Document Conversion to LaTeX with OCR

Logan Short, Christopher Wong, and David Zeng

***Abstract*—The end goal of physical document digitalization is the creation of editable representations of documents that contain the information of the original documents. Capturing images of physical documents yields digital document representations; however, these representations are not easily editable and do not store the informational content of the document. In this paper, we propose an end-to-end algorithm for automatic conversion of document images into LaTeX representations of the original document.**

1 Introduction

OPTICAL character recognition (OCR) is the process of converting images of typed or handwritten text into digital characters. In particular, we are considering the problem of using OCR techniques to turn an image of a typesetted document into LaTeX, a markup language commonly used to typeset scientific literature. The motivation for our problem comes from the fact that many old books and academic papers exist online only as scanned images, and to be able to convert these images into formattable LaTeX documents would allow these documents to be much more easily maintained by websites and readers. These documents could also be indexed and searched by search engines, and perhaps in the future we will be able to query academic papers directly with LaTeX markup.

2 Related Work in OCR

There has been much previous work done in various aspects of OCR. In Section 2.1, we summarize a few papers that were influential in the design of our system, and in Section 2.2, we discuss the main contributions of our work.

2.1 Summary of Previous Work

In [1], Jacobs et al. devise a method to perform OCR on low-resolution images. The first part of their work involves building a character recognizer which is achieved using a convolutional neural network architecture. When training and testing, words and characters are scaled such that the vertical extent of the font – including ascenders and descenders – fit within a 29-pixel tall window. Their network has four layers, and their training data includes six different font types with a variety of font styles and sizes. With the character recognizer, they correctly classify about 70% of all symbols (characters, digits, punctuation), which is very respectable given the poor resolution of the images. Also of particular interest is their word recognizer. The word recognizer horizontally scans through the image and invokes the character recognizer at specific locations throughout the word. To find these specific locations, the word is broken up into 2-pixel slices, and various combinations of slices are concatenated together to potentially form characters. This method effectively segments the word into distinct characters even at low resolutions. A dynamic programming algorithm is used to find the optimal combination of slices which then yields the actual character segments.

In [2], Frey and Slate tackle the problem of letter recognition using an adaptive classifier. They train their classifier on 20 different types of fonts while varying the scale, orientation, and distortion of individual character images. The classifier extracts a 16-dimensional feature vector from each character, and these attributes are carefully picked such that no feature vector can map to more than one type of character. Some of the attributes are very straightforward, such as the width and height of the symbol. Others are more complex, such as "the mean horizontal position of all *on* [dark] pixels relative to the center of the box divided by the width of the box." This feature is useful for determining whether an image is "left-heavy." For example, the letter `L` would yield a very negative value. The algorithm uses an adaptive rule-based system to classify characters, and the authors saw very good results.

In [3], Gupta et al. discuss methods for identifying document layout structure from images of documents. The key intuition behind the algorithm proposed in [3] is that the various components of documents, in this case technical papers, can be identified using the features of the bounding boxes encapsulating each section as well as the relative positioning and ordering of said bounding boxes. Gupta et al. first preprocess their document images by thresholding at 80% of the image intensity. Characters in the document are then located by identifying contours in the thresholded image and bounding boxes are formed around these characters. Bounding boxes that are close together are then combined horizontally to yield line level bounding boxes and combined vertically to obtain paragraph level bounding boxes. Following the construction of the paragraph level bounding boxes, features such as aspect ratio and average line height are used to differentiate between textual components and graphical components. At this point the known structure of technical papers is leveraged to classify the previously found bounding boxes as various document components. An image representation is generated in which the previously found bounding boxes are indicated by filled white rectangles while the rest of the image is black. Horizontal and vertical image profile histograms of this image are then used to classify bounding boxes as containing the document title, authors, abstract, main body, and other layout components using known qualities of technical papers. For example, the title is a textual component located at the top of the document, has a high aspect ratio and is immediately followed by the authors field. Once the bounding boxes are all classified as layout components of the document, the document is classified as one of several common technical

paper formats, e.g. IEEE or LNCS. The format determined through classification then cements the layout structure of the document.

In [4], Tapia and Rojas describe an algorithm for learning the appropriate LaTeX layout for a handwritten mathematical expression. The paper assumes that for a given expression, the individual symbols in the expressions have already been discovered and labeled, and a bounding box has been drawn around each. Given such a setup, Tapia and Rojas define a concept they call dominance. Each symbol dominates a region around it; for example, a summation symbol dominates the region above and below it, where subexpressions are usually located that define the range for the summation. To construct the appropriate LaTeX layout for the expression, Tapia and Rojas notice that LaTeX markup essentially forms a tree structure, where arguments, subscripts, and superscripts are essentially children of some given parent symbol. They devise a method for learning this tree structure. Each symbol is treated as a node and edges between two symbols are weighted based on the distance between the centroids of the bounding boxes of the two symbols. First, the concept of dominance is used to find a dominant baseline of symbols in an expression. The remaining symbols are attached to this baseline by finding the minimum spanning tree that includes the baseline. In terms of the LaTeX structure, the dominant baseline provides the sequence of top level symbols, whereas the branches of the minimum spanning tree that connect to the baseline are the arguments, superscripts, and subscripts of the baseline symbols. If these branches off of the baseline are complex enough, this process is recursively applied to find baselines for each branch, from which smaller minimum spanning trees are then built of the baseline for each of the branches.

