April 2012

Optical Character Recognition

Dhia elhak Ben Haddej
Worcester Polytechnic Institute

Sean Alexander O'Brien
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/mqp-all

Repository Citation

This Unrestricted is brought to you for free and open access by the Major Qualifying Projects at Digital WPI. It has been accepted for inclusion in Major Qualifying Projects (All Years) by an authorized administrator of Digital WPI. For more information, please contact digitalwpi@wpi.edu.
Optical Character Recognition

A Major Qualifying Project Report
submitted to the faculty of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science by

___________________________________
Sean O’Brien

___________________________________
Dhia Ben Haddej

April 26, 2012

___________________________________
Professor Gábor N. Sárközy, Major Advisor

___________________________________
Professor Stanley M. Selkow, Co-Advisor
Abstract

Our project aimed to understand, utilize and improve the open source Optical Character Recognizer (OCR) software, OCRopus, to better handle some of the more complex recognition issues such as unique language alphabets and special characters such as mathematical symbols. We extended the functionality of OCRopus to work with any language by creating support for UTF-8 character encoding. We also created a character and language model for the Hungarian language. This will allow other users of the software to preform character recognition on Hungarian input without having to train a completely new character model.
Acknowledgments

- András Kornai, Project Advisor and SZTAKI Contact
- Attila Zséder, SZTAKI Colleague
- Gábor Sárközy, MQP Advisor
- Stanley Selkow, MQP Co-Advisor
- Tom Breuel, OCRopus Developer
- Worcester Polytechnic Institute
- MTA-SZTAKI
  - Information Lab, MTA-SZTAKI
Contents

Chapter 1: Background.................................................................................................. 3
  1.1 Introduction........................................................................................................ 3
  1.2 History of OCR.................................................................................................. 3
  1.2.1 Template-Matching Method........................................................................ 4
  1.2.2 Peephole Method.......................................................................................... 6
  1.2.3 Structured Analysis Method......................................................................... 7
  1.2.4 Factors influencing OCR software performance ........................................ 8
  1.3 Independent Component Analysis .................................................................. 10
  1.4 Energy-based Models for sparse overcomplete representations .................. 16
  1.5 Finite State Transducers in Language and Speech Processing ...................... 17
    1.5.1 Sequential Transducers............................................................................ 18
    1.5.2 Weighted Finite State Transducers......................................................... 18
    1.5.3 Transducers in Language Modeling ....................................................... 20
  1.7 Image File Formats .......................................................................................... 21
    1.7.1 TIFF .......................................................................................................... 21
    1.7.2 PDF ........................................................................................................... 21
    1.7.3 PNG ........................................................................................................... 22
    1.7.4 JPEG .......................................................................................................... 22
  1.8 OCRopus File Formats ...................................................................................... 22
    1.8.1 Physical Layout......................................................................................... 22
    1.8.2 Page/Line/Character Segmentation File.................................................. 22
    1.8.3 Hypothesis Graph File................................................................................ 22
    1.8.4 Lattice File .................................................................................................. 23
    1.8.4 Hierarchical Database File.......................................................................... 23
Chapter 2: OCRopus .................................................................................................... 24
  2.1 Character Modeling............................................................................................ 24
    2.1.2 Character Model Training.......................................................................... 31
  2.2 Language Modeling............................................................................................ 32
    2.2.1 Language Model Implementation ............................................................. 33
    2.2.2 Mathematical Text..................................................................................... 38
  2.3 Using the OCRopus Software .......................................................................... 39
    2.3.1 Installation................................................................................................. 40
2.3.2 Running the OCRopus Pipeline: .................................................................41
2.4 Assessing OCR Accuracy .................................................................46
3 Conclusions .................................................................................................48
  3.1 Results ....................................................................................................48
  3.2 Conclusions on OCRopus........................................................................48
  3.3 Future Work ..........................................................................................49
References ......................................................................................................50
**Table of Figures**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Statistical Machine Design by Paul W. Handel</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Illustration of 2-D reduction to 1-D by a slit. (a) An input numeral “4” and a slit scanned from left to right. (b) Black area projected onto $x$ axis, the scanning direction of the slit.</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Illustration of the peephole method</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>The Solartron Electronic Reading Automaton</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>Extension of the peephole method to structure analysis.</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>The original signals.</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>The observed mixture of the source signals in Fig. 6.</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>The multivariate distribution of two independent gaussian variables.</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>Approach diagram of Linear Component Analysis</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>Simple Finite State Machine</td>
<td>19</td>
</tr>
<tr>
<td>11</td>
<td>Example of transducer composition</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>Raw scan of a page in an Algebra textbook written by László Fuchs</td>
<td>25</td>
</tr>
<tr>
<td>13</td>
<td>Binarized image from Figure 12.</td>
<td>26</td>
</tr>
<tr>
<td>14</td>
<td>Visual Representation of Raw Database (Example Character Z)</td>
<td>28</td>
</tr>
<tr>
<td>15</td>
<td>Visual Representation of Database After Clustering (Example Character Z)</td>
<td>29</td>
</tr>
<tr>
<td>16</td>
<td>Example of recognition error</td>
<td>30</td>
</tr>
<tr>
<td>17</td>
<td>Example of character recognition improvement between the English and Hungarian character models.</td>
<td>31</td>
</tr>
<tr>
<td>18</td>
<td>Dictionary Generation Code</td>
<td>35</td>
</tr>
<tr>
<td>19</td>
<td>Cost Dictionary for the Hungarian Language</td>
<td>35</td>
</tr>
</tbody>
</table>
Figure 20: Sample portion of an English language model FST. .......................................................... 36
Figure 21: English language model output on page of Hungarian text ........................................ 37
Figure 22: Hungarian language model output on page of Hungarian text................................. 38
Chapter 1: Background

1.1 Introduction

We are moving forward to a more digitized world. Computer and PDA screens are replacing the traditional books and newspapers. Also the large amount of paper archives which requires maintenance as paper decays over time lead to the idea of digitizing them instead of simply scanning them. This requires recognition software that is capable in an ideal version of reading as well as humans. Such OCR software is also needed for reading bank checks and postal addresses. Automating these two tasks can save many hours of human work.

