A Math Formula Extraction and Evaluation Framework for PDF Documents

Ayush Kumar Shah[0000−0001−6158−7632], Abhisek Dey[0000−0001−5553−8279], and Richard Zanibbi[0000−0001−5921−9750]

Rochester Institute of Technology, Rochester NY 14623, USA
{as1211,ad4529,rxzvcs}@rit.edu

Abstract. We present a processing pipeline for math formula extraction in PDF documents that takes advantage of character information in born-digital PDFs (e.g., created using LaTeX or Word). Our pipeline is designed for indexing math in technical document collections to support math-aware search engines capable of processing queries containing keywords and formulas. The system includes user-friendly tools for visualizing recognition results in HTML pages. Our pipeline is comprised of a new state-of-the-art PDF character extractor that identifies precise bounding boxes for non-Latin symbols, a novel Single Shot Detector-based formula detector, and an existing graph-based formula parser (QD-GGA) for recognizing formula structure. To simplify analyzing structure recognition errors, we have extended the LgEval library (from the CROHME competitions) to allow viewing all instances of specific errors by clicking on HTML links. Our source code is publicly available.

Keywords: math formula recognition · document analysis systems · PDF character extraction · single shot detector (SSD) · evaluation

1 Introduction

There is growing interest in developing techniques to extract information from formulas, tables, figures, and other graphics in documents, since not all information can be retrieved using text [3]. Also, the poor accessibility of technical content presented graphically in PDF files is a common problem for researchers, and in various educational fields [19].

In this work, we focus upon improving automatic formula extraction [21]. Mathematical expressions are essential in technical communication, as we use them frequently to represent complex relationships and concepts compactly. Specifically, we consider PDF documents, and present a new pipeline that extracts symbols present in PDF files with exact bounding box locations where available, detects formula locations in document pages (see Fig. 1), and then recognizes the structure of detected formulas as Symbol Layout Trees (SLTs). SLTs represent a formula by the spatial arrangement of symbols on writing lines [23], and may be converted to LaTeX or Presentation MathML.

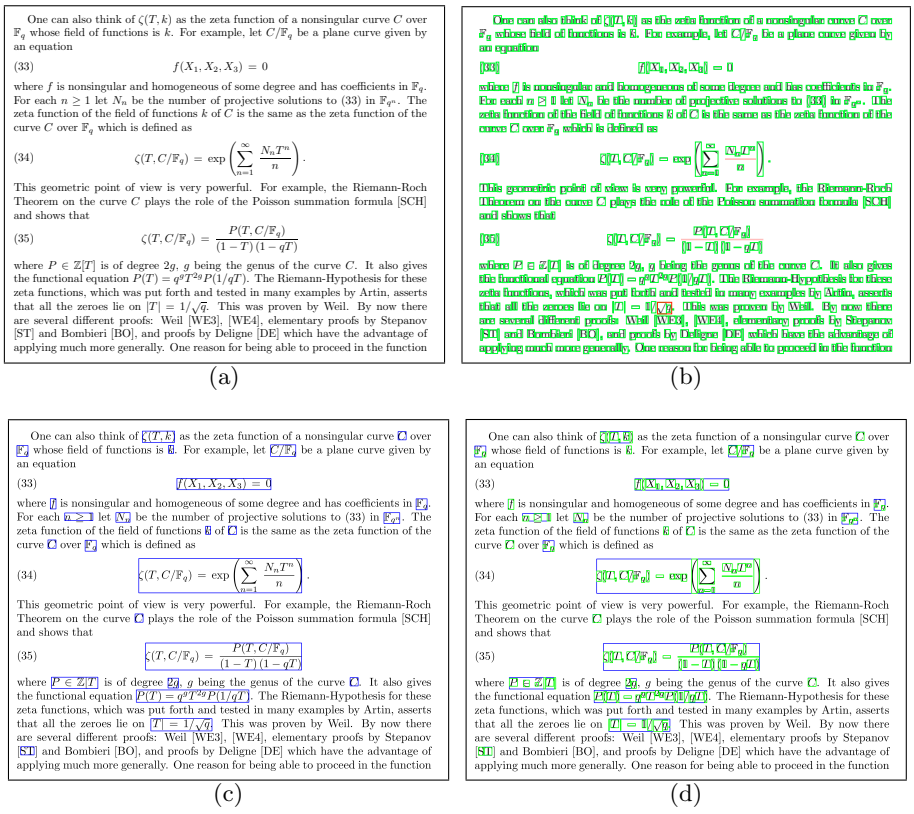


Fig. 1. Detection of symbols and expressions. The PDF page shown in (a) contains encoded symbols shown in (b). (c) shows formula regions identified in the rendered page image, and (d) shows symbols located in each formula region.

There are several challenges in recognizing mathematical structure from PDF documents. Mathematical formulas have a much larger symbol vocabulary than ordinary text, formulas may be embedded in textlines, and the structure of formulas may be quite complex. Likewise, it is difficult to visualize and evaluate formula structure recognition errors - to address this, we present an improved evaluation framework that builds upon the LgEval library from the CROHME handwritten math recognition competitions [14,12,10]. It provides a convenient HTML-based visualization of detection and recognition results, including the ability to view *all* individual structure recognition errors organized by ground truth subgraphs. For example, the most frequent symbol segmentation, symbol classification, and relationship classification errors are automatically organized in a table, with links that take the user directly to a list of specific formulas with the selected error.

The contributions of this work include:

1. A new open-source formula extraction pipeline for PDF files,[1]
2. new tools for visualization and evaluation of results and parsing errors,
3. a PDF symbol extractor that identities precise bounding box locations in born-digital PDF documents, and
4. a simple and effective algorithm which performs detection of math expressions using visual features alone.