2.2 Main Contributions of Our Work

The goal of our project is to create an end-to-end system that takes in as input an image of a typesetted document and outputs the matching LaTeX for the document. Our main contribution resides not so much in significantly improving upon the OCR techniques described in the previous works, but rather in designing a top-down approach for converting a document image into LaTeX by building upon smaller modules that utilize the OCR techniques we have researched. Many of the cited OCR techniques are targeted at isolated problems such as recognition of handwritten characters or math equations. Since we are instead concerned with full images of typesetted documents, the technical challenges at each level of our system are different and slightly simpler compared to those discussed in the corresponding prior works. We also coded up our OCR techniques using as few library calls as possible, in order to really understand the tradeoffs between different methods presented in previous work. Details of the techniques we use in our implementation will be discussed in Section 3.2.

3 Implementation

We begin by giving an overview of our system in Section 3.1, and in Section 3.2, we give specific implementation details of each part of our algorithm.

3.1 Summary of Implementation

The key tasks in constructing a LaTeX representation from an image of a document are extracting the document contents and information from the image and converting this information into LaTeX form.

OCR techniques for extracting information from a document achieve the highest accuracy when the document is oriented in a left to right and top to bottom manner. Scans or photos of documents are not always of optimal quality, and thus document images are not guaranteed to be oriented exactly in this ideal fashion. It is therefore advantageous for our algorithm to first rotate the document image to the proper orientation before any other techniques for extracting document information are utilized.

Given a document image whose orientation is properly aligned in a left to right and top to bottom manner, the next step in extracting document contents is to identify the sections of the image that contain textual information. Focusing on these sections exclusively allows for finer tuned character recognition and information extraction. In general for academic papers, document information is primarily contained in plain text sections and display mode equation blocks. The next phase of our implementation deciphers which parts of the document image correspond to these information containing sections. The structure of information differs between the two types of information blocks, and so during this step it is important to record the type of a document section as well as its location in the image.

Analyzing blocks of text is performed using a hierarchical approach. The algorithm begins by splitting text blocks into lines of text. Each of these lines of text is split into its component words, and the text of each word is then determined using a text classifier. The deciphered word texts are concatenated horizontally to form textual representations of each line and these lines are then concatenated vertically to obtain the contents of the text block.

Identifying the contents of display mode equation sections uses a similar approach where the section is broken down into its individual equations. Each equation is then analyzed to obtain a LaTeX representation. The LaTeX layout for each equation is learned using the method in [4] described in section 2.1. The obtained LaTeX representations are then encapsulated into a display mode LaTeX block which is returned as the contents of the equation section.

After the contents of each document section are extracted from the image, a LaTeX representation of the document is constructed by concatenating the contents of each individual section in the top to bottom order in which they appear. Layout is kept simple as the main focus of the implementation is to maximize the accuracy of the data contained in the document.

3.2 Implementation Details

We now discuss the technical details of the implementation our system. We coded our project in Python with the help of OpenCV 3.0.

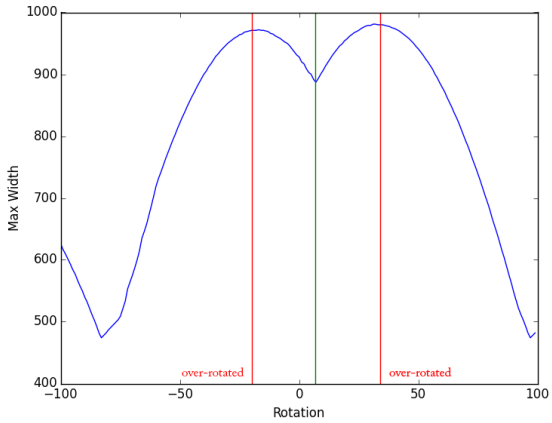


Fig. 1. Encapsulating bounding box widths as a function of rotation for a short and wide document rotated -7°, green line corresponds to desired left to right and top to bottom orientation

3.2.1 Orientation Correction: Technical documents contain information formatted in a rectangular shape with the left, right, top, and bottom margins defining the general structure of this rectangle. If the document were to be rotated, the rectangular form of the document context would also be rotated by the same amount. This leads to the intuition that a left to right and top to bottom oriented bounding box that contains all of a document's context would have a minimum width when the document is in a left to right and top to bottom orientation. If the document were not in said orientation, then the width of the encapsulating bounding box would be some diagonal of the rectangular structure of the document's contents. Naturally, if the rectangular structure of the contents of a document had a greater width than height, the left to right and top to bottom orientation would not yield a global minimum for the width of the encapsulating bounding box. Figure 1 demonstrates such a case. The figure shows that when the image is rotated by large values, the wide and short bounding box of the proper orientation is rotated into a thin and tall bounding box which then yields smaller values for the width. This problem is addressed by taking into consideration the fact that the document images are generated with an understanding of the proper orientation of the document. Thus the probability that a document image will contain a large rotational error is very low and it is reasonable to assume that the rotation of the document will not greatly vary from the optimal orientation. In such cases, the rotation error will be such that using gradient descent to find the local minimum bounding box width will yield the optimal left to right and top to bottom rotation. Figure 1 demonstrates this fact; all points between the two red lines will result in the proper local minimum when gradient descent is used.