These two major trends lead OCR software to be developed and licensed to OCR contractors. “There is one notable exception to this, which is OCRopus open source OCR software that Google is helping to develop” [3].

OCRopus was created by Professor Tom Breuel from the DFKI (German Research Center for Artificial Intelligence at Kaiserslautern, Germany). Google sponsored the project on April 09, 2007 with the goal of providing an open source OCR system capable of performing multiple digitization functions. The application of this software ranged from general desktop use and simple document conversion to historical document analysis and reading aids for visually impaired users.

1.2 History of OCR

The idea of OCR technology has been around for a long time and even predates electronic computers.
This is an image of the original OCR design proposed by Paul W. Handel in 1931. He applied for a patent for a device “in which successive comparisons are made between a character and a character image.” [5]. A photo-electric apparatus would be used to respond to a coincidence of a character and an image. This means you would shine a light through a filter and, if the light matches up with the correct character of the filter, enough light will come back through the filter and trigger some acceptance mechanism for the corresponding character. This was the first documented vision of this type of technology. The world has come a long way since this prototype.

1.2.1 Template-Matching Method
In 1956, Kelner and Glauberman used magnetic shift registers to project two-dimensional information. The reason for this is to reduce the complexity and make it easier to interpret the information. A printed input character on paper is scanned by a photodetector through a slit. The reflected light on the input paper allows the photodetector to segment the character by calculating the proportion of the black portion within the slit. This proportion value is sent to a register which converts the analog values to digital values. These samples would then be matched to a template by taking the total sum of the differences between each sampled value and the corresponding template value. While this machine was not commercialized, it gives us important insight into the dimensionality of characters. In essence, characters are two-dimensional, and if we want to reduce the dimension to one, we must change the shape of the character for the machine to recognize it.

Figure 2: Illustration of 2-D reduction to 1-D by a slit. (a) An input numeral “4” and a slit scanned from left to right. (b) Black area projected onto x axis, the scanning direction of the slit.
1.2.2 Peephole Method

This is the simplest logical template matching method. Pixels from different zones of the binarized character are matched to template characters. An example would be in the letter A, where a pixel would be selected from the white hole in the center, the black section of the stem, and then some others outside of the letter.

![Illustration of the peephole method.](image)

Each template character would have its own mapping of these zones that could be matched with the character that needs to be recognized. The peephole method was first executed with a program called Electronic Reading Automation in 1957.
Figure 4: The Solartron Electronic Reading Automaton

This was produced by Solartron Electronics Groups Ltd. and was used on numbers printed from a cash register. It could read 120 characters per second, which was quite fast for its time, and used 100 peepholes to distinguish characters.

1.2.3 Structured Analysis Method

It is very difficult to create a template for handwritten characters. The variations would be too large to have an accurate or functional template. This is where the structure analysis method came into play. This method analyzes the character as a structure that can be broken down into parts. The features of these parts and the relationship between them are then observed to determine the correct character. The issue with this method is how to choose these features and relationships to properly identify all of the different possible characters.

If the peephole method is extended to the structured analysis method, peepholes can be viewed on a larger scale. Instead of single pixels, we can now look at a slit or ‘stroke’ of pixels and determine their relationship with other slits.
This technique was first proposed in 1954 with William S. Rohland’s “Character Sensing System” patent using a single vertical scan. The features of the slits are the number of black regions present in each slit. This is called the cross counting technique.

1.2.4 Factors influencing OCR software performance

OCR results are mainly attributed to the OCR recognizer software, but there are other factors that can have a considerable impact on the results. The simplest of these factors can be the scanning technique and parameters.

The table below summarizes these factors and provides recommendations for OCR scanning on historic newspapers and other old documents.
<table>
<thead>
<tr>
<th>Process Steps</th>
<th>Factors influencing OCR</th>
<th>Recommended actions for historic newspapers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obtain original source</td>
<td>Quality of original source</td>
<td>• Use original hard copies if budget allows (digitization costs will be considerably higher than for using microfilm)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Hard copies used for microfilming/digitization should be the most complete and cleanest version possible</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Use microfilm created after establishment and use of microfilm imaging standards (1990’s or later)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Use master negative microfilm only (first generation) or original copies, no second generation copies.</td>
</tr>
<tr>
<td>Scan file</td>
<td>Scanning resolution and file format</td>
<td>• Scanning resolution should be 300 dpi or above to capture as much image information as possible</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• File format to be lossless e.g. TIFF so that no image information (pixels) are lost.</td>
</tr>
<tr>
<td>Create good contrast between black and white in the file (Image preprocessing)</td>
<td>Bit depth of image, Image optimization and binarization process, Quality of source (density of microfilm)</td>
<td>• Scan the image as grayscale or bi-tonal.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Image optimization for OCR to increase contrast and density needs to be carried out prior to OCR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>either in the scanning software or a customized program.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If the images are grayscale, convert them to image optimized bi-tonal (binarization).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Obtain best source quality.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Check density of microfilm before scanning.</td>
</tr>
<tr>
<td>OCR software - Layout of page analyzed and broken down</td>
<td>Skewed pages, Pages with complex layouts, Adequate white space between lines, columns and at edge of page</td>
<td>• De-skew pages in the image preprocessing step so that word lines are horizontal.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Layout of pages and white space cannot be changed, work with what you have.</td>
</tr>
</tbody>
</table>
| OCR software - Analyzing stroke edge of each character | **so that text boundaries can be identified** | **Optimize image for OCR so that character edges are smoothed, rounded, sharpened, contrast increased prior to OCR.**  
**Obtain best source possible (marked, mouldy, faded source, characters not in sharp focus or skewed on page negatively affects identification of characters).** |
| OCR software - Matching character edges to pattern images and making decision on what the character is | • Image optimization  
• Quality of source | Select good OCR software.  
Select good OCR software. |
| OCR software – Matching whole words to dictionary and making decisions on confidence | • Pattern image in OCR software database  
• Algorithms in OCR software | Select good OCR software.  
Select good OCR software. |
| Train OCR engine | • Depends on how much time you have available to train OCR | • Purchase OCR software that has this ability.  
• At present it is questionable if training is viable for large scale historic newspaper projects |

Table 1: Potential methods of improving OCR accuracy.

