TJHSST Senior Research Project
Idocrase: Development of a Generic Font OCR System
2007-2008

Nathan Harmata

June 6, 2008

Abstract

OCR (Optical Character Recognition) is a very practical field of Computer Science. Since the late 1980's, researchers have been developing systems to identify text from non electronic text sources, like pictures or papers. The use of OCR systems has spanned from making books in Braille to sorting mail by zip code by United States Post Office. This project describes the development of an entire OCR system in Java.

Keywords: OCR, Optical Character Recognition, Image Processing, Computer Vision

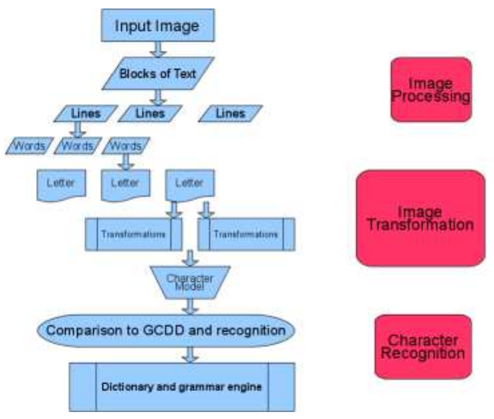


Figure 1: An overview of the OCR process.

1 Introduction

The goal of this project is to create an application that can read text from electronic picture files. One of the main focuses is developing a generic way to recognize characters of different fonts, rather than hardcoding in definitions for specific fonts. All of the algorithms and processes used in this project are original work.

Although OCR is by no means a "new" field, it has still yet to be fully explored. Many common computer users either don't have access to an OCR program or don't know they have one, and some of the ones that are free of charge are lacking in performance and consistancy.

2 Background

OCR systems have been around since the late 1980's. Still, they are not widely available or used by the public. The results from a review of the free ones on the Linux operating system are not very promising. [1] Although most of them had measured accuracies above 94 percent, that is not good enough. The one commercial product tested, Aspire OCR, was accurate only 91.5 percent of the time. The most likely industry standard, Tesseract, is also one of the oldest OCR systems. The review measured it to have an accuracy rate of 99 percent. Development on it started in 1985 and it is still used as the OCR engine for Ocropus, Google's textual analysis application. The goal of this project is to make a fairly accurate,

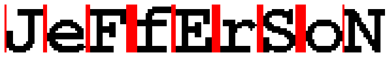


Figure 2: Parsing the word "JeFfErSoN" into its letters.

working OCR system.

3 Procedures

The Idocrase OCR system works by accepting an image of text, and, through a series of parsings and transformations, is a able to produce a basic form for each character of text in the image. There are three main steps in the OCR process: Image Processing, Image Transformation, and Character Recognition.

3.1 Image Processing

The input image is not just going to be plain text in a nice and simple format. The objective is to get the text into such a form through a series of parsings. First, paragraphs are recognized and separated. Then, paragraphs are broken down into individual lines, and lines into words. This is all done on the premise that the image is of computer generated text. This ensures that there are straight lines of whitespace between adjacent paragraphs, lines, words and words. One key part of the text parsing process is that the optimal separations are made; paragraphs, lines, and words are "boxed" as closely as possible, to minimize errors. Each word is then processed into its individual letters. Once again, the premise of the existence of whitespace between adjacent characters is used.

3.2 Image Transformation

Each character image then undergoes a series of transformations into what are called "Attributes." Attributes of the image are then combined into what is called a "Character Model." A Character Model serves as a generic character definition. That is, it defines what the character would be like irrespective of font. The purpose of this is that each Character Model can be compared to a pre-generated database of generic character definitions for each possible letter, number, and symbol, and the best match can be found. The steps through which each character image goes are as follows.