In order to rotate an image without accidentally cropping the image, the output image size is first specified to be a square with side length equal to the maximum value between the image width and height. An empty image of this output size is then created and the original document image is translated such that its center is aligned with the center of the new image. This image is then rotated around its center by the desired amount. Defining the homogenized coordinates of the image matrix as H, the rotation angle to be θ, the original image width to be w, the original image height to be h, and the homogenized image coordinates of rotated image matrix to be H' the specified transformation is given by:

$$\alpha = \cos\theta$$
$$\beta = \sin\theta$$
$$T = \begin{bmatrix} 1 & 0 & \frac{\max(w,h)-w}{2} \\ 0 & 1 & \frac{\max(w,h)-h}{2} \\ 0 & 0 & 1 \end{bmatrix} H$$
$$H' = \begin{bmatrix} \alpha & \beta & (1-\alpha-\beta)\frac{\max(w,h)}{2} \\ -\beta & \alpha & (1-\alpha+\beta)\frac{\max(w,h)}{2} \\ 0 & 0 & 1 \end{bmatrix} T$$

In accordance with our findings, the algorithm for finding the correct document orientation utilizes gradient descent to find the rotation of the document that results in a local minimum of the encapsulating bounding box width. A bounding box containing the contents of the document is first constructed using methods similar to those discussed in section 3.2.2. Defining θ to be a rotation angle and the encapsulating bounding box width after a rotation of θ to be W_θ, an approximation of the derivative of the bounding box width with respect to the rotation angle is given by:

$$\frac{W_{\theta-\delta} - W_{\theta+\delta}}{2\delta}$$

where δ represents a slight difference in rotation angle. Since the algorithm operates under the assumption that the document is not rotated by an extreme value, this derivative is first calculated for a rotation of 0 to determine whether increasing or decreasing the rotation will yield a smaller bounding box width. The algorithm then continues increasing or decreasing the rotation until the bounding box width stops decreasing. At this point the local minimum bounding box width has been located and with it the rotation that yields the left to right and top to bottom orientation of the document image. Applying this rotation to the document image thus yields a properly oriented document image which can be used for accurate data extraction.

3.2.2 Text and Equation Sectioning: In order to accurately extract the content of a document from the document image, the sections of the document that contain meaningful information must first be identified. Once these sections are located, a hierarchical approach is used in order to obtain finer bounding boxes from which data can be accurately extracted.

The first step in this document sectioning is to identify which areas of the document image contain data and which areas of the document are meaningless background. In order to achieve this result a grayscale representation of the document image is obtained. A general horizontal and vertical Sobel operator is then applied to this grayscale image in order to obtain an image representation that focuses on the edges of the characters in the document. The intuition for this step follows

from the idea that characters are very edge heavy figures and consequently areas of a document containing characters can be identified by looking for areas of the document with a large density of edges. Following the application of the Sobel operator, the image is binarized using a threshold. A standard closing morphological transformation, which is a dilation followed by an erosion, is then used in order to connect previously identified edges present in the image. At this point our image representation will consist of large white blocks which correspond to the areas of the image containing meaningful information with the rest of the image being black. Identifying the contours in the image then allows us to obtain bounding boxes containing the desired sections of the document. During the contour finding, bounding boxes with small areas are discarded as these correspond either to meaningless artifacts in the document image or characters in the image which do not contain important information, e.g. page numbers. Steps in this process are shown in Figures 2, 3, and 4.

Answer to Question 1(a)

The convolutional neural network architecture was chosen because it outperformed every other algorithm for OCR on handwriting digit recognition using the MNIST benchmark [6,11]. The quick brown fox jumped over the lazy dog. The lazy dog wanted to know why the quick brown fox was going for a random walk. We model the length of this random walk with the random variable Z. Thus, the lazy dog wants to know $E[Z]$. Using math, we have

$$\begin{aligned} E[Z] &= 1 - \beta \\ &= 1 - (1-\alpha) * (1-\gamma) \\ &= 1 - (1-\alpha) * \sum_{i=1}^{4} (1-\beta_i) \\ &= 2x + y \end{aligned}$$

At this point, the dog gave up on trying to understand what was going on. The math seemed to jump to random conclusions, many of which were doubtful in the first place. The dog thought that $x = \beta + 3$ and not $y = 2x + 73$, but it was wrong.

Fig. 2. Original document image.

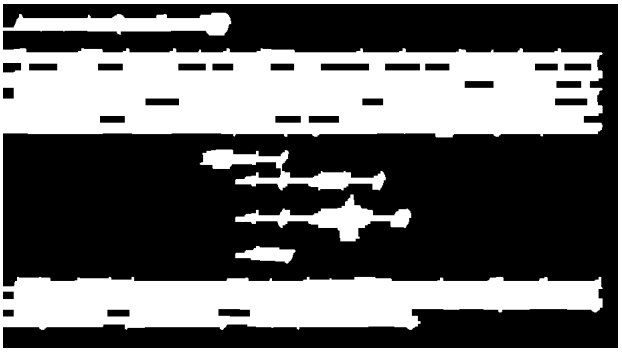


Fig. 3. Document image after processing and close morphology transformation.

Following the identification of document sections containing information, the algorithm next classifies the obtained bounding boxes as corresponding to either display mode equation blocks or plain text paragraphs. The classification process focuses on the location of the left edge of the bounding box and classifies bounding boxes that are aligned near the left margin of the document as text while other bounding boxes are classified as equation blocks.

Given a section of an image corresponding to a block of equations, the goal of the algorithm is then to separate

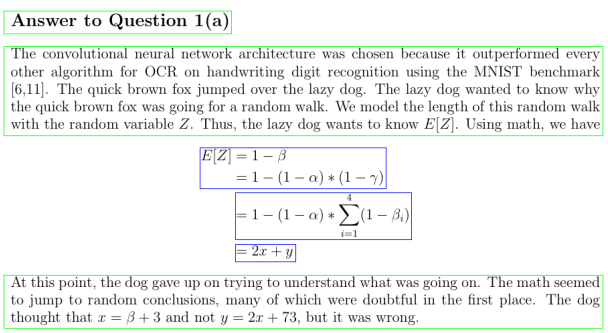


Fig. 4. Content bounding boxes in document image, text is represented by green boxes and equation blocks are represented by blue boxes

this equation block into its component equations. This is accomplished using a method similar to the higher level document sectioning but with finer granularity. First the equation block image is scaled to a standard width of 700 pixels and a grayscale representation is generated. A Gaussian Blur is then applied to this grayscale representation in order to slightly increase the size of the characters in the equations while maintaining the spacing between them. This results in an increased cohesiveness between the entities of a single equation without significantly decreasing the spacing between separate equations. A Laplace operator is then applied to obtain tracings of the equations and the image is binarized using a threshold. Following the intuition that the entities in an equation are contained in approximately the same line, a strong horizontal closing morphological transformation is applied to the thresholded image in order to link the symbols of each equation.