We summarize related work for previous formula extraction pipelines in the next Section, present the pipeline in Section 3, our new visualization and evaluation tools in Section 4, preliminary results for the components in the pipeline in Section 5, and then conclude in Section 6.

2 Related Work

Existing mathematical recognition systems use different detection and recognition approaches. Lin et al. identify three categories of formula detection methods, based on features used [7]: character-based (OCR-based), image-based, and layout-based. Character-based methods use OCR engines, where characters not recognized by the engine are considered candidates for math expression elements. Image-based methods use image segmentation, while layout-based detection uses features such as line height, line spacing, and alignment from typesetting information available in PDF files [7], possibly along with visual features. Likewise, for parsing mathematical expressions, syntax (grammar-based) approaches, graph search approaches, and image-based RNNs producing LaTeX strings as output are common.

This Section summarizes the contributions and limitations of some existing math formula extraction systems, and briefly describes our system's similarities and differences.

Utilizing **OCR and recognition as a graph search problem**, Suzuki et al.'s INFTY system [18] is perhaps the best-known commercial mathematical formula detection and recognition system. The system uses simultaneous character recognition and math-text separation based on two complementary recognition engines; a commercial OCR engine not specialized for mathematical documents and a character recognition engine developed for mathematical symbols, followed by structure analysis of math expressions performed by identifying the minimum cost spanning-trees in a weighted digraph representation of the expression. The system obtains accurate recognition results using a graph search-based formula structure recognition approach.

Using symbol information from PDFs directly rather than applying OCR to rendered images was first introduced by Baker et al. [2]. They used a syntactic pattern recognition approach to recognize formulas from PDF documents using an expression grammar. The grammar requirement and need for manually segmenting mathematical expressions from text make it less robust

[1] https://www.cs.rit.edu/~dprl/software.html

than INFTY. However, the system was faster by avoiding rendering and analyzing document images, and improved accuracy using PDF character information. In later work, Sorge et al. [16] were able to reconstruct fonts not embedded in a PDF document, by mapping unicode values to standard character codes where possible. They then use CC analysis to identify characters with identical style and spacing from a grouping provided by `pdf2htmlEX`, allowing exact bounding boxes to be obtained. Following on Baker et al.'s approach to PDF character extraction [2], Zhang et al. [24] use a dual extraction method based on a PDF parser and an OCR engine to supplement PDF symbol extraction, recursively dividing and reconstructing the formula based on symbols found on the main baseline for formula structure analysis. Later, Suzuki et al. improved the recognition rate in INFTYReader [18], by also utilizing extracted PDF character information from PDFMiner [19].

Some PDF characters are composed of multiple glyphs, such as large braces or square roots (commonly drawn with a radical symbol connected to a horizontal line). These 'compound' characters were identified by Baker et al. [2] using overlapping bounding boxes in modern PDF documents containing Type 1 fonts.

Image-based detection using RNN-based recognition was first proposed by Deng et al. [4], inspired by RNN-based image captioning techniques. A more recent example of this approach is Phong et al.'s [15], which uses a YOLO v3 network based on a Darknet-53 network consisting of 53 convolutional layers for feature extraction and detection, and an advanced end to end neural network (Watch, Attend and Parse (WAP)) for recognition. The parser uses a GRU with attention-based mechanisms, which makes the system slow due to the pixel-wise computations. Also, error diagnosis in these recurrent image-based models is challenging, since there is not a direct mapping between the input image regions and the LaTeX output strings. To address this issue for the CROHME 2019 handwriten math competition, evaluation was performed using the LgEval library, comparing formula trees over recognized symbols (e.g., as represented in LaTeX strings), without requiring those symbols to have assigned strokes or input regions [10]. This alleviates, but does not completely resolve challenges with diagnosing errors for RNN-based systems.

This paper. We introduce algorithms to create a unified system for detecting and recognizing mathematical expressions. We first introduce an improved PDF symbol extractor (SymbolScraper) to obtain symbol locations and identities. We also present a new Scanning Single Shot Detector (ScanSSD) for math formulas using visual features, by modifying the Single Shot Detector (SSD) [8] to work with large document images. For structure recognition, we use the existing Query-driven Global Graph Attention (QD-GGA) model [9], which uses CNN based features with attention. QD-GGA extracts formula structure as a maximum spanning tree over detected symbols, similar to [18]. However, unlike [18], the system trains the features and attention modules concurrently in a feed-forward pass for multiple tasks: symbol classification, edge classification, and segmentation resulting in fast training and execution.

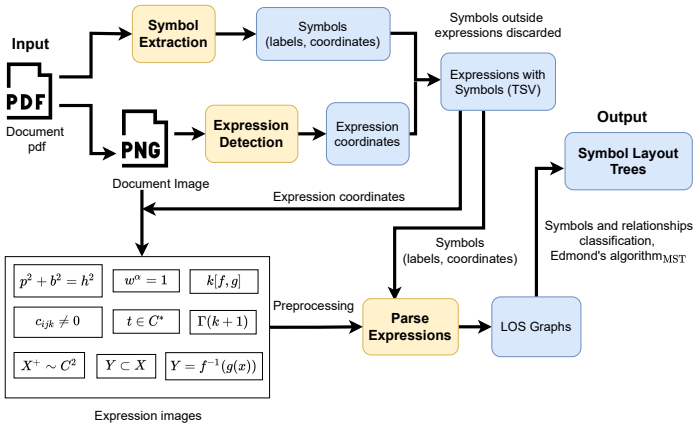


Fig. 2. Mathematical Expression Extraction Pipeline. *Symbol Extraction* outputs symbols and their bounding boxes directly from drawing commands in the PDF file, avoiding OCR. However, formula regions and formula structure are absent in PDF, and so are identified from rendered document pages (600dpi).