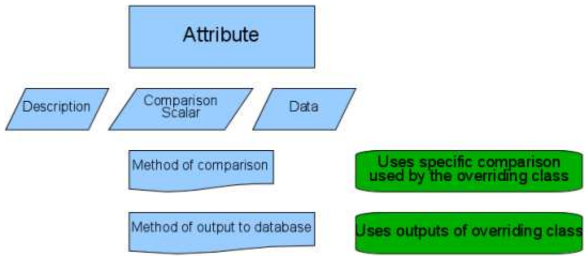

Attribute
Description
Comparison Scalar
Data
Method of comparison
Method of output to database
Uses specific comparison used by the overriding class
Uses outputs of overriding class

Figure 3: An overview of the Attribute class.

3.2.1 Attribute

An Attribute is one specific representation of the image and is used as a comparison heuristic. Each Attribute is able to compare itself to other Attributes of the same type and is able to output its relevant data. Each Attribute also knows what type of Attribute it is. This fact is very important and will be explained later. In the actual code of this project, there is an Attribute class which each specified characteristic extends. Currently, this project uses two Attributes, "Sector Vector" and "Gap Vector."

1. **Sector Vector**

   A Sector Vector consists of three pieces of data: the number of "sectors" in the image, the total number of line segments in the transformed form of the image, and the sign of the slope of the first such line segment. A sector is defined as a region of the image that passes the vertical-line test; that is, for each x-coordinate in the image, there is at most one y-value. The purpose of parsing the image into sectors is so each sector can be parsed into line segments. Since the line segment parsing is based on the slopes of adjacent pairs of pixels, sectors parsing is necessary. The algorithm of transforming an image into a Sector Vector consists of two steps, Sector Parsing and Slope Field Parsing.

   (a) Sector Parsing

   Starting from the top of the image, the program progresses downward until it reaches a point at which there is a conflict with the portion of the image already processed that would cause a failure of the vertical-line test. This point represents the end of

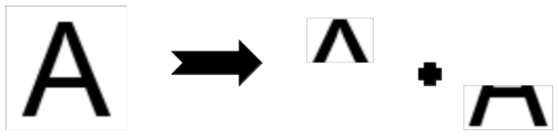


Figure 4: An example of Sector Parsing. The "A" is parsing into two sectors: the upper half and the lower half.

one sector and the beginning of another. This process is repeated until the entire image is parsed in sectors. The result is that each sector is as large as possible, within the constraints. The important piece of data gained from this process is the number of sectors in the image. Each sector is then parsing into line segments.

(b) Slope Field Parsing

Starting with the left-most pixel in the sector, the program progresses to the right. At each x-coordinate, a line segment is constructed between the pixel at that x-coordinate and last pixel considered. If the slope of this line segment is similar, in sign and magnitude, to the slope of the previous line segment, then it is incorporated into the previous by changing the last pixel of the previous one to the last pixel of the current one. If the slope is radically different, then a new line segment is constructed. The results is that the image in transformed into a collection of line segments. There are two important pieces of data derived from this step: the number of line segments and the sign of the slope of the first line segment in this sector.

A sample sector vector is:

```
-2 3
```

This means that the image contains two sectors, the sign of the slope of the first line segment in the first sector is negative, and there are a total of three line segments. This happens to be the Sector Vector representation of "C" for most fonts.

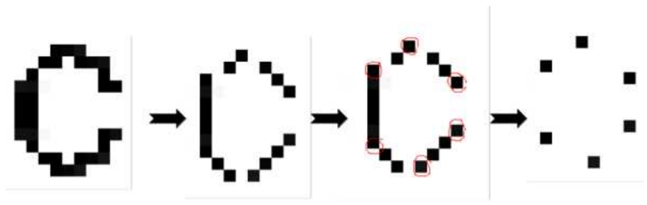


Figure 5: An example of parsing a "C" into line segments. The image on the right contains the pixels which, when connected, form the line segments representing this image. Note that the Sector Parsing of the "C" is not shown in this diagram.