At this point, finding contours yields bounding boxes that completely contain the vast majority of equations, however, disconnected objects such as numerators will be contained in their own individual bounding boxes. This can be remedied by extending the bounding boxes found so far horizontally and joining the boxes that intersect into one larger object. Since entities in each equation are relatively in line with each other but separate equations have vertical separation between them, extending the bounding boxes and joining them will connect the disconnected equation components without connecting separate equations. Steps in this process are shown in Figures 5, 6, and 7.

$$\begin{aligned} E[Z] &= (1-\beta) * 1 + \beta * (1 + E[Z]) \\ E[Z] &= 1 + \beta E[Z] \\ (1-\beta) E[Z] &= 1 \\ E[Z] &= \frac{1}{1-\beta} \end{aligned}$$

Fig. 5. Original equation block.

The primary objective when segmenting text blocks is to segment the previously located paragraphs into bounding boxes containing each individual word. This is accomplished

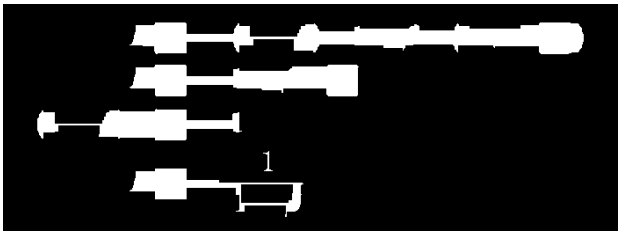


Fig. 6. Equation block after processing and close morphology transformation.

Fig. 7. Equation block after bounding box extension and fill.

by first segmenting the paragraph into lines, and then segmenting the lines into words. Paragraphs are segmented into lines in a similar manner to how equation blocks are separated into equations. The paragraph image is scaled to a standard width of 700 pixels after which the image is converted to grayscale and a Gaussian Blur is applied. Applying a Laplace operator, threshold, and then strong horizontal closing morphological transformation yields blocks that correspond to the lines in the paragraph. The bounding boxes of these contours then define the locations of these lines. An example segmentation can be seen in Figure 8.

The key difference when segmenting lines into words is that the spacing between words in a line is horizontal whereas the spacing between lines in a paragraph is vertical. In order to segment line images into words, the line image is first standardized to a height of 20 pixels before being converted to grayscale. A Laplace operator is applied followed by a threshold and a strong vertical and slight horizontal closing morphological transformation. The slight horizontal component of the closing morphology closes the spaces between characters in the word while the strong vertical component guarantees that the words are represented by solid blocks in the line. Locating the spaces between the resulting blocks allows the algorithm to construct bounding boxes for each word in the line. An example segmentation of a line can be seen in Figure 9.

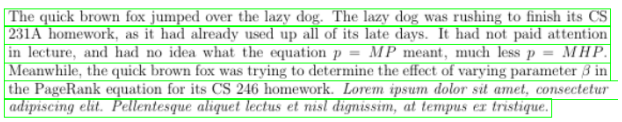


Fig. 8. Example of segmentation of a paragraph into lines.

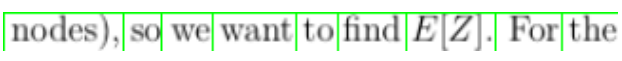


Fig. 9. Example of segmentation of a line into words.

3.2.3 Individual Character Recognition: The lowest level of our system is a character recognizer which can classify various symbols. Our current implementation supports alphabetical, numeric, and Greek letters; common punctuation, parentheses, and brackets; and common math symbols such as $+$, $=$, $\leq$, $\sum$, $\int$, ∂, and $\sqrt{\cdot}$. Given a bounding box around a symbol in an image, the goal of the character recognizer is to classify the symbol and return a key containing the appropriate LaTeX string. This is used to determine the characters that make up an individual word, which we will discuss later in this section, and the symbols that are in mathematical expressions, which will be discussed in Section 3.2.4.

Given a bounding box around a character, our classifier first constructs a feature vector corresponding to the image. We implemented two methods of extracting feature vectors from characters. The first approach is fairly straightforward; we start by scaling the box to be a 20×20 square and then simply use the pixel values as features. To do so, we convert the image to a "binary" format by applying a threshold. The dark pixels which make up the character are considered to be "on" pixels represented by a 1. White pixels are represented as 0. Thus, we can describe each character by its pixel values using a feature vector in $\mathbb{R}^{400}$ comprised of 1s and 0s. Since these vectors are very sensitive to the location of the character itself within its bounding box (i.e. slight translations of the character may produce very different feature vectors), we standardize this by cropping the given bounding box even further to be as small as possible and tightly wrapped around the image. This is done before scaling to 20×20 pixels. Note that this additional cropping necessarily means that this type of feature vector is invariant to the size of the letter, which has implications we discuss in Section 4.

