2] 1Bpp means 1 bit per pixel.

4. Programming implementation

In choosing a programming language to implement this solution, preferences were given to high-level languages with strong support for object-oriented programming. This is to simplify algorithm implementation because user-defined classes is central in representing image and formatting data in this algorithm. Due to the limitation on performance of the algorithm of choice, i.e. K-Nearest Neighbor algorithm combined with Point Match character recognition, the performance limitation from programming language itself would be negligible.

The language of choice turned out to be C#, for the above consideration, and some advantages unique to the language:

1. **Easy implementation on lists**. C# provides various List data structure which turned out to be very handy in storing image and formatting data. It comes with useful built-in functions such as foreach() enumerator and Sort(), both of which were heavily exploited in the code of this project.
2. **Support for object and pointer arithmetic**. C# is a language within the C programming language family, whose members provide useful libraries in pointer manipulation. In addition, C# has good support for object-oriented programming as well.
3. **Language-level asynchronous programming**. With some of the APIs being time-consuming, it was the student's original idea to take advantage of C#'s language-level asynchronous programming to coordinate different modules of the solution, even though asynchronous programming eventually turned out to be unnecessary in the scope of this project.

Three major programming modules were developed to complete the solution: Web UI, ImageHelper class, and Knn namespace for character recognition.

5. Image pre-processing algorithm

The purpose of image pre-processing is to divide one big image containing multiple lines, words and digits into separate single-digit images, while preserving formatting information in the original image. This algorithm takes an image as input and processes it in five stages: thresholding, noise removal, line segmentation, word segmentation and character segmentation.

Notice that this algorithm, with proper input and output channel connected, can be used to generate training set images very efficiently. In fact, this is how the training set used was generated.

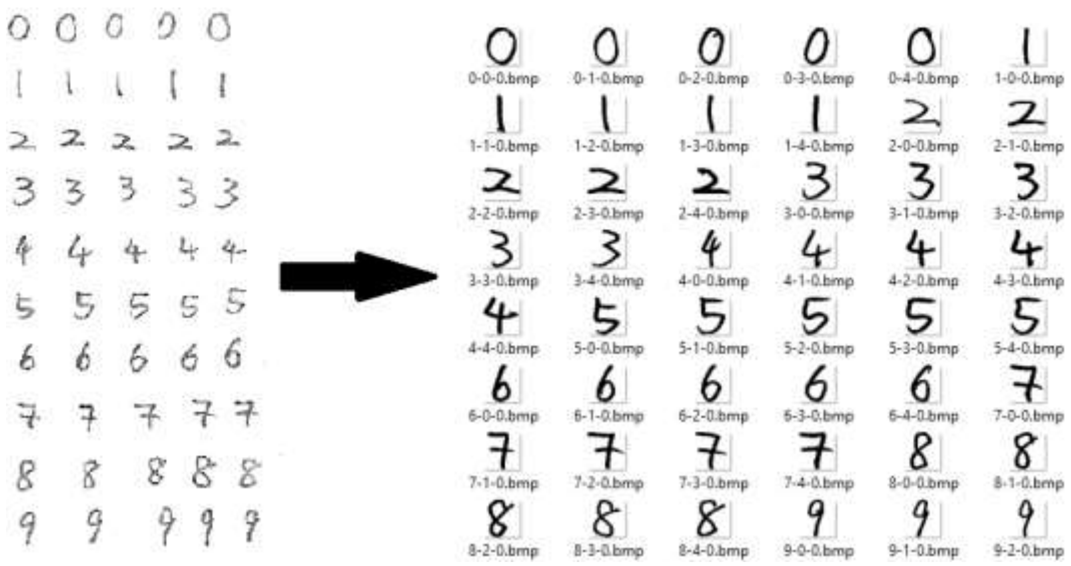
Figure 2 image pre-processing example, with processed images named LINE-WORD-CHARACTER.bmp

5.1 Thresholding

A thresholding function has a “threshold” value built-in. It takes an image as input and performs image binarization by cutting off any pixel below threshold value to blank, and promotes any pixel above that value to a full black one. The result is a strictly black-and-white image, but its pixel format is unchanged.

To make two processed images easily comparable, the algorithm also needs to copy the result image, possibly 24Bpp or 32Bpp ARGB formatted, into a 1Bpp indexed bitmap, which is equivalent to a binary-valued matrix.

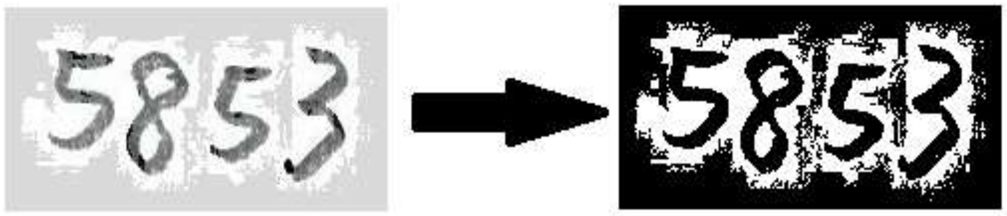
Figure 3 an example of thresholding (extra gray added to background to illustrate the effect of function)

5.2 Noise removal

Many techniques can be used for image noise removal, such as Mean smoothing and Median smoothing.