1.3 Independent Component Analysis

This is a method that was developed with the goal of finding a linear representation of nongaussian data so that the components are statistically independent. Data is nongaussian if it does not follow a normal distribution. The cocktail party problem is a great example of the need for a way to analyze mixed data. In this problem, there are two signal sources, two people speaking at the same time, and two sources, microphones, to collect this data. We would like to
be able to take the mixed data of the two speakers collected from these two microphones and somehow separate the data back to their original signals. Each microphone will have a different representation of the mixed signal because they will be located in different positions in the room. If we represent these mixed recorded signals as $s_1(t)$ and $s_2(t)$ we could express this as a linear equation:

$$x_1(t) = a_{11}s_1 + a_{12}s_2$$
$$x_2(t) = a_{21}s_1 + a_{22}s_2$$

where $a_{11}, a_{12}, a_{21}, a_{22}$ are parameters that depend on the distances of the microphones from the speakers [1]. This gives us the nongaussian data we need to properly analyze these signals in an effort to realize the original signals.

![Figure 6: The original signals.](image-url)
In order to properly execute Independent Component Analysis the data must go through some initial standardization along with one fundamental condition: nongaussianity. To show why Gaussian variables make ICA impossible, we assume we have an orthogonal mixing matrix and our sources are all gaussian. Then $x_1$ and $x_2$ are gaussian, uncorrelated, and of unit variance. The expression for their joint density will be:

$$p(x_1, x_2) = \frac{1}{2\pi} \exp \left( - \frac{x_1^2 + x_2^2}{2} \right)$$

The distribution for this equation is shown in the following figure.
Figure 8: The multivariate distribution of two independent gaussian variables.

The density of this distribution is completely symmetric and does not contain any relevant information about directions of the columns of the mixing matrix. Because there is no relevant information, we have no way to make estimates about this data \[1\]. We thus need a measure of nongaussianity, this can be done using kurtosis or negentropy.

Kurtosis is the older method of measuring nongaussianity and can be defined for \( y \) as:

\[
kurt(y) = E(y^4) - 3(E(y^2))^2
\]

This simplifies to \( E(y^4) - 3 \) because \( y \) is of unit variance and can be interpreted as the normalized fourth moment \( E(y^4) \). Kurtosis is usually either positive or negative for nongaussian random variables. If kurtosis is zero, then the random variable is Gaussian. For this reason we generally take the absolute value or the square of kurtosis as a measure of gaussianity.

The use of kurtosis has been commonly used in ICA because of its simple formulation and its low computational cost. The computation cost is in fact reduced when using the fourth moment of the data as estimation for its kurtosis. This is due to the following linear properties:
\[ kurt(x_1 + x_2) = kurt(x_1) + kurt(x_2) \]
\[ kurt(\alpha x_1) = \alpha^4 kurt(x_1) \]

Although kurtosis proved to be very handy for multiple applications, it did have one major weakness; its sensitivity to outliers. This means that when using a sample data in which the distribution is either random or has some errors, kurtosis can fail at determining its gaussianity. This lead to the development of another method called negentropy.

As the name suggests negentropy is based on entropy measure which is a fundamental concept of information theory. Entropy describes the amount of information that can be taken out of the observation of a given variable. A large entropy value means the data is random and unpredictable.

For a discrete random variable \( Y \), its entropy is expressed as follow:
\[
H(Y) = - \sum_i P(Y = a_i)logP(Y = a_i)
\]

In a similar manner the entropy of a continuous random variable \( y \) can be expressed as:
\[
H(y) = - \int f(y)logf(y)dy
\]

Information theory established that out of all random variables of equal variance, the Gaussian variable will have the highest entropy value which can also be attributed to the fact that Gaussian distribution is the most random distribution [1].

The precedent result shows that we can obtain a measure of gaussianity through differential entropy which is called negentropy.

For a variable \( y \) we define its negentropy as:
\[
J(y) = H(y_{gauss}) - H(y)
\]

where \( y_{gauss} \) a Gaussian random variable that has the same covariance matrix as the variable \( y \).
Negentropy is zero if and only if y has a Gaussian distribution, thus the higher its measure the less Gaussian the variable is. Unlike kurtosis, negentropy is computationally expensive. A solution to this problem is to find simpler approximations of its measure. The classical approximation of negentropy was developed by in 1987 by Jones and Sibson as follows:

\[ J(y) \approx \frac{1}{12} E[y^3]^2 + \frac{1}{48} kurt(y)^2 \]

with the assumption that y has zero mean and unit variance.

A more robust approximation developed by Hyvärinen makes use of nonquadratic functions as follows:

\[ J(y) \approx \sum_{i=1}^{p} k_i [E[G_i(y)] - E[G_i(v)]]^2 \]

where \( k_i \) some positive constans, \( v \) the normalized Gaussian variable and \( G_i \) some non quadratic functions.

A common use of this approximation is to take only one quadratic function \( G \), usually

\[ G_1(u) = \frac{1}{a_1} \log \cosh a_1 u, \quad G_2(u) = - \exp \left( - \frac{u^2}{2} \right) \]

and the approximation will then be in the form:

\[ J \propto [E[G(y)] - E[G(v)]]^2 \]

We then have obtained approximations that provide computational simplicity comparable to the kurtosis measure along with the robustness of negentropy.