Some important distinctions between previous work and the systems and tools presented in this paper include:

1. Our system focuses solely on formula extraction and its evaluation.
2. Grammars and manual segmentation are not required (e.g., per [2])
3. Recognition is graph-based, with outputs generated more quickly and with more easily diagnosed errors than RNN models (e.g., [15]).
4. Symbol information from born-digital PDFs is used directly; where absent, characters are recognized from connected components (CCs) in images.
5. Structure recognition errors can be directly observed in graphs grounded in input image regions (e.g., CCs), and the LgEval library [13,20,22,10] has been extended to visualize errors both in isolation, and in their page contexts.

3 Formula Detection Pipeline Components

We describe the components of our formula processing pipeline in this Section. As seen in Fig. 2, the extraction pipeline has three major components: 1) symbol extraction from PDF, 2) formula detection, and 3) formula structure recognition (parsing). The final outputs are Symbol Layout Trees (SLTs) corresponding to each detected mathematical expression in the input PDF documents, which we visualize as both graphs and using Presentation MathML (see Fig. 5).

3.1 SymbolScraper: Symbol Extraction from PDF

Unless formula images are embedded in a PDF file (e.g., as a .png), born-digital PDF documents provide encoded symbols directly, removing the need for character recognition [2]. In PDF documents, character locations are represented by

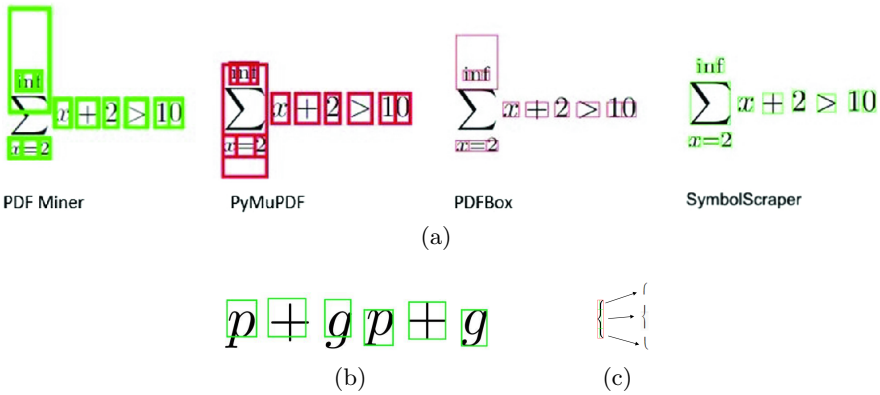


Fig. 3. Symbol extraction from PDF. (a) Symbol locations in a PDF formula from various tools. (b) correcting bounding box translations from glyph data. (b) Compound character: a large brace ('{') drawn with three characters.

their position on writing lines and in 'words,' along with their character codes and font parameters (e.g., size). Available systems such as PDFBox, PDF Miner, and PyMuPDF return the locations of characters using the drawing commands provided in PDF files directly. However, they return inconsistent locations (see Fig. 3(a))(a).

We resolve this problem by intercepting the rendering pipeline, and using the character outlines (*glyphs*) provided in embedded font profiles in PDF files. Glyphs are vector-based representations of how characters are drawn using a series of lines, arcs, and 'pen' up/down/move operations. These operations detail the outlines of the lines that make up the character. Glyphs are represented by a set of coordinates with *winding rules* (commands) used to draw the outline of a character. The glyph along with font scaling information is used to determine a character's bounding box and relative position in the PDF. Our symbol extractor extends the open-source PDFBox [1] library from Apache.

Character Bounding Boxes. For high-quality page rendering, individual squares used to represent the character outlines (i.e., glyphs) are known as 'em squares.' The em square is typically 1000 or 2048 pixels in length on each side, depending upon the font type used and how it is rendered.[2] Glyphs are represented by vector drawing commands in the em square 'glyph space,' which has a higher resolution (e.g., a 1000 x 1000) than that used to render the character in page images (in 'text space'). To obtain the bounding box for a character on a page, we need to: 1) convert glyph outline bounding boxes in glyph space to a bounding box in 'text space'(i.e. page coordinates), and 2) translate the box based on the *varying* origin location within each glyph's em square representation. The difference in origin locations by glyph reflects the different writing

[2] You can see the various ways we obtain the em square value by looking at the `getBoundingBox()` method in our `BoundingBox` class

line positions for symbols, and allows the whole box to be used in representing character shapes (maximizing their level of detail/resolution).

Glyph outlines are translated to page coordinates using character font size and type information. A scaling matrix for the character, defined by the font size, text scaling matrix, and current transformation matrix is used for the conversion. PDF files provide a baseline starting point coordinate (b_x, b_y) for characters on writing lines, but this provided coordinate does not take character kerning, font-weight, ascent, descent, etc., into account, as seen in Fig. 3(a). To produce the appropriate translations for character bounding boxes, we use the textline position of the character provided within the em square for each glyph (in glyph space). We use displacement (δ_x, δ_y) of the bottom left coordinate within the glyph em square to obtain the corrective translation. Next, we scale the glyph space coordinates to text space coordinates and then use coordinate list G from the character's glyph, and find the width and height using the minimum and maximum coordinates of G.