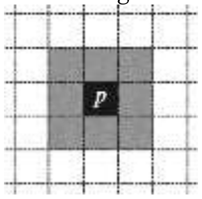

Figure 4 8-neighbors of p

The mean smoothing performs each pixel value's averaging with its 8 neighbors. The median smoothing replaces each pixel of the original with the median of neighboring pixel values. This project uses median smoothing as its noise removal technique.

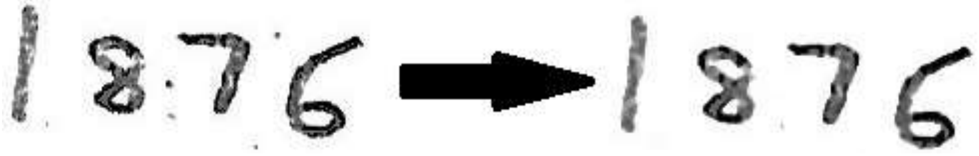

Figure 5 Noise removal example

5.3 Line segmentation

Line segmentation slices one big rectangle image into several "belt" images with the same width as original. It works as following:

1. Scan the image from top to bottom to find the first black pixel.
2. Draw a horizontal line tangential to this pixel, mark it as the top of a belt image, and scan downward to find the first horizontal line with all white pixels.
3. Once such a line is found, mark it as the bottom of a belt image.

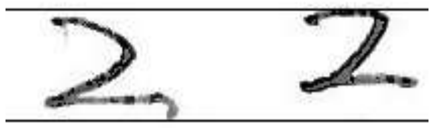

Figure 6 locating belt top and bottom

4. Add the belt image defined by the top and bottom lines to a list of belt images, and keep scanning downwards for the next belt image until reaching the bottom of the original image.

This step is sensitive to noise pixels and therefore requires the previous steps to operate beforehand.

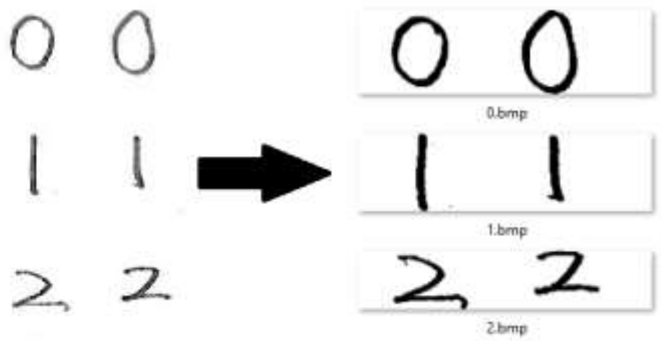

Figure 7 line segmentation example

5.4 Word segmentation

Word segmentation "chops up" a belt line image into several images, each representing a word. Such words, separated by spaces, are located using similar techniques as line segmentation: instead of searching horizontal lines marking the top and bottom lines of belt images, word segmentation searches vertical lines marking the leftmost and rightmost boundaries of words. After such boundaries are located, a belt image is cropped into individual images representing words.

There is, however, a noticeable difference between line and word segmentation: when line segmentation finds the top of a belt image, it looks for the bottom of that image by looking for the first horizontal line with all white pixels. This technique cannot be borrowed by word segmentation because even digits within a word are close to each other, there are still white spaces between them. Therefore, word segmentation algorithm must be able to set a threshold value for white space width, and ignore any white space thinner than threshold in word segmentation.