The second approach to extracting a feature vector from a character image is adapted from what is presented in [2]. For simplicity, our feature vector used 10 of the 16 attributes discussed by Frey and Slate. Some of the attributes were not applicable to our system, while others were too complex for the scope of our project. For example, since we standardize the height of each line to be 20 pixels before segmenting into words and characters, the "height" attribute of the image is of little use to us. We noticed that the value range of each attribute varied greatly, so we scaled each attribute to lie approximately within the range [0,15], similar to what was done in [2]. That is, during training, we first recorded the maximum and minimum value for each attribute and then used this range to scale each value.

We trained our classifier on hundreds of images of individual characters while varying the resolution, size, and orientation of each symbol. Our training set necessarily included both regular text and math mode symbols. We experimented with many different types of models used in machine learning, such as k-nearest neighbors (k-NN), SVMs, logistic regression, and decision trees. Ultimately, we decided to focus on two models – k-NN and logistic regression – since, in addition to classifying the character, these models also provided a "score" that indicated the confidence of the prediction. A confident classification would yield small distances in k-NN and high probabilities in logistic regression. The importance of having

a "score" will be discussed in a few paragraphs.

After segmenting paragraphs into individual words using the methods in Section 3.2.2, the final step is to segment each of these words into characters so that our character recognizer can classify them. This final segmentation is one of the more difficult parts to implement, since it is sensitive to even the slightest variation in character resolution and/or spacing. We tried two main approaches to solve the character segmentation problem, each with its own benefits and drawbacks.

The first approach involves applying transformations to the image of the word, similar to what was done at the paragraph- and line-level in Section 3.2.2. To handle images with lower resolution, we first apply a Gaussian Blur and closing morphological transformation in order to ensure that the pixels within a character are indeed connected. The challenge arises due to the fact that characters in the same word are packed very tightly with little whitespace in between, so our Gaussian Blur and closing morphological transformation are very small to avoid gluing two characters together. For example, at our standard height of 20 pixels, we found that the (3,3) kernel seemed to do best for our blur. Finally, we apply a weak threshold to attempt to remove any small artifacts from the word image, and then we proceed to draw contours and bounding boxes. Figure 10 shows an example of successful character segmentation using transformations.

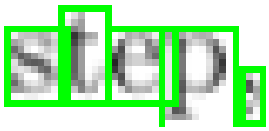


Fig. 10. Successful segmentation of "`step,`".

Our second approach is an adaptation of the slice method explored in [1]. In [1], Jacobs et al. break up words into 2-pixel slices, and various combinations of slices are concatenated together to potentially form characters. In our approach, we use this as motivation and implement a sliding window algorithm that scans the word horizontally. The intuition builds off the fact that our models provide a "score" of confidence for each classification, and we assume our classifier is robust enough to yield high scores for images of actual characters. First, we assume that the width of any symbol must fall within the range of 2 to 15 pixels. Then, starting at the beginning of the word, we call our classifier on all bounding boxes of integer width w between 2 and 15. Thus, each size bounding box yields a classification value and score, and so we pick the width w' (and corresponding value) that yields the highest score. Under the assumption that text characters in paragraphs do not overlap horizontally, we then move our window forward by w' to find the next character. Below is an approximate outline of our sliding window procedure, with Figure 11 providing a visual example of what occurs.

1) Initialize $x \leftarrow 0$.
2) Get (key,score) of all bounding boxes with horizontal length spanning $[x, x+w]$ for all widths $w \in \{2,3,4,\ldots,15\}$.
3) Let w' be the width producing the best score.
4) Record key corresponding to w'.
5) $x \leftarrow x + w'$
6) Repeat steps 2-4 while $x <$ wordWidth.

Fig. 11. The sliding window segments the beginning `n` in "`nodes),`". Boxes in light blue are all possible tested bounding boxes; the line in red yielded the highest score.

3.2.4 Learning LaTeX Expression Layout: Transforming images of single line mathematical expressions requires us to not only recognize individual symbols used in the expression, but also the layout of the expression. To perform this, we used a simplified version of the method used in [4], which we will now describe in detail.

The first step in transforming a mathematical expression to LaTeX markup is to classify each individual symbol in the expression. This is done by calling upon our character recognizer discussed in Section 3.2.3. The character recognizer gives, or classifies, each bounding box with a key which contains the appropriate LaTeX string for the symbol. Once symbols are matched with the individual LaTeX strings, the next step is to learn the structure of the expression to assemble theses strings together in a coherent fashion.

To do this, we work with a few more attributes of each symbol. Each bounding box of a symbol is described by four values, x_m, y_m, w and h, where x_m and y_m denote the (x,y) coordinates of the top-left corner of the box and w and h denote the width and height of the box. Coordinates of pixels in the image have x values that increase from left to right and y values that increase from top down.

For each bounding box detected in the image for an expression, we define three additional values. The *centroid* of a symbol is a point in the bounding box that indicates, roughly, where the center of the symbol lies. The *sup-threshold* of a symbol denotes the upper bound for the y value of a superscript, if one were to be attached to the symbol, while the *sub-threshold* denotes the lower bound for the y value of a subscript. Figure 12 gives a visual representation for each of these definitions.

Each symbol is categorized as one of three types: *ascendant*, *descendant*, *central*. These distinctions allow us to distinguish characters that are imbalanced in the y direction. For example, characters like ∂ and b are ascendant symbols, γ and p are descendant symbols, and c and $\int$ are central symbols. Based on the type of the symbol, we compute the values for the centroid, sup-threshold, and sub-threshold differently, as shown in Table I. These values, together with x_m, y_m, w and h are used to compare locations between symbols, which we explain next.

One of the main ideas presented in [4] is the concept of dominance. Each symbol is capable of dominating some subset of five different regions: *above*, *below*, *superscript*, *subscript*, and *subexpression*. Tapia and Rojas define a few additional regions in [4], but for the sake of simplicity, the five regions we have defined cover all of the symbols we currently support.