The extracted character bounding boxes are very precise even for non-latin characters (e.g., $\sum$) as seen in Fig. 3(a) at far right. Likewise, the recognized symbol labels are accurate. Some documents contain characters embedded in images, in which case these characters must be identified in downstream processing. We currently use font tables provided by PDFBox for mapping characters to glyphs. Using these font tables, we have been able to retrieve glyphs for the following font families: TrueType, Type 1, Type 2, and CID. Font information is extracted using the `getFont()` method in the PDFBox `TextPosition` class. There are cases where the bounding boxes do not align perfectly with characters, due to the use of rarer font types not handled by our system (e.g., Type 3, which seems to be commonly used for raster fonts in OCR output). We hope to handle Type 3 in the future. a rare type of fonts, not handled by the current

Compound Characters. Compound characters are formed by two or more characters as shown for a large brace in Fig. 3(c). For simplicity, we assume that bounding boxes for sub-character glyphs intersect each other, and merge intersecting glyphs into a single character. As another common example, square roots are represented in PDF by a radical symbol and a horizontal line. To identify characters with multiple glyphs, we test for bounding box intersection of adjacent characters, and intersecting glyphs are merged into a compound character. The label *unknown* is assigned to compound characters other than roots, where the radical symbol is easily identified.

Horizontal Lines. Horizontal lines are drawn as strokes and not characters. To extract them, we override the `strokepath()` method of the `PageDrawer` class in the PDFBox library [1]. By overriding this method, we obtain the starting and ending coordinates of horizontal lines directly. Some additional work is needed to improve the discrimination between different line types, and other vector-based objects could be extracted similarly (e.g., boxes, image BBs).

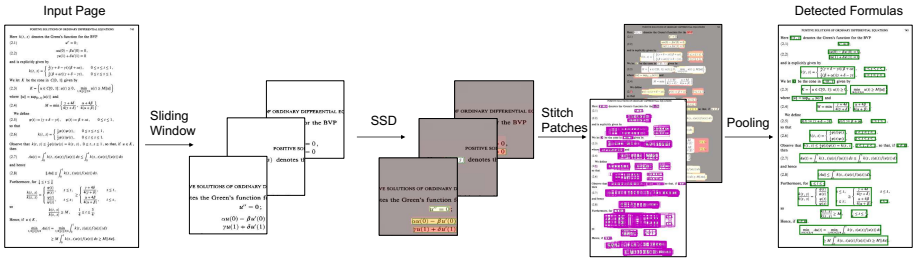


Fig. 4. ScanSSD architecture. Heatmaps illustrate detection confidences with $gray \approx 0$, $red \approx 0.5$, $white \approx 1.0$. Purple and green bounding boxes show formula regions after stitching window-level detections and pooling, respectively.

3.2 Formula Detection in Page Images

We detect formulas in document page images using the Scanning Single Shot Detector (ScanSSD [11]). ScanSSD modifies the popular Single Shot Detector (SSD [8]) to work with sliding windows, in order to process large page images (600 dpi). After detection is performed in windows, candidate detections are pooled, and then finally regions are selected by pixel-level voting and thresholding.

Fig. 4 illustrates the ScanSSD workflow. First, a sliding window samples overlapping page regions and applies a standard 512×512 Single-Shot Detector to locate formula regions. Non-Maximal Suppression (NMS) is used to select the highest confidence regions from amongst overlapping detections in each window. Formulas detected within each window have associated confidences, shown using colour in the 3rd stage of Fig. 4. As seen with the purple boxes in Fig. 4, many formulas are found repeatedly and/or split across windows.

To obtain page-level detections, we stitch window-level detections on the page, and then use a voting-based pooling method to produce output detections (green boxes in Fig. 4). Details are provided below.

Sliding Windows and SSD. Starting from a 600 dpi page image we slide a 1200×1200 window with a vertical and horizontal stride (shift) of 120 pixels (10% of window size). Windows are roughly 10 text lines in height. The SSD is trained using ground truth math regions cropped at the boundary of each window, after scaling and translating formula bounding boxes appropriately.

There are four main advantages to using sliding windows. The first is data augmentation: only 569 page images are available in our training set, which is *very* small for training a deep neural network. Our sliding windows produce 656,717 sub-images. Second, converting the original page image directly to 300×300 or 512×512 loses a great deal of visual information, and detecting formulas using sub-sampled page images yielded low recall. Third, windowing provides multiple chances to detect a formula. Finally, Liu et al. [8] mention that SSD is challenged when detecting small objects, and formulas with just one or two characters are common. Using high-resolution sub-images increases the relative size of small formulas, making them easier to detect.

The original SSD [8] architecture arranges initial hypotheses for rectangular dections unfiromly over grids at multiple resolutions. At each point in a grid, aspect ratios (width:height) of $\{1/3, 1/2, 1, 2, 3\}$ are used for initial hypotheses, which are translated and resized during execution of the SSD neural network. However, many formulas have aspect ratios greater than 3, and so we also include the wider default boxes sizes used in TextBoxes [6], with aspect ratios of 5, 7, and 10. In our early experiments, these wider default boxes increased recall for large formulas.

The filtered windowed detections are post-processed to tightly fit the connected components that they contain or intersect. The same procedure is applied after pooling detections at the page level, which we describe next.

Page-Level Detection Pooling. SSD detections within windows are stitched together on the page, and then each detection region votes at the pixel level (see Fig. 4). Detections are merged using a simple pixel-level voting procedure, with the number of detection boxes intersecting each pixel used as a coarse detection confidence value. Other confidence values, such as the sum of confidences, average confidence, and maximum confidence produced comparable or worse results while being more expensive to compute.