To give a brief explanation on why gaussianity is strictly not allowed we can say that it makes the data completely symmetric and thus the mixing matrix will not provide any information on the direction of its columns.
As mentioned above, data preprocessing is crucial in that it makes the ICA estimation simpler and better conditioned. Many preprocessing techniques can then be applied such as “Centering” that consists in subtracting the mean vector of $x$

$$m = E[x]$$

so as to make $x$ a zero-mean variable and “Whitening” which is the linear transformation of the observed vector $x$ so that its components become uncorrelated and its variances equal unity, this vector is then said to be white.

### 1.4 Energy-based Models for sparse overcomplete representations

Initially there were two approaches to Linear Components Analysis: The Density Modeling Approach and the Filtering approach. Density Modeling is based on causal generative models whereas the Filtering approach uses information maximization techniques. Energy based models emerged as a unification of these methods because it used Density Modeling techniques along with filtering techniques [7].
Energy based models associate an energy to configuration of relevant variables in graphical models, this is a powerful tool as it eliminates the need for proper normalization of the probability distributions. “The parameters of an energy-based model specify a deterministic mapping from an observation vector $x$ to a feature vector and the feature vector determines a global energy, $E(x)$” [7]. Note that the probability density function of $x$ is expressed as:

$$p(x) = \frac{e^{-E(x)}}{Z}$$

where $Z$ is a normalization vector.

### 1.5 Finite State Transducers in Language and Speech Processing

Finite State Machines are used in many areas of computational linguistics because of their convenience and efficiency. They do a great job at describing the important local phenomena
encountered in empirical language study. They tend to give a good compact representation of lexical rules, idioms, and clichés within a specific language.

For computational linguistics, we are mainly concerned with time and space efficiency. We achieve time efficiency through the use of a deterministic machine. The output of a deterministic machine is usually linearly dependent on the size of the input. This fact alone allows us to consider it optimal for time efficiency. We are able to achieve space efficiency with classical minimization algorithms for deterministic automata.

1.5.1 Sequential Transducers

This is an extension of the idea of deterministic automata with deterministic input. This type of transducer is able to produce output strings or weights in addition to deterministically accepting input. This quality is very useful and supports very efficient programs.

1.5.2 Weighted Finite State Transducers

The use of Finite state automata contributed a lot to the development of speech recognition and of natural language processing. Such an automaton provides a state transition depending on the input it receives until it reaches one of the final states; the output state.
Nowadays in natural language processing the use of another type of finite state machines has become widely spread, these machines are the Transducers.

These transducers keep all the functionality of a simple FSM (finite state machine) but add a weight to each transition. In speech recognition for example this weight is the probability for each state transition. In addition, in these transducers the input or output label of a transducer transition can be null. Such a null means that no symbol needs to be consumed or output during the transition. These null labels are needed to create variable length input and output strings. They also provide a good way of delaying the output via an inner loop for example.

Composition is a common operation in the use of transducers. It provides a way of combining different levels of representation. A common application of this in speech recognition is the composition of a pronunciation lexicon with a word-level grammar to produce a phone-to-word transducer whose word sequences are restricted to the grammar [8].
1.5.3 Transducers in Language Modeling

Initial approaches to language modeling used affix dictionaries to represent natural languages. This method came in handy to represent languages like English by having a list of the most common words along with possible affixes. However, when trying to represent more languages, it was quickly clear that such an approach fails with agglutinative languages.

An agglutinative language is a language in which word roots change internally to form other nouns. Unlike the English language in which we generally add suffixes to obtain other word forms like the suffix –ly for adverbs. Hungarian falls under the agglutinative languages for which we needed to create a dictionary and a language model in FST (finite state transducer) format. The representation of such a language can be done by “having the last node of the
portion of the FST, which encodes a given suffix, contain outgoing arcs to the first states of portions of the FST which encode other suffixes” [10]. The advantage of this technique is that when applied to all the possible affixes, it will then have a solid representation of the agglutination nature of the language.

1.7 Image File Formats

There are many different file formatting options available for character recognition software. We primarily dealt with PNG files because it was the only usable format in OCRopus but we were faced with some challenges during image conversion. Image quality has a huge impact on the effectiveness of any OCR software and when trying to change between formats, one has to be aware of lossy vs. lossless compression. These were the formats we ran into during this project:

1.7.1 TIFF

This is a Tagged Image File Format and can be used as a single or multi image file format (multiple pages in the same file). The TIFF format is very desirable because the most common compression schemes are all lossless. This means that these types of compression can reduce the file size (and later returned to their original size) without losing any quality.

1.7.2 PDF

Personal Document Format is currently an open source standard created by Adobe. While the ability for a PDF to contain text and images is very useful for some applications, this is an unnecessarily, robust quality that only adds to the file size. A TIFF is much more desirable because it is can specifically only contain images.
1.7.3 PNG

Portable Network Graphic formatting is a lossless data format and the one that is used by OCRopus. They are a single image, open, color image format and were created to replace the GIF image format, which only supported a maximum of 256 colors.

1.7.4 JPEG

The acronym ‘JPEG’ comes from the founding company of the file format, Joint Photographic Experts Group. This is a lossy image format but can be scaled to tradeoff between storage size and image quality. This is not ideal for OCR software, but can be used as long as the data is never compressed.

1.8 OCRopus File Formats

1.8.1 Physical Layout

This format is intended to represent the division of a document page into special areas, columns, paragraphs, and text lines.

1.8.2 Page/Line/Character Segmentation File

These file types represent the intended document for analysis broken down into smaller parts. Each character segmentation file contains an entire line of text, but each character within the line is represented with a different RGB value. In a very long line of text, it is possible to see the spectrum of color ranging from black to dark blue.

1.8.3 Hypothesis Graph File
This is a file format for OpenFST, a library for constructing, combining, optimizing, and searching weighted finite-state transducers.

1.8.4 Lattice File

The lattice file format is used for the recognition output and contains two types of lines: Segment lines and Character lines. The character lines contain recognition results from the preceding segment line.