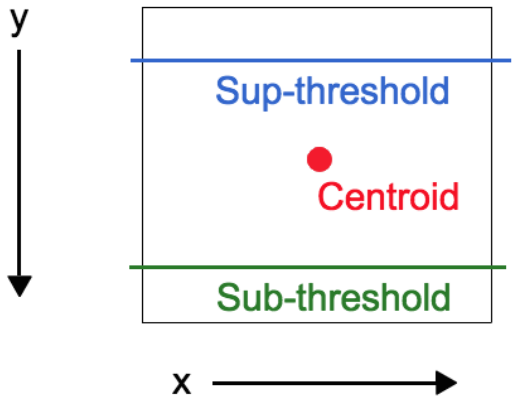


Fig. 12. Points and lines of interest for a bounding box

TABLE I
CENTROID AND THRESHOLD VALUES FOR DIFFERENT SYMBOL TYPES

	centroid y	sup-threshold	sub-threshold
ascendant	$y_m + 0.66h$	$y_m + 0.2h$	$y_m + 0.8h$
descendant	$y_m + 0.33h$	$y_m + 0.1h$	$y_m + 0.4h$
central	$y_m + 0.5h$	$y_m + 0.25h$	$y_m + 0.75h$

The locations of each of these regions, relative to the bounding box, is show in Figure 13. A symbol A is said to dominate a symbol B if B lies in one of the regions dominated by A, and B does not exceed A in both width and height. Now, given

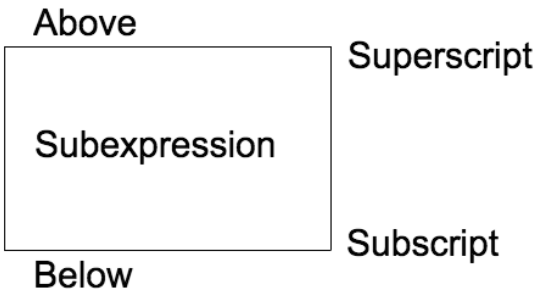


Fig. 13. Dominated regions of a symbol

the image of the expression and the bounding box for each symbol, we wish to learn the LaTeX layout of the symbols. For example, consider the following expression

$$\sum_{n=-\infty}^{\infty} \left| \left\langle f, \frac{e^{inx}}{\sqrt{2\pi}} \right\rangle \right|^2 = \|f\|^2$$

The actual LaTeX markup corresponding to this expression can be broken down into a tree structure, shown in Figure 14. To learn this tree structure, we first find the *dominant baseline* of symbols, which consists of all of the children of the root node. These are the primary symbols in our LaTeX expression that are not dominated by any other symbols, and in general will have centroids that lie roughly on the same line. To find this baseline, we find the most dominant expression, defined as the expression with the largest bounding box perimeter that is not dominated by any other expressions. In our example expression, the most dominant expression is given by the summation symbol. We then compute the centroid of the most dominant expression and find all other dominant expressions with centroids that have similar y coordinates. Figure 15 shows the bounding boxes and the baseline symbols for the example expression.

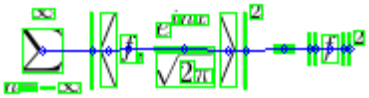


Fig. 15. Baseline symbols

Once the dominant baseline is established, the centroids of the nodes in the baseline are connected from left to right to form a line structure. All remaining symbols are attached to the baseline nodes using Prim's algorithm for minimum spanning trees. For each branch of the spanning tree connected to the baseline, this process of finding the most dominated symbol, followed by finding the dominant baseline, and lastly constructing the minimum spanning tree is recursively repeated. Ultimately, this yields a tree like the one in Figure 16. The recursive layering of this algorithm naturally produces well-defined parent-child relationships between the nodes of the tree. For each parent symbol, child nodes in same dominated regions are grouped together as sibling nodes, which allows us to construct the LaTeX tree in Figure 14. From the tree, we can construct the appropriate LaTeX markup string, since children either serve as arguments, superscripts, or subscripts of their parents.

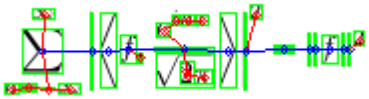


Fig. 16. Final spanning tree of symbols

3.2.5 Latex Construction: The final step is to connect all the parts of our system and generate the actual LaTeX markup for the document image. Given our top-down design, this part is very straightforward. At the top level, we orient the document properly and identify which sections are text paragraphs and which sections are math equations. For text paragraphs, we split into lines, then words, then characters. Our classifier gives a key containing the predicted LaTeX for each character, so each word can aggregate a list of keys, one for each character. Similarly, each line can construct a list of words, and finally, each paragraph can construct a list of lines. At this point, the paragraph has all the information it needs in its nested lists of keys. Each key contains the actual LaTeX markup as well as a flag denoting if it is a math symbol. In the case of our text paragraphs, a string of math symbols must be surrounded by dollar signs (i.e. `$...$`).