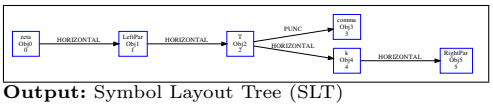
After voting, pixel region intersection counts are thresholded (using $t \geq 30$), producing a binary image. Connected components in the resulting image are converted to bounding boxes, producing the final formula detections. We then expand and/or shrink the detections so that they are cropped around the connected components they contain and touch at their border. The goal is to capture entire characters belonging to a detection region, without additional padding. Before producing final results, we discard large graphical objects like tables and charts, using a threshold for the ratio of height and area of the detected graphical objects to that of the document page.

Identifying Extracted Symbols in Formula Regions. We identify overlapping regions between detected formula regions and extracted symbol bounding boxes, discarding symbols that lie outside formula regions as shown in Fig. 1(d). We then combine formula regions and the symbols they contain, and write this to tab-separated variable (TSV) file with a hierarchical structure. The final results of formula detection and symbol extraction are seen in Fig. 1(d).

3.3 Recognizing Formula Structure (Parsing)

We use QD-GGA [9] for parsing the detected mathematical formulas, which involves recognizing a hierarchical graph structure from the expression images. In these graphs, symbols and unlabeled connected components act as nodes, and spatial relationships between symbols are edges. The set of graph edges under consideration are defined by a line-of-sight (LOS) graph computed over connected components. The symbol bounding boxes and labels produced by SymbolScraper are used directly in extraction results, where available (see Fig. 5), avoiding the need for character-level OCR. Otherwise, we use CCs extraction and allow QD-GGA [9] to perform symbol recognition (segmentation and classification).

Input: image w. extracted chars. (BBs shown)



Output: Symbol Layout Tree (SLT)

```
\(\zeta\left( {{T,}\left. k \right)} \right.\)
```