1.8.4 Hierarchical Database File

This is a database file format designed to store large amounts of numerical data. In OCRopus, it is also used to store the PNG images for each character mapped to a character class and a set of probabilities.
Chapter 2: OCRopus

The project was initially expected to run for three years as a support for three Ph.D. students but was later released as software under the Apache license. This means that the project is now open source and free to use with the preservation of the copyright notice and disclaimer. The major advance in the program’s development was the incorporation of the Tesseract character recognizer along with growing language modeling tools. The last operational version is OCRopus 0.4.4 (alpha). This was very successful because of its adaptive language model training. This feature allowed for the user to create their own language and character models by inputting their own data. Currently the system is being revised and OCRopus v0.5 is an attempt to reorganize all the open source additions. The creators are in the process of consolidating and standardizing the code to be able to add more complex functionality and to attain a recognition level comparable to the professional software. They have also implemented their own character recognizer to remove their dependency on the Tesseract software.

2.1 Character Modeling

If we want to use OCR software on any document, we would like to be able to recognize the fundamental, smallest components. Interpreting individual characters is the basis of Independent Component Analysis and Optical Character Recognition. In the life sciences or even computer science, we strive to understand cells or bits in their most basic form to learn more about the overall structure. If we have a way to interpret the smallest elements of document images, we can piece these together and make inferences about the text as a whole.
The best way to do this is to compare each character to a template character representing a letter or symbol found in a given language. There are some pre-existing templates that are used for OCR, but this is not ideal. A generic template would only be successful for a specific font and character type. Depending on the language of the text, a generic template might also not include all of the characters of this language. If we want to recognize handwriting, this adds another level of complexity, and makes such a template obsolete. We would ideally like to have a template that is close enough to the text we are trying to interpret, but with the ability to morph and adapt the template to match our target document.

OCRopus has a built in character recognizer which will do this. We needed to first preprocess the images to reduce image errors (unclear/useless image noise) and binarize (change all pixels to either black or white) the pages.

Figure 12: Raw scan of a page in an Algebra textbook written by László Fuchs
When we input a set of preprocessed page images, OCRopus will segment the lines of text, and give a best path text interpretation based on lattices generated by the character recognizer. These lattice files contain probability weights for each individual character in a text line. These probabilities represent how close the letters are to each character template. A best path interpretation of the text line means we will get text files of each line of the document where we select the highest probability template matches of the characters. If the quality of the document scan is very good, these text files can contain very accurate representations of the actual text, but achieving one hundred percent recognition at this step is unrealistic. To increase our accuracy
with this step, we need to combine the best path with a language model interpretation, which we will elaborate on later in this report.

These templates are known as character models. The two that exist for the OCRopus package are very good, but only have templates of English characters. For our project we are testing this software with a Hungarian-English dictionary and a Hungarian mathematical textbook. Between both of these, we would like to have a character model that can represent Hungarian, English and mathematical symbols such as Greek letters. Fortunately, with the OCRopus software, we have the capability to create, adapt, and train our own character model.

2.1.1 Character Model Implementation

In OCRopus, we created a completely new character model to represent the Hungarian language. While we did not work with scripts like Cyrillic or Chinese calligraphy that share no similarities with the English language, we still had to change the character model to work with Hungarian text. The built-in English character models could not handle accented characters because they are only found in words borrowed from other languages. To start, we ran through the steps to digitize a Hungarian-English dictionary with the English character model and adapted the results to handle Hungarian.

At a certain point in the character recognition process (described in detail in Section 2.3.2) we had a large collection of segmented character images. These were images of each character in the document and, after comparing these to the English character model, were placed into an h5 database file. This database file also contained a character mapped to each image by the program after comparing them to the template characters within the character model. For the dictionary, we were working with over 12000 characters and it would have been unreasonable to
manually check each character for errors. At this point we clustered the database file. Clustering is a process that groups characters that are extremely similar and picks one to represent the entire collection. The other characters are then removed from the database and we are left with a clean and concise representation of our characters. This step reduced the size of our database by 90% and left us with a manageable amount of data. Once we had this clustered database, we were able to open the file in a program called Iledit. This is a built-in part of OCRopus that gives a visual GUI for editing an h5 character database.

![Visual Representation of Raw Database (Example Character Z)](image)

Figure 14: Visual Representation of Raw Database (Example Character Z)
We used this to find any errors in the automatic character assignments. Most of the Hungarian characters were listed as their non-accented equivalents. It was simple enough to find and change these images to their correct character. This also had the functionality to sort images by confidence. This means we were able to order them by how accurate the program thought it was at determining the characters. With this feature, it was easy to find incorrectly classified images by sorting by reverse confidence.

The software as it stood was not capable of receiving input of Hungarian characters into Iledit. We edited the code to allow for this type of input. Having a Hungarian keyboard or the ability to change the keyboard layout to a different region streamlined the input of these new characters. We used the accessibility options on-screen keyboard and switched to the Hungarian keyboard to do this.

Figure 15: Visual Representation of Database After Clustering (Example Character Z)
The most common character mix-up was with the Hungarian ‘ö’ and ‘é’ letters. These would be interpreted as the number ‘6’.

![Figure 16: Example of recognition error.](image)

Because this error was so consistent, it was much easier to fix. The lledit interface allows the user to select multiple characters at once and do a group assignment. There is also a function that allows the user to select a single character and find all characters that are visually similar to this one. This also allows for manual clustering, but is unnecessary given the other clustering functions within OCRopus.

Once the database was completely edited we ran it though a local linear nearest neighbor classification. This created a new database file from the original unclustered and the new clustered database. The characters in the clustered database acted as prototypes for the original database. That means any characters that were listed in the prototypes were considered for the character model in this step. The local linear nearest neighbor classification function compared
the prototypes with the images in the original database file and picked a sample of images that would best represent the new character model.