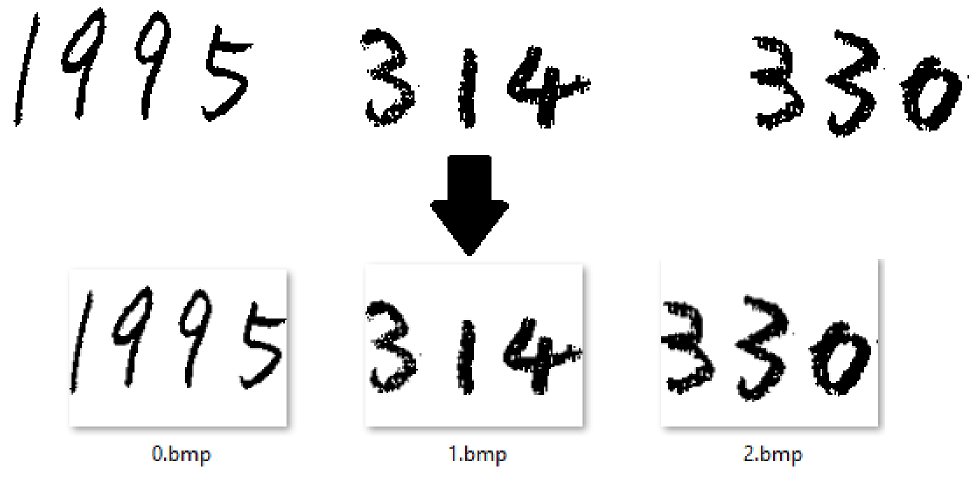

Figure 8 word segmentation example

5.5 Character segmentation

After line segmentation and word segmentation, the program obtained images on individual words. Character segmentation takes these words and extract single digits out of them. The technique used in this step is new: instead of locating image boundaries, this step uses neighbor-finding, a connected-component labeling technique.

The program iterates through all pixels in a bitmap image and once it hits a pixel not visited before, recursively marks all its neighbors, and their neighbors, and so on until no more unvisited neighbor can be found. This way, different connected components will be labeled into different equivalent classes, which is later translated into images of single digits. 8-neighbors are looked for in this step.

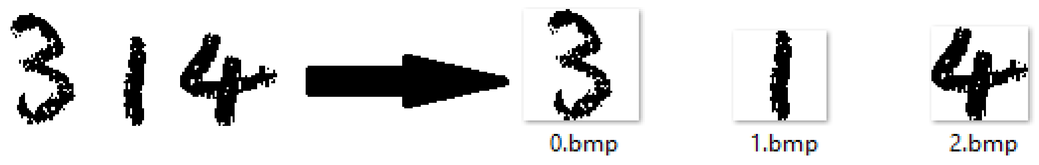

Figure 9 character segmentation example

To follow the formatting standard where single character image must be 25 × 25 pixels[3], the cropped image is padded into the smallest square containing it with white pixels, then resized to 25 × 25 pixels, before being returned.

[3] See next section for details on image formatting standard

6. Character recognition algorithm

The input of a character recognition algorithm is an image containing one single digit. The output is a char or int value which can be sent to an output UI. This algorithm is the heart of this project, and the following subsections provide a detailed discussion on its design and implementation.

6.1 K-Nearest Neighbor (KNN) algorithm

KNN algorithm is a commonly used algorithm in some basic machine learning problems: classification and regression. Specifically, this project uses KNN for classification purposes (we are trying to CLASSIFY an input image into one of the pre-defined CLASSIFICATIONS). The input of a KNN algorithm is an input unit to be classified, together with at least K training units, with a classification pre-defined for each training unit. The output is a classification membership.

After the algorithm starts, it will compare the input unit against each of the training units, and obtain a "difference" value between the input and each training unit. After all difference values are collected, they will be sorted and the K training units with smallest difference values will be defined as the "K nearest neighbors" of the input unit, and the input unit's classification will be decided by a simple majority vote by the K nearest neighbors.

For the scope of this project, the KNN algorithm works as follows:

1. Load training set images (10 digits * 20 images per digit = 200 images), and label each with a classification that is the digit it represents.
2. Given an input image, compare it against all 200 training images.
3. Sort all the difference values, and select the K training set images with smallest difference values.
4. Decide the classification of input image by a simple majority vote by the K nearest neighbors.

6.2 Training set

Because the input units for this program are images of single digits, the training set also consists of images of single digits. Each character from our alphabet is given 20 slightly different images.

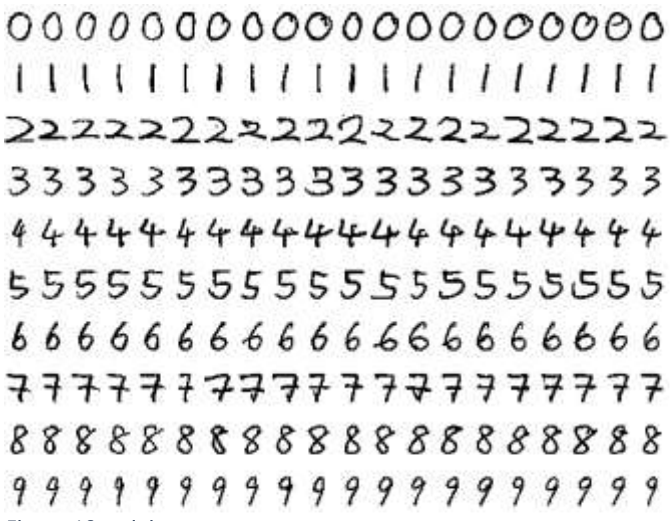
Figure 10 training set

Each training set image (and input image) follows the following formatting standard: the pixel format is 1Bpp indexed, hence the image must be monochrome. The dimension of an image is 25 × 25 pixels, which dictates the algorithm must process 625 pixel value pairs for each pair of input and training image. ImageHelper class provides an API to convert an image following such formatting standard to a two-dimensional bit array for comparison (ImageHelper.MonoBitmapToBoolArray()).