SLT in LaTeX

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
<mrow>
    <mi xml:id="0:">ζ</mi>
    <mrow>
        <mo xml:id="1:">(</mo>
        <mrow>
            <mrow>
                <mi xml:id="2:">T</mi>
                <mo xml:id="3:">,</mo>
            </mrow>
            <mrow>
                <mi xml:id="4:">k</mi>
                <mo xml:id="5:">)</mo>
            </mrow>
        </mrow>
    </mrow>
</mrow>
</math>
```

SLT in Presentation MathML

Fig. 5. Parsing a formula image. Formula regions are rendered and have characters extracted when they are provided in the PDF. QD-GGA produces a Symbol Layout Tree as output, which can be translated to LaTeX and Presentation MathML.

Symbol segmentation and classification are decided first, using provided locations and labels where available, and for connected components in the LOS graph defined by binary 'merge' relationships between image CCs, and then choosing the symbol class with the highest average label confidence across merged CCs. A Maximum Spanning Tree (MST) over detected symbols is then extracted from the weighted relationship class distributions using Edmond's arborescence algorithm [5] to produce the final formula interpretation.

QD-GGA trains CNN-based features and attention modules concurrently for multiple tasks: symbol classification, edge classification, and segmentation of connected components into symbols. A graph-based attention module allows multiple classification queries for nodes and edges to be computed in one feed-forward pass, resulting in faster training and execution.

The output produced is an SLT graph, containing the structure as well as the symbols and spatial relationship classifications. The SLT may be converted into Presentation MathML or a LaTeX string (see Fig. 5).

4 Results for Pipeline Components

We provide preliminary results for each pipeline component in this section.

SymbolScraper. 100 document pages were randomly chosen from a collection of 10,000 documents analyzed by SymbolScraper. Documents were evaluated based on the percentage of detected characters that had correct bounding boxes. 62/100 documents had all characters detected without error. An additional 9 documents had over 80% of their characters correctly boxed, most with just one or two words with erroneous bounding boxes. 6 documents were not born-digital (producing no character detections); 10 were OCR'd documents, represented entirely in Type 3 fonts that SymbolScraper could not process, and 9 failed due to a bug in the image rendering portion of the symbol extractor. The remaining 4 documents had fewer than 80% of their character bounding boxes correctly detected.

Table 1. Formula Detection Results for TFD-ICDAR2019

	IOU ≥ 0.75			IOU ≥ 0.5		
	Precision	Recall	F-score	Precision	Recall	F-score
ScanSSD	**0.774**	**0.690**	**0.730**	**0.851**	**0.759**	**0.802**
RIT 2	0.753	0.625	0.683	0.831	0.670	0.754
RIT 1	0.632	0.582	0.606	0.744	0.685	0.713
Mitchiking	0.191	0.139	0.161	0.369	0.270	0.312
Samsung‡	0.941	0.927	0.934	0.944	0.929	0.936

‡ Used character information

For documents with more than 80% of their text properly extracted, the number of pages with character extractions without error is 94.7% (71 of 75). We compare our SymbolScraper tool against PDFBox, PDFMiner and PyMuPDF. As shown in Fig. 3(a), only SymbolScraper recovers the precise bounding boxes directly from the PDF document. SymbolScraper is much faster than other techniques that obtain character bounding boxes using OCR or image processing. In our preliminary experiments, running SymbolScraper using a single process on a CPU takes around 1.7 seconds per page on a laptop computer with modest resources.

ScanSSD. The top of Table 1 shows formula detection results obtained by ScanSSD along with participating systems in the ICDAR 2019 Typeset Formula Detection competition [10]. Systems are evaluated using a coarse detection threshold with $IoU \geq 50\%$, and a fine detection threshold with $IoU \geq 75\%$. We use the implementation provided by the competition [10] to calculate intersections for box pairs, as well as precision, recall and f-scores.

ScanSSD outperforms all systems that did not use character locations and labels from ground truth in their detection system. Relative to the other image-based detectors, ScanSSD improves both precision and recall, but with a larger increase in recall scores for IOU thresholds of both 0.75 (fairly strict) and 0.5 (less strict). The degradation in ScanSSD performance between IOU thresholds is modest, losing roughly 7% in recall, and 7.5% precision, indicating that detected formulas are often close to their ideal locations.

Looking a bit more closely, over 70% of ground truth formulas are located at their ideal positions (i.e., with an IOU of 1.0). If one then considers the character level, i.e., the accuracy with which characters inside formulas are detected, we obtain an f-measure of 92.6%. The primary cause of the much lower formula detection rates are adjacent formulas merged across textlines, and splitting formulas at large whitespace gaps (e.g., for variable constraints). We believe these results are sufficiently accurate for use in prototyping formula search engines applied to collections of PDF documents. Examples of detection results are visible in Figs. 1 and 6.

Running the test set on a desktop system with a hard drive (HDD), 32GB RAM, an Nvidia RTX 2080 Ti GPU, and an AMD Ryzen 7 2700 processor, our PyTorch-based implementation took a total of 4hrs, 33 mins, and 31 seconds

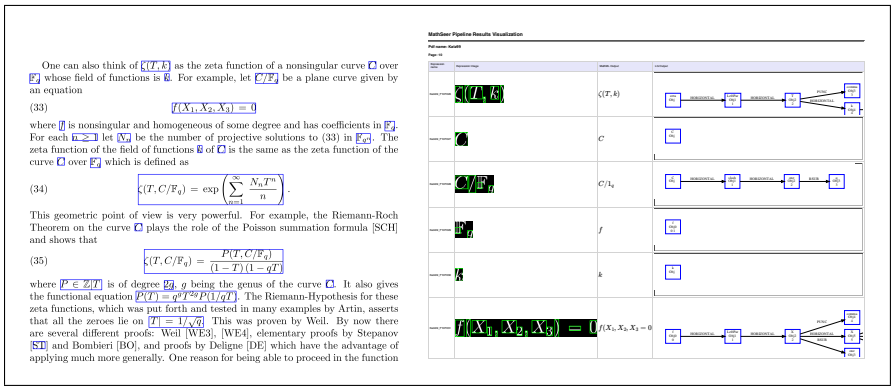


Fig. 6. HTML visualization for formulas extracted from a sample PDF page with detected formula locations (left), and a table (right) showing extracted formulas and recognition results as rendered MathML and SLT graphs.

to process 233 pages including I/O operations (average: 70.4 secs/page). While not problematic for early development, this is too slow for a large corpus. We identify possible accelerations in the conclusion (see Section 6).

When using an SSD on whole document pages in a single detection window (e.g., 512×512), very low recall is obtained because of the low resolution, and generally bold characters were detected as math regions.

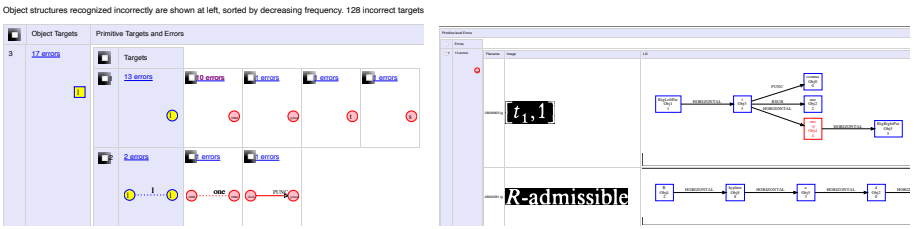