For math equations, we use the tree structure along with our classifier to determine the LaTeX markup. In particular, the classifier again gives us the markup for each character (we

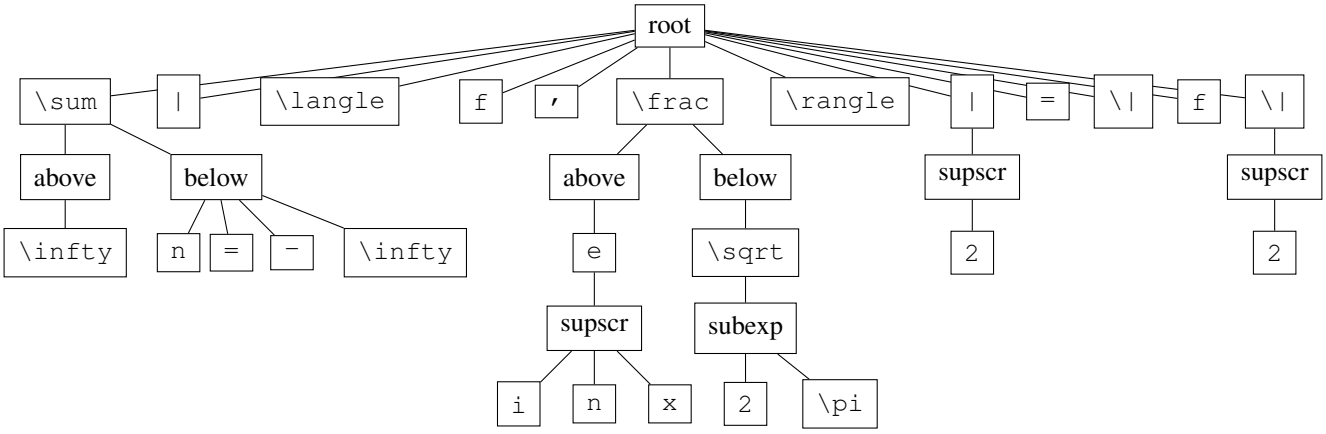


Fig. 14. LaTeX markup tree

can ignore the key flag this time), and the tree structure lets us know when to insert the appropriate markup for subscript, superscript, underset, etc.

Finally, at the top level, we arrange the LaTeX of each text paragraph and math equation in proper order and surround it with the necessary LaTeX boilerplate, such as the preamble. Our final output is the entire LaTeX markup that is ready to be compiled.

4 Experimentation Results

We begin by first analyzing the effectiveness of our classifier. As discussed in Section 3.2.3, we implemented two different ways of extracting feature vectors from characters. The first approach used the actual pixel values to form a feature vector in $\mathbb{R}^{400}$, while the second approach used attributes discussed by Frey and Slate in [2] to form a feature vector in $\mathbb{R}^{10}$. To test our classifier outside of our system, we generated 300 images of characters in a manner similar to how we generated our training data. Thus, the test images are not the same as our training data, but we ensure that we test only on symbols that we have attempted to support. Again, we use two models: k-NN and logistic regression. Table II shows the correct prediction rates for each model.

TABLE II
Correct classification rates: all characters

features	k-NN	logistic regression
pixel values	0.867	0.883
Frey-Slate	0.840	0.873

We can see that using the logistic regression model and pixel values as features seems to be slightly better, but not by any significant amount. After performing some error analysis, however, we discovered an interesting pattern that gave us insight regarding the strength and weaknesses of the two types of features. Table III shows the correct prediction rates for each model without considering punctuation symbols.

There are significant differences in our results here. Without considering punctuation symbols, using the pixel values as features performs much better than using the Frey-Slate attributes. Thus, we can see that pixel values predict medium and large symbols very well, while the Frey-Slate attributes do not perform strongly. However, the pixel values have difficulty classifying small punctuation symbols, whereas the performance of the Frey-Slate attributes actually increases.

TABLE III
Correct classification rates: no punctuation

features	k-NN	logistic regression
pixel values	0.946	0.951
Frey-Slate	0.772	0.786

This makes sense in accordance with how we extract feature vectors for each character. When using pixel values as features, we crop the bounding box to be as tight as possible around the "on" pixels in order to mitigate the effects of slight translations. We noted that this made our feature vectors invariant to the relative size of the character itself, and this is why punctuation mark prediction accuracy suffers. For example, when we are extracting pixel values for the period symbol ".", the cropped bounding box only leaves about a 2×2 box of all 1s. When it is scaled to 20×20, this means the entire feature vector is comprised of all 1s. Consider another geometric shape like the minus sign "–". The cropped bounding box is about a 5×2 box, again of all 1s. Thus, the pixel value feature vector for the period and minus will be exactly the same, and so our classifier will have trouble differentiating between the two. On the other hand, the Frey-Slate attributes take character size into account along with other properties such as "bottom-heavy," so punctuation marks do not perform any worse.

As for non-punctuation symbols, it is likely that pixel values perform well because we have constrained our documents to use the same font type. Thus, for each character type, the pixel value feature vector in $\mathbb{R}^{400}$ carries much more useful information than the Frey-Slate attribute vector in $\mathbb{R}^{10}$. It would be interesting to see how well each type of feature vector performs if we were to extend our problem to consider different font types, which we discuss briefly in Section 5. In the end, these results prompted us to use the logistic regression model on pixel value feature vectors as our character classifier.

We now discuss the strengths and weaknesses of each of

our approaches to segmenting words into characters. One approach was to use a sliding window over each word to find the optimal bounding boxes that identified each character. While the sliding window usually worked very well, as shown in Figure 11, error analysis revealed instances in which it performs very poorly. This occurs when we have a symbol such as `w` in which a subset of the symbol can also feasibly be a different one. In this case, half of a `w` closely resembles a `v`. Figure 18 visualizes this phenomenon.

Fig. 17. The sliding window has trouble resolving remaining ambiguities. Here, the `w` may also be interpreted as two `v`s.