The next step was to create the actual character model file from the database file. We used OCRopus’ dump command which would reorder the database into a binary file useable by the software.

2.1.2 Character Model Training

Training a character model is the process of running the model through many documents to increase the robustness of the recognition process. If the character model produced by OCRopus was not satisfactory the first time through a document, it is possible to continue to cycle this model through the same or many different documents.

Figure 17: Example of character recognition improvement between the English and Hungarian character models.
This quality of OCRopus is very unique and useful. If the user had a specific font they knew they would continue to recognize, they would be able to train a model specifically for this font or typeset.

We ran our character model through the same document to improve the quality of recognition on future passes. We were also working with two old printed books with an archaic font type. These were most likely produced with a typewriter and called for this type of training. Training the character model literally increased the file size of the character model and allowed the recognition step of OCRopus to work with better data. The templates in the new character model were much accurate and increased our rate of individual character recognition. While this accuracy is very important, we only needed so much before the language model could handle the rest.

2.2 Language Modeling

In natural language processing, language models are probabilistic models whose main goal is to assign a probability to a word or a sentence. In speech recognition for example, they are used to predict the next word given the previous ones or revise an entire sentence and compare its likelihood against another similar sounding sentence. For OCR systems, language models are used in a similar fashion to determine the probability of a generated word to occur in a sentence. This can be done in different ways.

One way is the grammar approach, which defines the possible links and relations between words depending on their position in a sentence. This can help determine whether we are expecting the next word to be a verb, adjective, noun, conjunction, etc. and proceed to reject a word that cannot fit. One obstacle facing this approach is long term dependencies in languages.
such as English. For example, we can have sentences like “The building that I painted collapsed” and “The building that I painted yesterday”. In the first sentence we have two verbs in a row. In the second we have “I painted yesterday” which is a very weak sentence and if we do not consider the entire context and the possibility of having long term dependencies, this can lead to a lot of complexity. It is almost impossible to have an exhaustive model of the entire grammar of a language.

Another language modeling method is to use the word frequencies and this can be done with different levels of complexity. The simplest one known as a unigram model is to use a word corpus which is generated from a set of texts in the desired language and then each word is associated its frequency in those texts. This information can then be used to rank the words by their probability of appearing in that language. The size of the word corpus can improve the accuracy but can also increase the ambiguity when it includes extremely rare words. For this reason there exists custom word corpora depending on the size, the topics and the era of the text supports it was generated from.

N-gram models are an extension of unigram models. They simply look at the probability of the occurrence of a word following two or more other words. In practice unigram models proved to be sufficient and there are some trends to combine different language modeling approaches to reach better accuracy.

2.2.1 Language Model Implementation
OCRopus opted for the use of weighted final transducers to create language models. These FSTs provide a lot of advantages such as the possibility of composing language models or concatenating them.

The OCRopus documentation specifies that language modeling is done within the ocrofst collection that provides different tools and scripts for creating language models in FST form. These scripts are written in the python programming language that has a dedicated library for FST handling called “pyopenfst”.

We decided to use and modify these scripts to create a language model for the Hungarian Language. The ocropus-lm-dict2linefst.py script is a unigram fst language model generator that takes in a dictionary with word entries and their corresponding inverted natural logarithm probabilities to produce the corresponding FST. The reason for using the ln() operation is to avoid underflow since the probabilities are in the 0 to 1 range. The first step was to generate such dictionary from a word corpus. We chose to use the “Hungarian Webcorpus” that can be found at http://mokk.bme.hu/resources/webcorpus/ as it is available under a permissive Open Content License. For this we wrote a python script that can create the needed dictionary from any word corpus from the “Hungarian Webcorpus”
This script generated the file hungDict.txt that contained the Hungarian words along with their negative log probabilities as shown in the screen shot below.

Figure 18: Dictionary Generation Code

Figure 19: Cost Dictionary for the Hungarian Language
The next step was to use the “ocropus-lm-dict2linefst.py” script to generate the corresponding FST to this Hungarian dictionary. When trying to run this script we immediately faced the issue of not being able to process all Hungarian characters. The reason for this is that the script was initially written to support the Latin-2 encoding that does not have the capability of representing letters with accents; a necessary feature in the Hungarian Language.

To solve this problem we had to change the script to read the Hungarian dictionary in the UTF-8 encoding and make the appropriate changes in the FST creation pipeline. This lead us to refine the AddString() method in the pyopenfst library which is now able to accept any UF8 encoded character.

![Diagram of an English language model FST](image)

**Figure 20: Sample portion of an English language model FST.**

Finally after creating our language model FST, we introduced it into the OCRopus pipeline.
This picked out errors and incoherencies of the raw output from the character model recognition step.

Figure 21: English language model output on page of Hungarian text

This figure shows the results of the language model step in OCRopus. It is clear that this is not an accurate representation of the original text because this model was not prepared to handle the Hungarian language.
The output of our Hungarian language model shows a significant improvement in accuracy. The mistakes that exist seemed to be related to segmentation errors rather than inconsistencies in our character and language model.

2.2.2 Mathematical Text

A mathematical text can be considered a language on its own. This is because it has a different syntax (equations, inequalities, letters representing variable names etc.) and a set of unique symbols such as the use of Greek letters along with the extensive use of symbols, integrals and sums along with the unique alignment of the text around them.
Our approach in dealing with this issue consisted of creating a language model containing the unique features of mathematical text. As we did for the Hungarian language, we needed to extend our support for the mathematical symbols. Thus the first step has been to train our character model to support the most commonly used symbols.

The software was able to segment and recognize the lines and characters of mathematical text with reasonable accuracy, so it was only a matter of training the database for our character model. Because the shapes of these mathematical characters were so unique, the character recognizer would automatically classify many of these as rejects. This means that, at the point of creating the character model, all of those characters would have been deleted. Within the database we were able to select these characters from the reject class and properly label them as their correct character.