QD-GGA. We evaluated QD-GGA on InftyMCCDB-2[3], a modified version of InftyCDB-2 [17]. We obtain a structure recognition rate of 92.56%, and an expression recognition rate (Structure + Classification) of 85.94% on the test set (6830 images). Using a desktop system with a hard drive (HDD), 32GB RAM, an Nvidia GTX 1080 GPU, and an Intel(R) Core(TM) i7-9700KF processor, QD-GGA took a total of 26 mins, 25 seconds to process 6830 formula images (average: 232 ms/formula).

5 New Tools for Visualization and Evaluation

An important contribution of this paper is new tools for visualizing recognition results and structure recognition errors, which are essential for efficient analysis and error diagnosis during system development and tuning. We have created these in the hopes of helping both ourselves and others working in formula extraction and other graphics extraction domains.

Recognition Result Visualization (HTML + PDF). We have created a tool to produce a convenient HTML-based visualization of the detection and recognition results, with inputs and outputs of the pipeline for each PDF document page. For each PDF page, we summarise the results in the form of a table, which contains the document page image with the detected expressions at the

[3] https://zenodo.org/record/3483048#.XaCwmOdKjVo



(a) (b)

Fig. 7. Error analysis (errors shown in red). (a) Main error table organized by decreasing frequency of errors. (b) Specific instances where 'l' is misclassified as 'one,' seen after clicking on the '10 errors' link in the main table.

left and a scrollable (horizontally and vertically) sub-table with each row showing the expression image, the corresponding MathML rendered output, and the SLT graphs (see Fig. 6).

We created HTMLs for easy evaluation of the entire pipeline, to identify component(s) (symbol extraction, expression detection, or structure recognition) producing errors on each page. This makes it easier to diagnose the causes of errors, identify the strengths and weaknesses of different components, and improve system designs.

Improved Error Visualization. The LgEval library [13,14] has been used for evaluating formula structure recognition both for recognition over given input primitives (e.g., strokes) and for output representations without grounding in input primitives [10] . We extend LgEval's error summary visualization tool, `confHist`, to allow viewing all instances of specific errors through HTML links. These links take the user directly to a list of formulas containing a specific error. The tool generates a table showing all error types for ground truth SLT subtrees of a given size, arranged in rows and sorted by frequency of error, as shown in Fig. 7(a) (for single symbols). Each row contains a sub-table with all primitive level target and error graphs, with errors shown in red. Errors include missing relationships and nodes, segmentation errors, and symbol and relationship classification errors - in other words, all classification, segmentation, and relationship errors.

New links in the error entries of the HTML table open HTML pages, containing all formulas sharing a specific error along with additional details arranged in a table. This includes all expression images containing a specific error along with their SLT graph showing errors highlighted in red Fig. 7(b). The new tool helps to easily identify frequent recognition errors in the contexts where they occur. For example, as seen in Fig. 7(b), we can view all expression images in which 'l' is misclassified as 'one' by clicking the error entry link (10 errors) in Fig. 7(a) and locate the incorrect symbol(s) using the SLT graphs.

Both visualization tools load very quickly, as the SLT graphs are represented in small PDF files that may be scaled using HTML/javascript widgets.

6 Conclusion

We have presented a new open-source formula extraction pipeline for PDF documents, and new tools for visualizing recognition results and formula parsing errors. SymbolScraper extracts character labels and locations from PDF documents accurately and quickly by intercepting the rendering pipeline and using character glyph data directly. The new ScanSSD formula detector identifies formulas with sufficient accuracy for prototyping math-aware search engines, and very high accuracy at the character level. The existing QD-GGA system is used to parse formula structure in detected formula regions.

In the future, we would like to design an end-to-end trainable system for both formula detection and parsing, and also extend our system to handle more PDF character encodings (e.g., Type 3 fonts). Future work for SymbolScraper includes improved filtering of horizontal bars, integrating compound characters classification within the tool, and faster implementations (e.g., using parallelization). For detection using ScanSSD, we are looking at ways to accelerate the non-maximal suppression algorithm and fusion steps, and to improve the page-level merging of windowed detections to avoid under-segmenting formulas on adjacent text lines, and over-merging of formulas with large whitespace gaps (e.g., caused by variable constraints to the right of a formula).

We believe that our pipeline and tools will be useful for others working on extracting formulas and other graphics type from documents. Our system will be available as open source before the conference.

Acknowledgements

A sincere thanks to all the students who contributed to the pipeline's development: R. Joshi, P. Mali, P. Kukkadapu, A. Keller, M. Mahdavi and J. Diehl. Jian Wu provided the document collected used to evaluate SymbolScraper. This material is based upon work supported by the Alfred P. Sloan Foundation under Grant No. G-2017-9827 and the National Science Foundation (USA) under Grant Nos. IIS-1717997 (MathSeer project) and 2019897 (MMLI project).

References

1. Apache: PDFBOX - a Java PDF library, https://pdfbox.apache.org/
2. Baker, J.B., Sexton, A.P., Sorge, V.: A Linear Grammar Approach to Mathematical Formula Recognition from PDF. In: Carette, J., Dixon, L., Coen, C.S., Watt, S.M. (eds.) Intelligent Computer Mathematics. pp. 201–216. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02614-0_19
3. Davila, K., Joshi, R., Setlur, S., Govindaraju, V., Zanibbi, R.: Tangent-V: Math Formula Image Search Using Line-of-Sight Graphs. In: Azzopardi, L., Stein, B., Fuhr, N., Mayr, P., Hauff, C., Hiemstra, D. (eds.) Advances in Information Retrieval. pp. 681–695. Lecture Notes in Computer Science, Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-15712-8_44