Interestingly, a similar method of segment parsing was independently developed by two researchers. It does not use a process similar to Sector Parsing; instead it parses the image into a set of predefined line segments. [2]

2. **Gap Vector**

A Gap Vector is simply what, if any, "gaps" are present in the image. A gap is defined as a breakage of pixels on one of the four edges of the image: top, right, bottom, and/or left. The purpose of having such a comparison heuristic is the assertion that gaps are more representative of a character than line segment parsing. That is, same characters of different fonts are more likely to have the same gaps than they are to have the same Sector Vector. Also, Gap Vector provides information that is exclusive from the information given by Sector Vector. The presence of a gap isn't likely to have any correlation to the presence of sectors. Unlike sectors, the definition for a gap isn't simple. Both concepts were invented for the purpose of this project, as were their working definitions. This project defines a gap using the algorithm by which they are located.

(a) Corning Finding

The first step in the gap finding process is to locate the four corners of the image. Each image is defined to have four corners; even if it is not visually obvious to a human that there are four corners, such as is the case with a letter like "O," four corners are forced on the image. There are four corners: top left, top right, bottom left, and

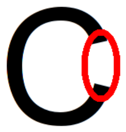


Figure 6: The portion of the image circled in red is a "gap" in the "C".

bottom right. A corner is defined as the intersection of the path starting from one corresponding extremum of the image and the path starting from the other corresponding extremum of the image. For example, the top left corner is the intersection of the path starting from the bottom left extremum of the image and the path starting from the top left extremum of the image. The corner is, of course, a single point. Therefore this intersection is the one that occurs when the path progress towards each other at the same rate. This process is repeated to find the locations of the four corners of the image.

(b) Path Tracing

The next step is to use the corners to determine if there are any gaps in the image. Between any two adjacent corners lies one of the sides of the image. Any gap on that side, by definition, must be between those corners. The algorithm uses this fact to its advantage. For each pair of adjacent corners, it iterates across the straight line between them. As each point on this line, it determines if the corresponding point in the actual image is, in respect to the side of the image on which the computation is occuring, in front of or behind of the line. For example, consider finding a gap on the left side of an image. On the left side, the slope of the path between the two corners forming that path, the top left and bottom left, is in respect to a vertical line. That means that, for a coordinate on the line (a) and a coordinate on the image (b), a comparison of the x-coordinates of those coordinates can be used to determine the relative location of the coordinate on the image. Since the

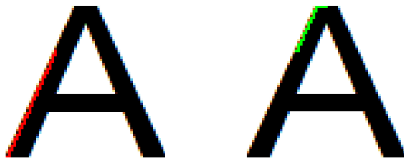


Figure 7: An example of the process of finding the top left corner of "A". The image on the left shows the path from the bottom in red and the image on the right shows the path from the top in green.

left side of the image is being considered, if the difference between them is positive, then the one on the image is behind the one on the line. That is a "is in front of" b if:

$$a_x - b_x > 0 \tag{1}$$

The sum of all the distances between the line and the corresponding points is calculated, keeping in mind whether the point was in front of or behind the line. The result is that if more of the pixels are behind the line, this sum is negative. That means that more of the image itself is behind the line, which implies a gap. This computation is actually simply comparing the area of the part of the image in front of the line with the area of the image behind the line, in respect to either a straight line or a horizontal line, whichever is more appropriate. Thus, a gap exists on a side if there are more pixels behind the line between the corners forming that side than in front of it.

The result is a list of gaps, represented by the strings, 'T', 'R', 'B', and 'L' for 'Top', 'Right', 'Bottom', and 'Left', respectively. A sample Gap Vector is:

```
R
```

The means that there is a gap on the right side of the image, such as is the case for a 'C'.

3. **Pixel Concentration Vector**

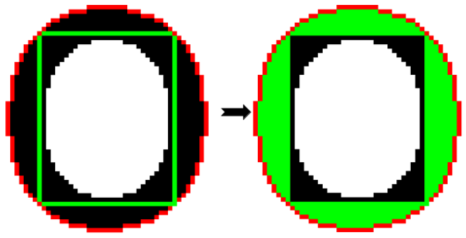


Figure 8: An example of the Path Tracing algorithm for "O". The pixels in red are the ones on the actual image. The green lines in the left image show the straight line paths between the adjacent corners. The areas in green in the right image show the portions of the image "in front of" their respective straight line paths. Since all of these areas are greater than the areas behind the paths, which happens to be 0 for "0", there are no gaps in the image.