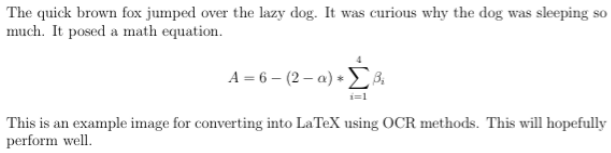
We also encountered edge cases in which a very skinny bounding box would capture the edge of a serif, and small punctuation marks such as the period "`.`" would then return a very high score. To handle these various failure cases, we added on various heuristics to our basic sliding window algorithm outlined in Section 3.2.3. For example, instead of simply picking the width w' that yielded the absolute highest score, we considered all widths that yielded scores above a threshold value, such as 0.9 probability. We then applied various rules to try to correctly pick the true correct character prediction, which marginally improved our results. The sliding window fails and is unrecoverable when a bad w' is picked and the x in our algorithm is then incremented incorrectly, so any small improvement is still very helpful.

Our second approach to word segmentation involved carefully applying transformations to the image and drawing bounding boxes around contiguous pixel regions. After many tests, we found that this approach is not nearly as consistent as the sliding window. Often times, low resolution and tightly-packed characters are too much for simple transformations to be able to overcome. Even after picking transformations parameters as best as we could, common errors included "gluing" two characters together, splitting a character, and identifying small artifacts. Figure 18 shows these errors.

Fig. 18. Incorrect segmentation of "`lecture.`". Some letters are glued together, the `u` is split, and there are small artifacts.

These problems were very difficult to overcome and thus, we were unable to improve performance in this area by very much. The issue revolves around the fact that the "gluing" and "splitting" problems are a tradeoff. Individual letters, such as `u`, are split when the necessary application of a threshold on a low-resolution image effectively separates the symbol, usually along a curve. Combating this by increasing the size of our Gaussian Blur kernel, though, fuels the other problem in that tightly packed letters may be glued together. One benefit of using this approach, however, is that even when incorrect bounding boxes are applied, it does not necessarily damage our ability to classify the rest of the image, whereas one incorrect w' in the sliding window ruins the rest of the word. Over many trials, though, we still felt that the sliding window performed better, and so we chose this as our primary approach.

Finally, we give an example of our entire end-to-end system in action. Figure 19 shows an image of a PDF document that we input into our system, and Figure 20 shows the LaTeX that we generate.


The quick brown fox jumped over the lazy dog. It was curious why the dog was sleeping so much. It posed a math equation.

$$A = 6 - (2 - \alpha) * \sum_{i=1}^{4} \beta_i$$

This is an example image for converting into LaTeX using OCR methods. This will hopefully perform well.

Fig. 19. Original image.

```
\documentclass[11pt]{article}
\usepackage{algorithm, algpseudocode, amsmath, amsfonts, cite, graphicx,
icomma, multirow, url, xspace, tikz, qtree}
\begin{document}
The qu$h]$ brown fox jumpeo ouI the lazy a. $—— $cu$-$os. why the $do$g was
sleepino so much. Itp$—$. a math equ$—$j \\ \\
\begin{align*}
A = 6 - \left( 2 - \alpha \right) * \sum_{i=1}^4 \beta \\
\end{align*}
Thi is an example image fd$-$ converIdh into La$1$ex using OCR me$—$ds. It w.i
h$-$ perform weH. \\ \\
\end{document}
```

Fig. 20. Corresponding LaTeX outputted by our system.

The qu$h]$ brown fox jumpeo ouI the lazy a. $---$cu$-$os. why the dog was sleepino so much. Itp$--$. a math equ$--$j

$$A = 6 - (2 - \alpha) * \sum_{i=1}^{4} \beta$$

Thi is an example image fd$-$ converIdh into La1ex using OCR me$--$ds. It w.i h$-$ perform weH.

Fig. 21. Compiled PDF from the LaTeX in Figure 20.

Overall, our system does moderately well. It is fairly easy to see the resemblance between the outputted LaTeX and the original image. There are, of course, some noticeable errors, but we can attribute a lot of them to our issues with the sliding window and character classifier that we discussed earlier in this Section.

5 Conclusion

In this paper we developed an end-to-end algorithm for the conversion of images of documents into LaTeX representations. While there were quite a few errors in the LaTeX documents generated by the algorithm, the results of each of the individual components of the algorithm yielded accurate results. It is likely that the combination of small errors in each of the individual components compounded into larger errors in the end-to-end algorithm. Notably, our character recognition accuracy was significantly lower when analyzing words that are a part of the document. This accuracy could

likely be improved by leveraging more advanced OCR libraries and leveraging previous work in OCR to implement more sophisticated classification procedures.

Our project served as a proof of concept that even with simpler OCR techniques, it was still possible to learn document and equation layouts as well as recognize individual characters and combine these together to convert an image of a document to compilable LaTeX markup. Though each step of our algorithm could be improved upon to reduce the overall error, the method we created to break the problem down step by step does seem to be a good approach, as it modularizes and compartmentalizes tasks so that each can focus on the different types of issues that can occur at different levels of granularity.

For future work, we would like to improve the robustness of our algorithm, particularly that of the individual character recognition step, and extend our algorithm to work on different fonts. With more advanced OCR techniques, it may even be possible to adapt our method to work for handwritten documents such as lecture notes.

REFERENCES

[1] C. Jacobs, P. Simard, P. Viola, J. Rinker. Text Recognition of Low-resolution Document Images. Microsoft, 2005.
[2] P. Frey, D. Slate. Letter Recognition Using Holland-Style Adaptive Classifiers. Machine Learning, 1991.
[3] G. Gupta, S. Niranjan, A. Shrivastava. Document Layout Analysis & Classification and Its Application in OCR. EDOCW, 2006.
[4] E. Tapia, R. Rojas. Recognition of On-line Handwritten Mathematical Expressions Using a Minimum Spanning Tree Construction and Symbol Dominance. LNCS, 2004.