6.3 Image comparison and difference value

The trickiest part of implementing this conceptually simple algorithm is how to compare two images and how to define the "difference" value between them. Given two pairs of integers, how would a program know which ones are closer to each other?

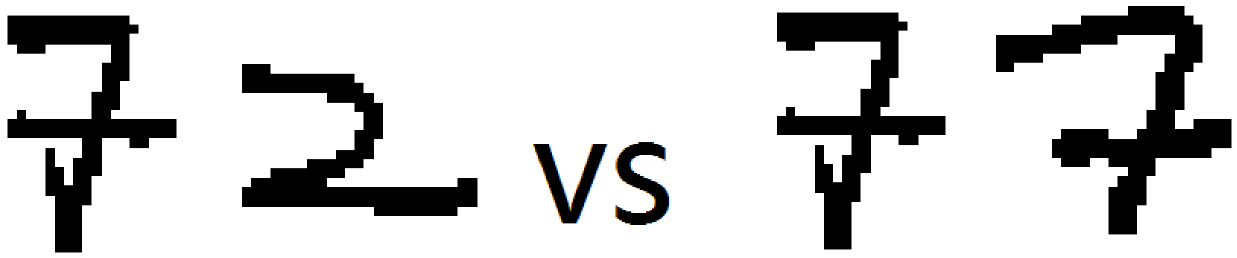

Figure 11 a human would obviously define the left pair as "closer match" - but how to make a program that does the same?

Several ideas were explored and tested to solve this problem, all of which are based on Point Matching strategy: whenever we find a point in one image that doesn't have a matching point in the other image, we increment the difference value between said two images.

6.3.1 Direct comparison

Given the formatting of our input and training images (monochrome, 25 × 25 pixels in dimension), it is a natural idea to simply overlay one image onto the other and observe the number of different pixels. As a matter of fact, such a strategy yields a high accuracy when the input characters aren't twisted too much from training set characters, and it is the fastest in performance, with an average running time of ~ 0.2 seconds per character.

So, how would we instruct the computer program to do this direct comparison? The approach taken here is to convert a monochrome bitmap image into a two-dimensional binary bit array, and compare all the entries at the same location.

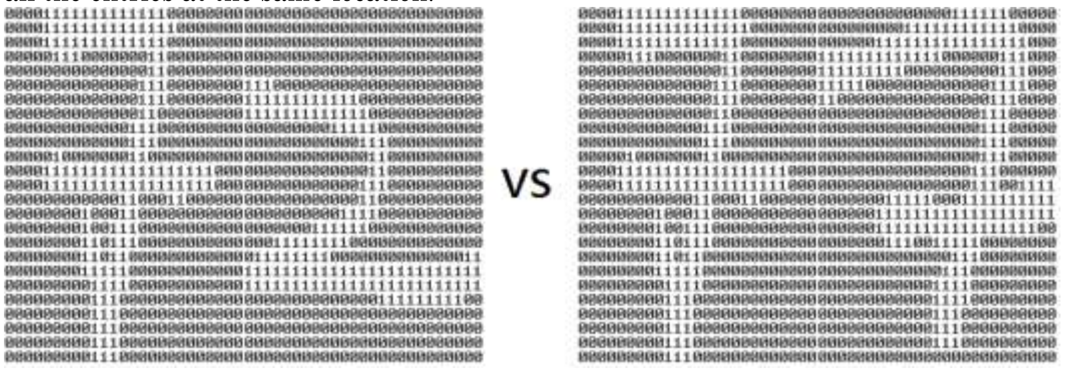

Figure 12 computer programs can easily compare two-dimensional bit arrays

Pseudocode for comparing two bit arrays:

```
Foreach (x,y) from (0,1) to (24,24)
    If BitArray1(x,y) != BitArray2(x,y)
        DifferenceValue ++;
```

6.3.2 Bold and hit

One obvious problem raised by direct comparison method described above is that, if two images are highly similar in shapes, but slightly distorted, rotated or misplaced, the difference value between them can balloon up unnecessarily.

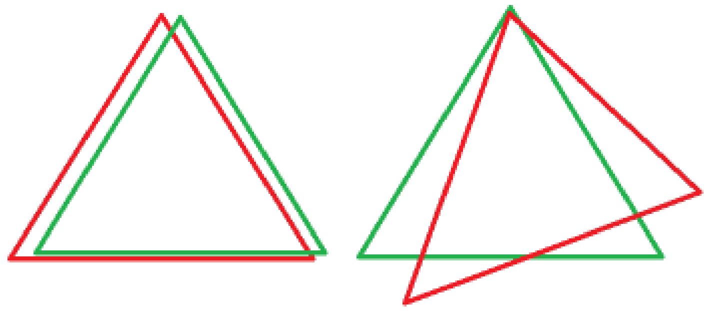
Figure 13 what if two images look like these two pairs?