A Pixel Concentration Vector contains two pieces of information about an image. It tells which horizontal side and which vertical side contains more pixels of text. This Attribute was created to deal with the fact that the Sector Vector and Gap Vector Attributes do not differentiate between similar characters that have been translated and/or rotated. For example, a lowercase "p" has almost an identical representation to a lowercase "b".

A Pixel Concentration Vector is generated for an image by simply counting the number of favorable (i.e. text) pixels to the left and right of the horizontal midpoint and to the top and bottom of the vertical midpoint. Becuase of this, it is important that the image has been properly processed and boxed before going through this process.

The values for the horizontal concentration are "left" and "right" and the values for the vertical concentration are "top" and "bottom". In addition, a concentration can have a value of 'balanced' if there is not a significant difference between the number of pixels on each side.

For example, the Pixel Concentration Vector for

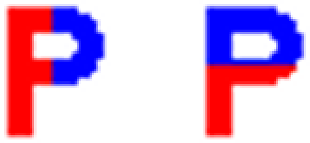


Figure 9: A example of the computations needed to determine the horizontal and vertical concentrations of "p".

lowercase "p" is:

```
top left
```

This means that more pixels were on the top side and left side of the image, respectively.

The importance of each Attribute knowing what type of Attribute (Sector Vector, Gap Vector) it is, called its "description", is so that different Attribute representations for an image can be easily grouped together. This grouping is called a "Character Model."

3.2.2 Character Model

A Character Model contains a collection of Attributes and the processes to use those Attributes for character recognition purposes. The Attributes are stored in a HashMap based on the hashcode of their description; this is done so that they are always stored in the same order, making comparisons between like Attributes easier.

1. Comparisons are relatively simple; two Character Models are treated as vectors and the magnitude of the vector difference between then is calculated. The "elements" in the "vectors" are Attributes; therefore the definition of a difference of Attributes is the result of the comparison between them defined by their Attribute class. Thus, a comparison between Character Models A and B, each having n Attributes, is:

$$\sqrt{\sum_{i=1}^{n}(A_i.compareTo(B_i))} \quad (2)$$

2. A Character Model also has a method of outputting its important data. It does this by using the output of its Attributes and their respective descriptions. For example, the Character Model output for the "C" image, which has been used as an example, is:

   ```
   SectorVector -2 3 GapVector R
   ```

   Character Models are hashed based on the hashcode of this output string, much like the hashing process for Attributes. This is done to ensure that Character Models that are identical in respect to their data have the same hashcode.

3. Character Models can also be averaged together. This is important for the generation of the Generic Character Definition Database. Character Models are averaged together by averaging each Attribute. Much like the comparison process, each Attribute defines how to average its own members. For example, the average of a set of Gap Vectors is the most popular gap. The average of Character Models A through P, each with n Attributes, is:

$$\sum_{i=1}^{n}(average(A_i, B_i, \cdots P_i)) \quad (3)$$

3.2.3 Generic Character Definition Database

The purpose of all these parsing and transformations is to get the input image into a generic form. This form can then be compared, using the methods of comparison already outlined, with a pre-generated database of generic forms. Such a database, called a Generic Character Definition Database (GCDD), is created by performing the analysis procedure on each character (letters, numbers, and other symbols) for several fonts and averaging the results, which are then outputted along with the characters they represent into a file. This database is handled by the system using the outputs of the Character Models in it. Because of this, it is important that the output includes the Attribute descriptions in addition to the Attributes data.

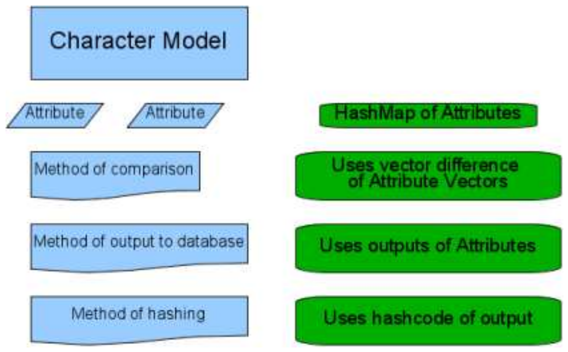


Figure 10: An overview of the Character Model class.