4. Deng, Y., Kanervisto, A., Ling, J., Rush, A.M.: Image-to-markup generation with coarse-to-fine attention. In: Precup, D., Teh, Y.W. (eds.) Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017. Proceedings of Machine Learning Research, vol. 70, pp. 980–989. PMLR (2017), http://proceedings.mlr.press/v70/deng17a.html
5. Edmonds, J.: Optimum branchings. Journal of Research of the National Bureau of Standards Section B Mathematics and Mathematical Physics **71B**(4), 233 (Oct 1967). https://doi.org/10.6028/jres.071B.032, https://nvlpubs.nist.gov/nistpubs/jres/71B/jresv71Bn4p233_A1b.pdf
6. Liao, M., Shi, B., Bai, X., Wang, X., Liu, W.: Textboxes: A fast text detector with a single deep neural network. In: Thirty-First AAAI Conference on Artificial Intelligence (2017)
7. Lin, X., Gao, L., Tang, Z., Lin, X., Hu, X.: Mathematical Formula Identification in PDF Documents. In: 2011 International Conference on Document Analysis and Recognition. pp. 1419–1423 (Sep 2011). https://doi.org/10.1109/ICDAR.2011.285, iSSN: 2379-2140
8. Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.Y., Berg, A.C.: SSD: Single shot multibox detector. In: European conference on computer vision. pp. 21–37. Springer (2016)
9. Mahdavi, M., Sun, L., Zanibbi, R.: Visual Parsing with Query-Driven Global Graph Attention (QD-GGA): Preliminary Results for Handwritten Math Formula Recognition. In: 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW). pp. 2429–2438. IEEE, Seattle, WA, USA (Jun 2020). https://doi.org/10.1109/CVPRW50498.2020.00293, https://ieeexplore.ieee.org/document/9150860/
10. Mahdavi, M., Zanibbi, R., Mouchere, H., Viard-Gaudin, C., Garain, U.: ICDAR 2019 CROHME + TFD: Competition on Recognition of Handwritten Mathematical Expressions and Typeset Formula Detection. In: 2019 International Conference on Document Analysis and Recognition (ICDAR). pp. 1533–1538. IEEE, Sydney, Australia (Sep 2019). https://doi.org/10.1109/ICDAR.2019.00247, https://ieeexplore.ieee.org/document/8978036/
11. Mali, P., Kukkadapu, P., Mahdavi, M., Zanibbi, R.: ScanSSD: Scanning Single Shot Detector for Mathematical Formulas in PDF Document Images. arXiv:2003.08005 [cs] (Mar 2020), http://arxiv.org/abs/2003.08005, arXiv: 2003.08005
12. Mouchère, H., Viard-Gaudin, C., Zanibbi, R., Garain, U.: ICFHR2016 CROHME: Competition on Recognition of Online Handwritten Mathematical Expressions. In: 2016 15th International Conference on Frontiers in Handwriting Recognition (ICFHR). pp. 607–612 (Oct 2016). https://doi.org/10.1109/ICFHR.2016.0116, iSSN: 2167-6445
13. Mouchère, H., Viard-Gaudin, C., Zanibbi, R., Garain, U., Kim, D.H., Kim, J.H.: ICDAR 2013 CROHME: Third International Competition on Recognition of Online Handwritten Mathematical Expressions. In: 2013 12th International Conference on Document Analysis and Recognition. pp. 1428–1432 (Aug 2013). https://doi.org/10.1109/ICDAR.2013.288, iSSN: 2379-2140
14. Mouchère, H., Zanibbi, R., Garain, U., Viard-Gaudin, C.: Advancing the state of the art for handwritten math recognition: the CROHME competitions, 2011–2014. International Journal on Document Analysis and Recognition (IJDAR) **19**(2), 173–189 (Jun 2016). https://doi.org/10.1007/s10032-016-0263-5, https://doi.org/10.1007/s10032-016-0263-5

15. Phong, B.H., Dat, L.T., Yen, N.T., Hoang, T.M., Le, T.L.: A deep learning based system for mathematical expression detection and recognition in document images. In: 2020 12th International Conference on Knowledge and Systems Engineering (KSE). pp. 85–90 (Nov 2020). https://doi.org/10.1109/KSE50997.2020.9287693, iSSN: 2164-2508
16. Sorge, V., Bansal, A., Jadhav, N.M., Garg, H., Verma, A., Balakrishnan, M.: Towards generating web-accessible STEM documents from PDF. In: Proceedings of the 17th International Web for All Conference. pp. 1–5. W4A '20, Association for Computing Machinery, New York, NY, USA (Apr 2020). https://doi.org/10.1145/3371300.3383351, http://doi.org/10.1145/3371300.3383351
17. Suzuki, M., Uchida, S., Nomura, A.: A ground-truthed mathematical character and symbol image database. In: Eighth International Conference on Document Analysis and Recognition (ICDAR'05). pp. 675–679 Vol. 2 (2005). https://doi.org/10.1109/ICDAR.2005.14
18. Suzuki, M., Tamari, F., Fukuda, R., Uchida, S., Kanahori, T.: INFTY: an integrated OCR system for mathematical documents. In: Proceedings of the 2003 ACM symposium on Document engineering. pp. 95–104. DocEng '03, Association for Computing Machinery, New York, NY, USA (Nov 2003). https://doi.org/10.1145/958220.958239, http://doi.org/10.1145/958220.958239
19. Suzuki, M., Yamaguchi, K.: Recognition of E-Born PDF Including Mathematical Formulas. In: Miesenberger, K., Bühler, C., Penaz, P. (eds.) Computers Helping People with Special Needs. pp. 35–42. Lecture Notes in Computer Science, Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-41264-1_5
20. Zanibbi, R., Pillay, A., Mouchere, H., Viard-Gaudin, C., Blostein, D.: Stroke-Based Performance Metrics for Handwritten Mathematical Expressions. In: 2011 International Conference on Document Analysis and Recognition. pp. 334–338 (Sep 2011). https://doi.org/10.1109/ICDAR.2011.75, iSSN: 2379-2140
21. Zanibbi, R., Blostein, D.: Recognition and retrieval of mathematical expressions. International Journal on Document Analysis and Recognition (IJDAR) **15**(4), 331–357 (Dec 2012). https://doi.org/10.1007/s10032-011-0174-4, https://doi.org/10.1007/s10032-011-0174-4
22. Zanibbi, R., Mouchère, H., Viard-Gaudin, C.: Evaluating structural pattern recognition for handwritten math via primitive label graphs. In: Document Recognition and Retrieval XX. vol. 8658, p. 865817. International Society for Optics and Photonics (Feb 2013). https://doi.org/10.1117/12.2008409
23. Zanibbi, R., Orakwue, A.: Math Search for the Masses: Multimodal Search Interfaces and Appearance-Based Retrieval. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) Intelligent Computer Mathematics. pp. 18–36. Lecture Notes in Computer Science, Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-20615-8_2
24. Zhang, X., Gao, L., Yuan, K., Liu, R., Jiang, Z., Tang, Z.: A Symbol Dominance Based Formulae Recognition Approach for PDF Documents. In: 2017 14th IAPR International Conference on Document Analysis and Recognition (ICDAR). vol. 01, pp. 1144–1149 (Nov 2017). https://doi.org/10.1109/ICDAR.2017.189, iSSN: 2379-2140