A solution to this problem was explored and showed good performance. This solution, which we will call "bold and hit", enhances (or "bold up") the training set image, and iterates through every black pixel in input image, checking if there is a matching black pixel in the enhanced training set image. If such a matching point cannot be found, the algorithm increments the difference value.

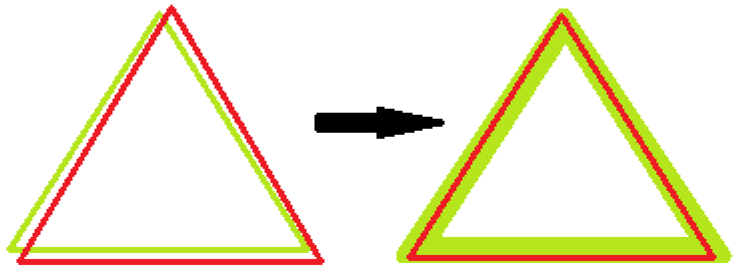
Figure 14 the lime triangle is bolded up, so when we use pixels from red triangle to hit them, there will always be a match.

ImageHelper class provides an API called BoldAndEnhance() to perform the task of enhancing a monochrome image.

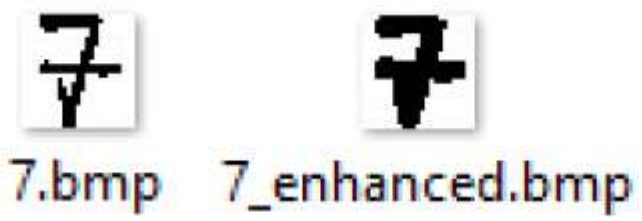

Figure 15 BoldAndEnhance() example

Pseudocode for bold and hit method:

```
Bold up training set image;
Foreach (x,y) from (0,0) to (24,24) in input image
        If inputImage(x,y) == blackPixel && trainingSetImage(x,y) != blackPixel
                DifferenceValue ++;
```

6.3.3 Procrustes analysis

Bold and hit provides a naïve approach to factor out the translation, scaling and rotation effects in shape comparison. Another mathematical approach, named Procrustes Analysis, was also explored in the attempt of obtaining translation, scaling and rotation invariant images. It uses a set of techniques, collectively referred to as "superimposition", to produce translation, scaling and rotation invariant images, and compare their differences afterwards.

In the scope of this project, Procrustes analysis is performed in the following stages[4]:

1. **Convert 1Bpp indexed bitmap images into collections of points**. The program defines a coordinate system with the origin lying at the bottom-left point of a 25 $\times$ 25 image, and each black pixel corresponds to a point with integer-value coordinates, between (0,0) and (24,24).
2. **Downsample** one of the two images to ensure two images have the same number of points, if necessary. Downsampling refers to systematically removing some data points from the collection with more points, in order for two collections to have the same cardinality. This is necessary in removing rotational components of two shapes.
3. **Factor out translational components**. Suppose a point collection contains k points in total, say $((x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k))$, then the mean of these points is given by $\bar{x} = \frac{x_1 + x_2 + \cdots + x_k}{k}, \quad \bar{y} = \frac{y_1 + y_2 + \cdots + y_k}{k}.$, and a translation of the centroid of the image to the origin is performed by $(x, y) \to (x - \bar{x}, y - \bar{y})$.
4. **Uniform scaling**. The purpose of this step is to ensure two shapes to be compared have the same size. An image's scale can be measured by its root mean square distance (RMSD):
   $$s = \sqrt{\frac{(x_1 - \bar{x})^2 + (y_1 - \bar{y})^2 + \cdots}{k}}$$
   Then we scale the image to size 1 by $((x - \bar{x})/s, (y - \bar{y})/s)$.
5. **Factor out rotational components**. For a point (x,y), rotating by an angle of θ is equivalent to multiplying a rotation matrix
   $$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}.$$
   For two sets of points $((x_1, y_1), \ldots)$ and $((w_1, z_1), \ldots)$, the angle θ to eliminate rotational differences between the two can be calculated by:
   $$\theta = \tan^{-1}\left(\frac{\sum_{i=1}^{k}(w_i y_i - z_i x_i)}{\sum_{i=1}^{k}(w_i x_i + z_i y_i)}\right).$$
6. Convert collections of data points back to monochrome bitmap images for bit array comparison. This can be done by mapping the range of x-coordinates and y-coordinates, respectively, of data points onto the range of [0,24], round each coordinate value to an integer value, and generate 25 $\times$ 25 images by pixel locations.

[4] Latex-formatted images taken from Wikipedia "Procrustes analysis" page.

Note: because training set images and input images were pre-processed into 25 × 25 monochrome images, in practice step 4, uniform scaling, is often found unnecessary.