3.3 Character Recognition

The Idocrase OCR system "reads" text by choosing the letter(s) that, after the input image has gone through the Image Processing and Image Transformation steps, is the "closest" to the Character Model(s) of the input image.

For now, assume the input image's text only contains one letter. The process of finding the best match can be represented graphically. Each Attribute (Sector Vector, Gap Vector, and Pixel Concentration Vector) is one of the coordinates axes. Thus, this space contains possible Character Models. Each member of the GCDD is plotted in this space. Then, the input image goes through the Image Processing and Image Transformation steps and is also plotted. The best matching character is the one that is the 'closest' in this Character Model space to the point representing the input image. This distance is equivalent to comparing the Character Model of the input image to the Character Model of the best match, which is defined by the 'compareTo' method of the Character Model class.

Images containing more than one character of text are more difficult to 'read.' The system finds the collection of the appropriate number of characters in the

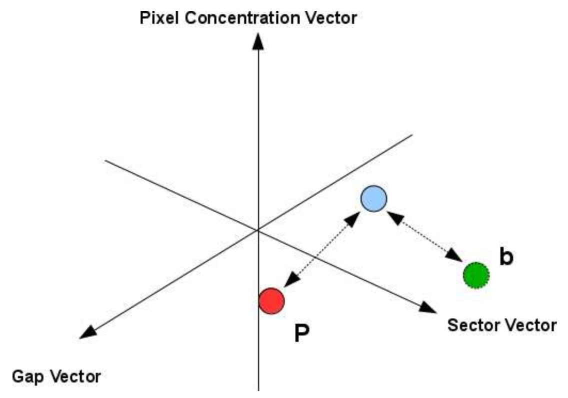


Figure 11: A visualization of the Character Recognition process for an input image with only one character. The blue dot is the Character Model representation of the input image and the red and green dots are two members of the GCDD that are "close" to it.

GCDD that have the closest Character Models to the ones of the input image and are also members of an English dictionary. This is done by generating all the possible words than can be formed from the top three matches for each character and choosing the overall best matching word. Even a simple method such as this has been shown to improve OCR accuracy. [6] In addition, a simple grammar reference can be used to improve accuracy. The Idocrase OCR system handles proper usage of punctation and capitalization.

4 Testing and Results

4.1 Testing Programs

Throughout the development of the system, various programs were developed to help analyze intermediate results in order to further the progress of the project. The most important one of these analyzed the working version of the GCDD to determine the following:

1. The deviation between a Character Model used to generate the GCDD and the contents of the GCDD, for each character in the GCDD.
2. How "close" members of the GCDD are to each other.
3. For each member of the GCDD, the actual characters who share that member.

Ideally, the value of the first relationship would be relatively small, and the value of the second relationship would be relatively large, and the value of the third relationship would be relatively large. This would mean that character definitions are more discrete; the results are more "spread out."

4.2 Results

An automated testing system was developed to determine the accuracy of the Idocrase system with respect to font and font size. Using Java's Graphics class, images of selected words of four different founts were generated and run through the system. The accuracy of the system was tested as the percentage of letters read correctly.

Table 1: System Accuracy Results

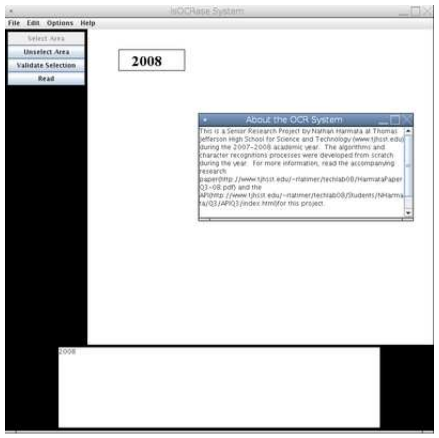
Font	Size			
	18	20	24	28
Dialog	68.9	91.5	95	90.4
Serif	93.2	100	89.6	100
Times New Roman	97.2	100	97.3	95
Courier	100	95	96.4	89

The system had a tested overall accuracy of 93.7 percent. The tested accuracy for reading words, however, is much lower, at around 50 percent. Several explanations for this are offered in the Appendix.