Following the training of our character model, we moved to create a language model that could handle the mathematical syntax. We first created a mathematical corpus and we parsed the mathematical symbols including Greek letters, summation and integral, inequalities and determined their frequencies. Then we created a cost dictionary in a similar fashion to the Hungarian dictionary (discussed in Section 2.2.2) except that we replaced the symbols with their actual Unicode codes. The reason for this is to guarantee a faithful and consistent representation of the symbols as some of them may not be represented correctly in some programs.

2.3 Using the OCRopus Software

The code base for OCRopus has gone through many revisions in the past few years and lacked sufficient updates to their documentation. This particular revision was meant to consolidate many of the OCR steps into simpler, concise functions. In the process, some less useful functions
remained in the code base and the useful ones were not marked as so. This made our initial use of this software challenging. To alleviate some of the challenges with this revision, we created a step-by-step guide starting from the installation of OCRopus and ending with text files, character models, and language models. This also includes some first-hand experience with errors and elapsed time for some of the more CPU intensive commands.

Note: Anything bolded in the following sections is a command that can be run in a Linux terminal.

### 2.3.1 Installation

The installation section was taken from the OCRopus 0.5 Reference Google Doc and is still valid as of April 10, 2012.

(https://docs.google.com/document/d/1RxXeuuYJRhrOkK8zcVpYtTo_zMG9u6Y0t4s-VoSVoPs/edit)

OCRopus is being developed on Ubuntu, and to install it, you need an up-to-date installation of Ubuntu (currently version 11.11).

To install OCRopus, first download the source code (this uses Mercurial sub-repositories to pull over multiple other repositories):

```
$ hg clone https://code.google.com/p/ocropus
```

You also need to make sure that all the prerequisites are installed:

```
$ sudo sh ./ocroinst packages
```

Next, compile (you don’t need to use sudo with this, it will ask for it when needed):

```
$ sh ./ocroinst install
```

Finally, you also need to download the model files:
$ sudo sh /ocroinst dl

To check for correct installation, just run the top-level OCRopus command:

$ ocropus ocropy/tests/testpage.png

Update your LD_LIBRARY_PATH to include /usr/local/lib with the following command:

$ export LD_LIBRARY_PATH=/usr/local/lib:/usr/lib:/lib

2.3.2 Running the OCRopus Pipeline:

1. Convert your document into a collection of .png files.

   We used the imagemagick `convert` function that comes standard with many Linux OS’s

2. Use the `ocropus-preproc` command on every .png file.

   This will do all of the preprocessing steps to your PNG files (whitening, alignment, binarization, etc).

   This can be accomplished with a consistent numbering system of your .PNG files and the use of ‘?’ character for the incrementing digits.

   Example:

   Collection of files with names: test1.png test2.png test3.png test4.png

   These 4 files would be represented with test?.png

   Sample command:

   $ ocropus-preproc -o sampledir/ testdir/test?.png

   The `-o` flag is used to designate the output location and requires a directory that does not already exist (it creates it for you).

   NOTE: the use of the `-O` flag will output into the same directory as the input files. This can be very messy if the files are not separated and it is not recommended.

   The `-Q` can be used to designate the number of parallel processes you would like to run.
Our experience with this command:

The `ocropus-preproc` command needs quite a bit of RAM to run without memory errors. It also creates a large amount of temporary files while it is running, which can be an issue when working with limited hard drive space. We could not run 4 parallel processes with 6gbs of RAM but were able to with 8gbs of RAM. A colleague was running 1 process on a machine with 2gbs of RAM - so a good rule of thumb would be 2GBs of RAM for every 1 process.

This took about 1 hour to process 3.3GBs of .png files on our machine and produced over 5GBs of additional files.

3. Use the `ocropus-prast` command on this new set of PNG files.

   This will segment the lines of the document into separate PNGs and organize them into numbered directories.

   To continue with the example from the previous step:

   ```
   $ ocropus-prast sampledir/?.png
   ```

   In this example they will be numbered from 1 to 4 (More question marks will be needed for each factor of 10).

   The output will organize the segments into ‘page’ directories (represented by the page number) so that each line of the page will be placed into a directory for the page to which it belongs.

   In our experience this step took approximately 30 minutes and did not give us any problems.

4. Use the `ocropus-lattices` command on each line segment file.

   This will create raw text (raw.txt), raw segment (.rseg) and lattice files corresponding to each line of the page.
$ ocropus-lattices -m ~/../../usr/local/share/ocropus/uw3unlv-240-8-40-nogeo.cmodel ?/??????.png

-\texttt{m} is a necessary flag to give this program the character model. If you downloaded everything correctly in the installation step, this should be the path for the given language model. There were 2 given. This one is more extensive than the other one as it contains a larger collection of English data.

The first ? in the 3rd line represents the organized directory (representing a ‘page’ of the book). The next group of ?’s (6 of them) will be the name of the segmented line. This one will always be represented with 6 characters.

This step took over 30 minutes to run. Our only issue was realizing this function required a character model.

5. Use \texttt{ocropus-lmodel} on the lattice files.

\begin{verbatim}$ ocropus-lmodel ????/??????.lattice\end{verbatim}

This will create text, cseg, and cost files from the lattices and rseg files (the function finds the corresponding rseg files on its own).

6. Use \texttt{ocropus-lalign} in the directory containing the collection on the .txt files

\begin{verbatim}$ ocropus-lalign -x output.h5 -g .txt .\end{verbatim}

This will output an .h5 file containing the segmented characters aligned with their corresponding text file. This means that each cseg will be given the character that was originally determined with the \texttt{ocropus-lattices} command.

The -\texttt{x} flag tells the program to extract the characters and wants you to specify a location (the name of a non-existent .h5 file). We use -\texttt{g} to change the extension of our ground truth text files to .\texttt{txt} because we assume that the information collected from the lattices operation is
accurate or good enough. We eventually train this .h5 file so we do not need the perfect ground truth.