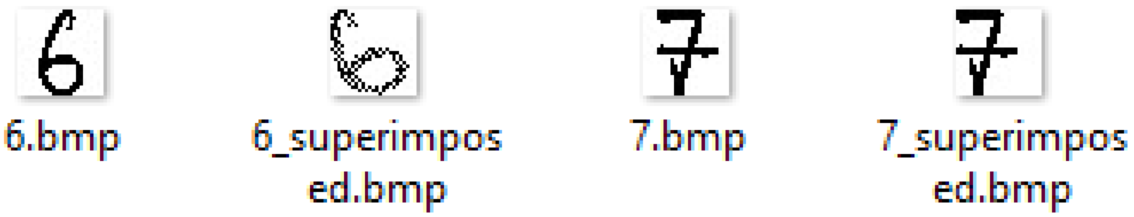

Figure 16 Procrustes analysis on a pair of images of 6 and 7

Pseudocode for Procrustes analysis method:

```
    Convert inputImage and trainingSetImage into data point collections;
    Superimpose collection1 and collection 2;
    Convert collection1 and collection2 back to images;
    Use direct comparison method to compare two processed images.
```

6.3.4 Hausdorff difference method

The above methods are all based on bit array comparison: they optionally process the images, and convert the images into bit arrays for comparison. What if we can calculate the differences between data points directly, not through a point-by-point binary comparison? In fact, normally Procrustes analysis calculates the difference between two superimposed shapes by a mathematical method:

$$d = \sqrt{(u_1 - x_1)^2 + (v_1 - y_1)^2 + \cdots}.$$

But because our data points in collections aren't ordered, we cannot apply this formula. How would we calculate the distance between two sets of points with a mathematical approach? Here we introduce Hausdorff distance, a measure of the distance between two sets of points. Intuitively, if every point of a set is close to some point of the other set, the two sets will have small Hausdorff distance.

The implementation of this concept can be complex, but in the scope of this project we use a naïve and time-consuming approach to calculate Hausdorff distance: for each data point in a set, loop through all points in the other set, find the one with smallest distance in 2-D space, and use increment the total Hausdorff distance by that smallest distance value.

Pseudocode for Hausdorff distance method:

```
    Convert inputImage and trainingSetImage into point collections, collection1 and collection 2;
    HausdorffDistance = 0;
    Foreach(point p1 in collection1)
        MinDifference = infinity;
        Foreach(point p2 in collection 2)
            If(difference(p1,p2) < MinDifference)
                MinDifference = difference(p1,p2);
        HausdorffDistance += MinDifference;
```

6.3.5 Edge detection

In image processing, an edge point refers to a point where the brightness of the image changes sharply. Edge points are usually used as "landmark" points when extracting shapes out of images. Edge detection refers to the activity of locating such edge points in an image.

In this project, edge points can be used to represent all the points in an image, therefore reduce the size of point sets and save computational time. Because we deal with monochrome images, an edge point can be defined as a black point with at least one white neighbor, either a 4- or 8-neighbor, because that would imply this point lies on the edge of the image.

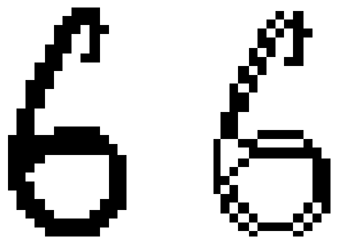
Figure 17 an image of 6 and its edge points

6.3.6 Choosing a method to calculate image difference

The techniques discussed above can be combined in implementing the algorithm to calculate the difference value between two images. For example, we can choose to apply edge detection on a pair of images, followed by Procrustes superimposition, then use direct comparison (bit array comparison) to obtain a difference value.

In the development of this project, various combinations of above-mentioned techniques were implemented and tested. As it turns out, in the scope of this project, where our alphabet consists of ten digits only, the following combination shows the highest accuracy:

Procrustes superimposition + *Bold and hit*.

This is the method chosen for the implementation of ScribeX. The API corresponding to this combination is ScribeX.Models.ScribeXCharacterRecognition.Knn.KnnCore.RecognizeSuperimposedAndEnhancedImageWithBoolArrayMethod().

6.4 A special case: 3 vs. 8

A special case where our chosen method of *Procrustes superimposition* + *Bold and hit* performs poorly is the recognition of number 3.

The method of choice observes the points present in input image but absent in enhanced training set image, which by its nature will perform poorly in recognizing number 3: with normal handwriting, a "3" is really a subset of an "8", so any point present in an image of 3 will easily be matched by a point in an image of 8, resulting in a difference value of (almost) 0.

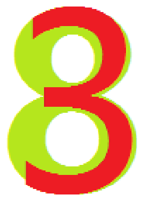

Figure 18 a "3" image is almost a subset of an "8" image

This fault happens by the nature of Bold and hit method, but it can be mitigated. Suppose at some point, the algorithm took an input image and classified it as an 8. There are two possibilities: either the original input image actually represents an 8, or it was a 3 and led to an error. However, notice that the confusion of 3 and 8 is unique to Bold and hit method, so we can use some other methods to re-evaluate any image classified as 8 by Bold and hit.