4.3 Resources

In addition to the ones already described, the following computer languages, algorithms and programs are used.

1. The Idocrase OCR system is written entirely in Java.



This is a Senior Research Project by Nathan Harmata at Thomas Jefferson High School for Science and Technology (www.tjhsst.edu) during the 2007-2008 academic year. The algorithms and character recognitions processes were developed from scratch during the year. For more information, read the accompanying research paper(http://www.tjhsst.edu/~rlatimer/techlab08/HarmataPaperQ3-08.pdf) and the API(http://www.tjhsst.edu/~rlatimer/techlab08/Students/NHarmata/Q3/APIQ3/index.html)for this project.

Figure 12: The Idocrase Application.

2. Java's ImageIO class is used for picture input and output.

3. Java's BufferedImage class is used to handle pictures.

4. KolourPaint is being used to make picture files for input and to precisely view images for debugging purposes.

5 Idocrase Application

An application was developed to allow users to utilize the Idocrase system. It is contained in a .jar file, so the user's computer needs to have the latest version of Java installed (1.6.0_05). The application allows user to load image files and select portions of them to be "read" by the system. The generated text is placed in a text box in the application, from where it can be copied and pasted into a text editor. The user also has the option of directly saving the output to a text file. The user also can turn off the usage of the dictionary and/or grammar references. If there are significant errors in the results, the user can view the debug statements generated by the system.

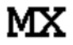


Figure 13: An example of an elision.

6 Conclusions

Further work could be done to improve the current methods for the detection and removal of noise. There are various methods that can be used to accomplish this. [5] In addition, there might be a prelimineray check to ensure that the portion of the image selected is, in fact, text in the proper orientation. [3] The Idocrase system checks to see if the selected image passes basic benchmark texts, but assumes that it actually contains text. There have already been methods developed for the purpose of distinguishes certain types of images from other types [4] and text from non-text. [7]

Overall, the Idocrase system met the goals of this project. It was developed virtually completely from scratch and gives the user some powerful OCR tools. Its accuracy isn't as good as some of the existing OCR systems, but it is still quite impressive considering the scope of this research project.

Appendices

A Unused Ideas

These are some of the more interesting ideas that, through the course of the development of this project, were either discarded or not used at all:

1. The initial iteration of the Idocrase system was made to only work on a specific font. It did not contain Character Model or Attribute structures. Instead, there was only one method of comparison, which was relative pixel coverage of each of the four quadrants in the image. This idea was obviously not very successful, and was just intended as a starting point for the development of the eventual final product.

2. The Character Model structure was actually invented to make it easier to add more methods of comparison (which evolved into the Attribute structure), rather than for general organization purposes. The inital plan was to use Java Reflections to read in the GCDD, assuming that each token was the name of an Attribute and the succeeding tokens were the required arguments for the constructor method for that Attribute. This idea was never implemented, but would make simple to add on to this project.

3. One major problem with the Image Processing step was ignored because there was not enough time to find a good solution to it. Certain letters of certain fonts "elide" when placed next to each other. This occurs rather often with capital letters in the Times New Roman font, most notably. This causes a large obstacle in the Image Processing step because it is assumed that there is whitespace between adjacent characters, meaning that elided characters are treated by the system as one character, which, of course, will not be "read" very accurately. A solution to this problem was formulated, but never implemented due to time constraints. From observation, most elisions occur near the top or the bottom of the boxed image. This, coupled with the fact that boxed images of elided characters are going to be much wider than those of non elided characters, can be used to detect elisions. The offending characters can then be easily separated, since the cause of the elision would be known.