7. Use the llccluster command on the .h5 file from the previous step.

    $ llccluster -o output.h5 input.h5

    The -o flag designates the output database file for our clustering.

    This function will perform per-class clustering of the database. This means that based on the numbers, letters, and symbols (each is represented as a ‘class’) assigned to each image, it will group them and prune the results. It does this by comparing images that are within the same class to each other and grouping them by similarity. Once they are grouped, one from each group is selected to represent the other images in its collection. This does not mean there will only be a single image per class, but rather several images representing the whole class after several groups were created within a single class. This process can vary depending on the size of the database and uniqueness of the characters.

    In our experience this took quite a bit of time (over 20 minutes for 262 pages) and reduced the size of the database by 90%.

8. Use lledit on the clustered database file from the previous step.

    $ lledit db.h5

    This gives the user a visual interface (shown in figures 12 and 13) for editing the classes of the images. There is a dropdown menu that allows the user to select groups of images clustered by class and gives the opportunity to correct any errors.

    Because this is a manual process, it is difficult to determine the length of this step. We were required to input characters that were not part of the US English keyboard layout which
required some additional time. We switched between keyboard settings and used an on-screen accessibility keyboard to facilitate this process.

9. Use the `llnlin` command on the trained database and the previous, un-clustered database.

   ```bash
   $ llnlin -p traineddb.h5 –o output.h5 unclustereddb.h5
   ```

   This will create a new database file with a column representing the local linear classifier model. The actual function name means nearest neighbor linear classifier and computes this model based on a comparison between the original database and the trained one.

   The `-p` flag represents the prototypes of our classes. These are the classes we would like to keep for our character model.

   This function took us about 20 minutes and outputs a database file that is much larger than the original h5 file.

10. Use the `lldump` command on the output database file containing the local linear classifier information.

   ```bash
   $ lldump -c ll -o output.cmodel lindb.h5
   ```

   This will finally produce the character model we have been shaping since the first step.

   The `-c` flag designates the classifier to use for the dumping process. There are a few other classifiers within the software but currently the local linear classifier was the only one giving us positive results.

   This program finishes in a matter of seconds because it essentially just reorganizes the database information into a more accessible file format for OCRopus.

11. Once we have the character model, we can reinsert this into the `ocropus-lattices` step with any document of our choosing.
2.4 Assessing OCR Accuracy

Any OCR system is expected to have as a final output the digitized version of the scanned document. At this level it is important to know how close the digitized copy from the original is. In OCR this is no easy task for a lot of reasons. When dealing with aged documents or historic newspapers the software’s accuracy measure will tell more about the condition of the scanned document than about its own performance. This means that any accuracy measure should at first take into account the state and quality of the original document.

The next issue facing accuracy measurement is how to compare it against the original data since the produced digitized copy may be the only digitized version of the document. One way to deal with this is to type the document manually and then do the comparison. Then the issue is that, other than being lengthy, this process defeats the purpose of the OCR system since typing out a document is a digitization process.

A middle solution would then be to typewrite portions of the scanned document and run the comparison on that portion assuming the document has an overall linear quality level, is composed of a single font and is written in one language. This will then provide us with a good estimation of the overall accuracy level of the OCR system.

The next issue to be addressed is what to look for in the comparison. Should it be character based or word based? This answer to this question is not trivial. This is because it is heavily dependent on the situation. One can argue initially that it is the entire word that matters and not the individual letters and thus say that a word based approach makes more sense. However consider this counter-example in which a sentence that has a missing letter in every word will have a 0% accuracy level in a word based approach. But such a sentence may be
completely legible and understandable since the amount of missing letters is negligible in each word, thus a character based approach will convey a better accuracy in such a situation.
3 Conclusions

3.1 Results

After successfully creating our Hungarian character and language models, we assessed the accuracy of the OCRopus software. We compared the results of our models versus the default English models on a Hungarian algebra textbook written in 1977 by László Fuchs. We were able to successfully recognize Hungarian accented characters and increase the overall accuracy. We used a character based approach to assess the accuracy and increase the rate of correct recognition by 8%. The original accuracy with the English character model was 86% on a sample of 1700 characters and we increased this to 93.5% with our Hungarian character model. We manually calculated the accuracy because the ground truth data for this text did not exist in digital form. From our tests, we have concluded that our character and language model yield significantly better results than the default English models.

3.2 Conclusions on OCRopus

The goal of OCRopus is to provide an accessible, flexible, and simple tool to perform optical character recognition. In its current state, it is not the most user friendly utility and still has many kinks to work out. This is all understandable because it is in an alpha stage of development, and will require some more attention before an official release. The actual theory behind character recognition is in place in the software. OCRopus does an amazing job preprocessing and segmenting images and allows for many fine adjustments to fulfill a variety of user needs. It is now just a matter of reorganizing and optimizing the code to create a user friendly experience.
With time, we believe OCRopus will be one of leading names in optical character recognition software.

### 3.3 Future Work

As we extended the current version of OCRopus for the Hungarian language we got familiar with the types of challenges presented by accented characters and dealt with them successfully. We thus anticipate a future extension of OCRopus to most languages based on the Latin script to be simple based off the current version.

For languages with different alphabets like Chinese and Arabic we think it possible for a future work project to adapt OCRopus to vertical and right to left character recognition since at the language model level, we defined Unicode to be the standard used encoding. This is consistent with the need to represent most written languages in unique encoding for further extensions to other languages. The training portion will then be the key for both the correcting representation and clustering of any new set of characters or alphabets.

As mentioned in section 2.3.2, the OCRopus software is run through multiple commands that represent each step of the recognition process starting from the preprocessing and segmentation and ending with the use of character and language models. We believe it will be very handy and useful to streamline these commands under a single command. This can save a lot of time during future revisions of the software as it is necessary for extensive testing to run it multiple times. Such a command can take in flags for the different operations within the digitization pipeline, and when omitted they will have default values for ease of use.
References