In the implementation, if Bold and hit classifies an image as 8, two "backup" methods, direct comparison and Procrustes superimposition + Hausdorff distance, are applied to recognize the input image. The output is then decided by a simple majority vote by the output from three methods. If the image was indeed an 8, 8 would get the most votes. If the image was a 3, the two backup methods will both recognize it as 3, and the original classification, 8, will be ruled out.

In testing, this trick eliminates about 60% of errors where 3 is misrecognized as 8.

7. Performance analysis

In general, ScribeX is a time-consuming application, with average processing time for each single digit image being approximately 1.5 – 1.8 seconds, because of the nature of its underlying algorithm, point matching. There are several elements affecting the overall performance. In general, the running time of the program follows:

T (pre-processing) = O (Number of lines × average number of words in a line × average number of characters in a word);
T (character recognition) = O (Number of training set samples × average number of points in a training sample × number of points in input sample).*

*Note that for direct comparison method, the average number of points in a training sample and the number of points in an input sample are always 625.

7.1 Training set size

The KNN algorithm must check the input image against each training set image, making the size of training set an important factor affecting overall performance. Users can use ScribeX's website with different training set size to verify that when training set size is 20 instances per character, the time need for an input image would be 10 – 15 times longer than using only 1 instance per character.

7.2 Procrustes analysis

Procrustes analysis, especially the superimposition part, proves to be also time-consuming: not only does it have to loop through each pixel in images to locate all the black pixels, but also it has to perform on each black pixel arithmetic operations, which are orders of magnitudes slower than binary comparisons. Generally a character recognition method takes about twice as much time when equipped with Procrustes superimposition.

7.3 Hausdorff distance calculation

When Hausdorff distance is used as the measure of difference between two images, the performance is generally worse than comparing the bit arrays directly. With the implementation of Hausdorff distance in this project, the cost to compute the Hausdorff distance between two images is:

Num (Calculating Hausdorff distance) = O (N^2), where Num is the number of arithmetic operations needed, and N is the average number of points (black pixels) in each image.

Meanwhile the cost of direct comparison by bit array method is:

T (bit array comparison) = O (Number of pixels in an image), where number of pixels in an image is fixed to be 625.

8. Limitations

Although widely studied in many introductory courses and tutorials in machine learning, handwriting recognition turns out to be a difficult problem. While the works introduced above provided a basic solution, many problems emerged in the development of this project were mitigated, not completely solved, leaving the solution of this project in a limited scope. Major limitations a user may notice when using ScribeX application include computation speed, limited alphabet size, and limitations on handwriting styles accepted by the program.

The most obvious limitation lies in time performance. As discussed in the previous section, computation time grow drastically when training set size, training set image dimension or number of input characters increase. With a point match strategy, the algorithm underlying is essentially a slow one. For example, with 20 training images per character, recognizing Sample Image 1 provided with the web app (13 characters) takes about 22 seconds, but recognizing Sample Image 2 (51 characters) requires about 90 seconds, and Sample Image 3 (96 characters) needs about 170 seconds.

Another limitation is the alphabet size. When this project was initially proposed, it was expected to be able to recognize the full English alphabet, namely 26 upper case letters and 26 lower case letters in addition to 10 digits currently available. However, when the training set expanded from 10 characters to 62 characters, not only did the running time grow dramatically, but also the accuracy of the classification algorithm went down to only ~50%. In order to distinguish 62 characters, it requires much more complicated algorithm design.

In addition to the limitations on character recognition, there is also a limitation on image pre-processing: the handwritten digits cannot be connected to each other. This is because the image pre-processing algorithm uses connected-component labeling to separate digits in a word, and therefore if one digit has a stroke connected to any part of another digit, they will be recognized as one single character, and the character segmentation algorithm would produce undesirable results.

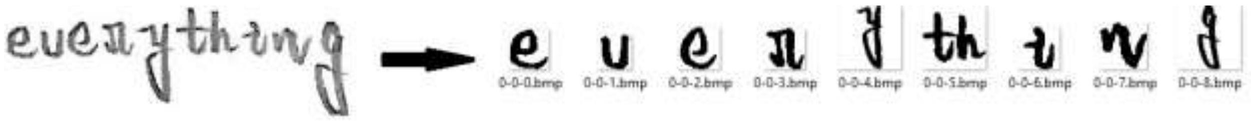

Figure 19 letters "t" and "h" are segmented into one single image because the horizontal stroke of "t" is touching the edge of "h".

9. Takeaways

This project was completed with a great deal of research and programming efforts. Despite the limitations on its final solution, it was an intensive, enjoyable and rewarding learning experience. By working on this project, the student was exposed to a full software development cycle: a challenge, an idea for solution, researches, programming implementation, and debugging and testing.