B GCDD

The version of the GCDD included in the Idocrase system was generated from size 20-point fonts. Based on the final system accuracy tests and previous testing during the development of the Idocrase system, it seems like there may be a relationship between the size of the text used to generate the GCDD and the accuracy of the system. There is a slight pattern showing that a GCDD made from smaller font sizes is better overall. There was not enough time to explore this further.

C System Accuracy

The system was tested to read individual characters at 93.7 percent accuracy. This is actually rather good; it is comparable to commercial OCR programs. It is discouraging, then, that the system reads words correctly only half the time. There are several explanations for this.

1. Bearing in mind that this project was developed independent of any existing OCR algorithms, it is possible that the current method of word recognition is simply suboptimal. Maybe it is not correct to find all the combinations of the top matches characters and then choose the best one.

2. Commercial OCR systems probably have extremely advanced grammar engines. These systems would know when to expect nouns as opposed to verbs, etc. Statistical methods could also be used to keep the frequency of certain words intact. It is likely that, if the Idocrase system had a powerful grammar engine, it could greatly improve its accuracy, even with its current method of word recognition. This would mean that once the system generates lists of possible words for each actual word, the problem becomes one of Computational Linguistics, instead of OCR.

3. Data about character recognition accuracy for commercial OCR systems were not available; only word recognition accuracy were. Although Idocrase's 93.7 percent accuracy seems very good, it might be the case that commercial OCR programs have character recognition accuracy close to 100 percent and there are other reasons, perhaps grammatical and/or formatting errors, that cause a lower word recognition accuracy.

D Idocrase Application

D.1 Working with Images

The system currently supports Tiff, Tif, Gif, JPEG, JPG, and PNG images. To load an image, the user can press "File" - "Open" and browse for his or her file. The maximum size for an image is 600x585 pixels. If the user's image is too large, try resizing it or using a portion of it using picture editing software like Microsoft Paint.

Once the user has loaded an image, he or she can select the portion of it to be read by the system. To do so, press the "Select Area" button on the left side of the interface and draw a box around the portion of the image by clicking, dragging, and then releasing the mouse. Be sure to leave at least some space between the box and the text in the image. If the user is unhappy with his or her selection, her or she can press the "Unselect Area" button and try again.

The user can save his or her selection to a file by pressing "File" - "Save Selection." He or she should be sure to give the file an appropriate extension (Tiff, Tif, Gif, JPEG, JPG, or PNG).

Once the user has part of the image boxed and selected, the OCR system needs to perform a preliminary check. Do this by pressing the "Validate" button, which will appear on the left side of the interface after an area has been selected. A possible cause of error is the image not being boxed properly.

D.2 Using the Isocrase System

A validated image can be "read" by the system. Pressing the "Read" button on the left side of the interface will output the best matching text to the text box in the bottom of the interface. From there, the user can either copy and paste it into a text editor or use the export function ("File" - "Export Text") to save it directly in a text file. Be sure to give it the appropriate ".txt" file extension.

By default, dictionary and grammar references are used. With the dictionary reference, the system will only choose English words as matching text. The user can turn either of these off by pressing "Options." With both references not in use, the text output will be the best match character-for-character, which may or may not be of use to the user.

References

[1] Austin Acton. A review of free optical character recognition software, 2007.

[2] Maher Ahmed and Rabab Kreidieh Ward. An expert system for genral symbol recognition, 1998.

[3] Hrishikesh B. Aradhye. A general method for determining up/down orientation of text in roman and non-roman scripts, 2004.

[4] Subhaijt Sanyal and S. H. Srinivasan. A system for detecting and matching logos in natural images, 2007.

[5] Faisal Shafait, Joost van Beusekom, Daniel Keysers1, and Thomas M. Breuel. Page frame detection for marginal noise removal from scanned documents, 2007.

[6] Kazem Taghva, Julie Borsack, and Allen Condit. An expert system for automatically correcting ocr output, 1994.

[7] Victor Wu, R. Manmatha, and Edward M.Riseman. Finding text in images, 1997.