In addressing the challenges of this project, the student explored a wide range of topics in the field of computer science: machine learning, KNN algorithm, image segmentation, connected-component labeling, shape comparison, Procrustes analysis and Hausdorff method, etc. Studying these topics helped the student brush up his research skills.

On the programming side, a lot of cutting-edge technologies were also explored. Programming in C# was a great opportunity to get familiarized with Microsoft .NET framework, and making the web application was a good practice in web development. This project is heavy on image processing, giving the student an opportunity to explore Windows GDI+ (Graphics Device Interface +) library. Because part of this program involved pointer arithmetic, which can cause memory leaks after the program started on the server, the student also got to play with Microsoft Azure Remote Debugging tools.

10. Facts and numbers

Some facts about this project:

- This project was completed in about 60 – 80 hours in total.
- The alphabet of ScribeX is the collection of single digits, i.e. {0,1,2,3,4,5,6,7,8,9}.
- Each character in the alphabet is provided with 20 training set images, making the training set a collection of 200 images.
- The average accuracy in digit recognition is ~90%.
- The character to get misrecognized most likely is 3. The character to get misrecognized least likely is 4.
- When the alphabet of this project was temporarily expanded to 62 characters including all English alphabet letters, the most misrecognized letter were a, v and w, often to each other.
- The source code of ScribeX has 855 lines of code, excluding comments, carriage returns, brackets that takes entire lines, html and css files, and testing project.

- 13 commits were committed in Git for this project. All of them happened after ImageHelper, KnnCore and ProcrustesAnalysis classes were finished.
- Average running time of ScribeX is about 1.5 – 1.8 seconds per character.
- The only NuGet library used in ScribeX project is AForge.NET, which helps in two ScribeX APIs: image thresholding and noise removal. Other APIs are all implemented by the student.
- Upon completion of this project, the student is expected to get 2 more hours of sleep every night.

Here is a list of APIs provided by this application that may be of other programmers' interest:

```
ScribeX.Utility.ImageHelper.PreprocessScannedImage()
ScribeX.Utility.ImageHelper.LoadImage()
ScribeX.Utility.ImageHelper.SaveImage()
ScribeX.Utility.ImageHelper.Convert1bppTo24bpp()
ScribeX.Utility.ImageHelper.MonoBitmapToBoolArray()
ScribeX.Utility.ImageHelper.CropImage()
ScribeX.Utility.ImageHelper.PadImage()
ScribeX.Utility.ImageHelper.BoldAndEnhance()
ScribeX.Utility.ImageHelper.GetIndexedPixel()
ScribeX.Utility.ImageHelper.ThresholdAndConvertTo1Bpp()
ScribeX.Utility.ImageHelper.RemoveNoise()
ScribeX.Utility.ImageHelper.LineSegmentation()
ScribeX.Utility.ImageHelper.WordSegmentation()
ScribeX.Utility.ImageHelper.CharacterSegmentation()
ScribeX.Models.ScribeXCharacterRecognition.Knn.KnnCore.RecognizeImageWithBoolArrayMethod()
ScribeX.Models.ScribeXCharacterRecognition.Knn.KnnCore.RecognizeSuperimposedImageWithHausdorffMethod()
ScribeX.Models.ScribeXCharacterRecognition.Knn.KnnCore.RecognizeSuperimposedImageWithBoolArrayMethod()
ScribeX.Models.ScribeXCharacterRecognition.Knn.KnnCore.RecognizeSuperimposedAndEnhancedImageWithBoolArrayMethod()
ScribeX.Models.ScribeXCharacterRecognition.Knn.KnnCore.RecognizeParagraph()
ScribeX.Models.ScribeXCharacterRecognition.Knn.ProcrustesAnalysis.ComputeHausdroffDistance()
ScribeX.Models.ScribeXCharacterRecognition.Knn.ProcrustesAnalysis.SaveSuperimposedImage()
ScribeX.Models.ScribeXCharacterRecognition.Knn.ProcrustesAnalysis.SaveTranslatedImage()
```

References:

[i] Plamondon, and Srihari. "On-line and Off-line Handwriting Recognition: A Comprehensive Survey." *IEEE Transactions on Pattern Analysis & Machine Intelligence* 22.1 (2000): n. pag. Web.
[ii] Li, Yujia, Kaisheng Yao, and Geoffrey Zweig. "FEEDBACK-BASED HANDWRITING RECOGNITION FROM INERTIAL SENSOR DATA FOR WEARABLE DEVICES." (n.d.): n. pag. Microsoft Research. Web.
[iii] Stanek, Steven, and Woodley Packard. "Greedy Point Match Handwriting Recognition." (2005): n. pag. University of California, Berkeley. Web.
[iv] Mathur, Aggarwal, Joshi and Ahlawat. "OFFLINE HANDWRITING RECOGNITION USING GENETIC ALGORITHM." (2008): n. pag. Sixth International Conference on Information Research and Applications. Web.