Handwritten Math Expression Recognition using a Recursive Approach

Bhuta, Bhavin
bsb5375@rit.edu

Chauhan, Dhaval
dmc8686@rit.edu

1 Design

We are using a parsing algorithm that recursively builds a layout tree through deconstruction of the math expression into small and simple sub-expressions. The recursive algorithm keeps breaking the expression until there are only either one, or two symbols left in the sub-expression, regardless of them making any sense. Our spatial relationship classifier then determines the relationship between the roots of the sub-expressions and the two symbols of the sub-expressions and saves it in Object-Relationship format used in Label-Graph format of [1].

Our motivation behind the use of this approach was to cover variety of different types of expressions given in the CROHME dataset [1]. Most algorithms work well when the expression is small and linear. So our idea of recognizing math expression is to recognize and segment out several small and simple sub-expressions in it, and then combine them together to get layout tree of the whole expression. Our idea is better explained using an example expression from CROHME dataset shown in figure (1).

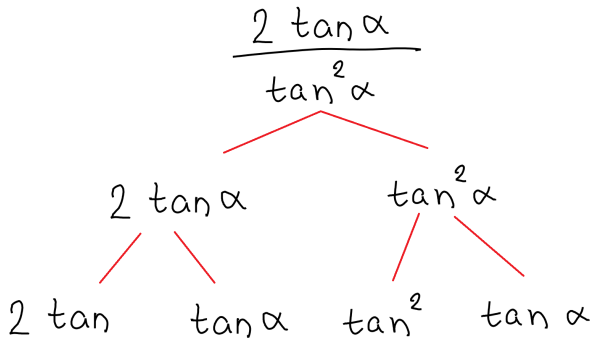

Figure 1: Construction

In this example, the fraction line symbol becomes the root of the tree and the symbols in numerator $2tan\alpha$, and denominator $tan^2\alpha$ become two sub-expressions. Our recursively algorithm then breaks these sub-expressions into small more sub-expressions. Expressions in CROHME dataset include 6 spatial relations {"*Right*","*Above*","*Below*","*Sup*","*Sub*","*Inside*"}, all of which can contain sub-expressions. If we successfully identify the sub-expressions, and their root symbols, we can get an accurate layout tree of the expression. Once the structure is obtained, our spatial relationship classifier, which is a trained Random Forest with 50 trees and 50 depth, will predict relationship labels of the tree edges.

2 Preprocessing and Features

2.1 Classification

Our features and preprocessing for classification of symbols are from [2] and [3], respectively. No changes were made to our classifier from project 1.

2.2 Segmentation

We are using our non-baseline segmenter from project 2 without any changes. It uses distance between averaged centers of the strokes from [4] as the sole feature to decide merging or splitting of them. We are using a naive binary classification function instead of a classifier model just like in project 2. We tried to replace that function with a classifier but we encountered a Memory error while trying to train it. We believe that it is because of high stroke pair count in the dataset. Similar thing happened when we tried to train our Random Forest Classifier for bonus in project 1 when we included junk symbols in the training set. Back then, there were around 170,000 symbol INKML files in total, including valid and junk symbols both. Whereas, this time we had over 400,000 total stroke pairs. For this reason, we kept our binary function and used it for merge/split decision making.

Our flow of preprocessing is different from preprocessing of classification, although, the choice of techniques are the same. For segmentation and parsing, we start with removal of duplicate points and then move on to smoothing and resampling of new datapoints without normalization. Normalizing two stroke/symbol pairs separately would result in loss of spatial relational data between the two. Therefore, we normalize only after we have our segmentation of symbols ready and before classification of those symbols.

2.3 Parsing

Our parser uses bounding box vector angles of the bounding boxes of the two symbols. Figure (2) explains this better. We connect the respective corner points, midpoints, and the center of the two bounding boxes and use the slope of the vectors scaled between -180 degree to 180 degree as features.

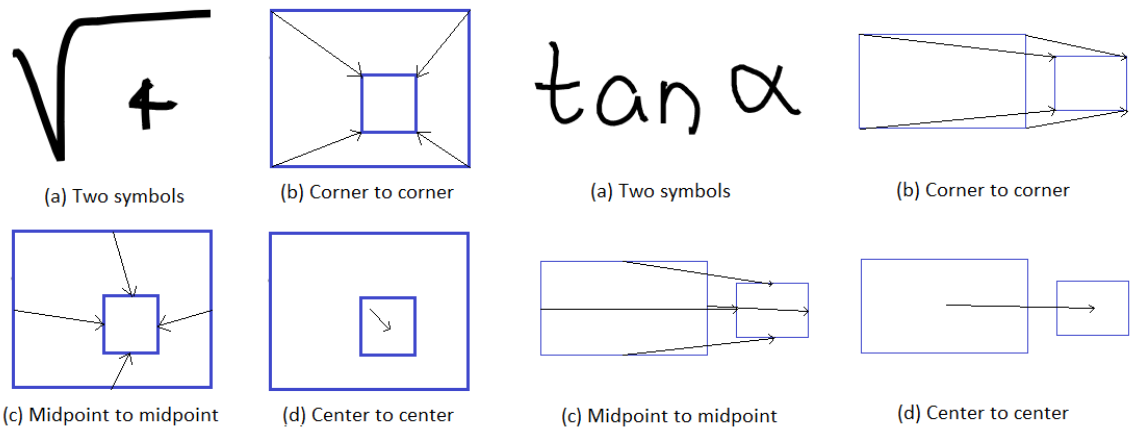

Figure 2: Bounding Box Vector Angle Features

Figure (a) of figure (2) shows the actual two symbols. In $\sqrt{4}$, $\sqrt{}$ is the source symbol and 4 is the sink symbol. In $tan\alpha$, tan is the source symbol and α is the sink symbol. In (b), respective corner points of the two symbols' bounding boxes are connected with a vector starting

from the source symbol to the sink symbol. With 4 corner points of the bounding boxes we get 4 angle values as our first 4 features.In (c), respective edge midpoints of the two symbols' bounding boxes are connected. With 4 edges and its midpoints, we get 4 angle values as our other 4 features. In (d), respective bounding box centers of the two symbols' bounding boxes are connected, and the slope of the vector is calculated. This results in a total of 9 features for spatial relationship classification.

We decided to choose this as feature, because, for each relationship from {"*Right*","*Above*", "*Below*","*Sup*","*Sub*","*Inside*"}, these 9 angles combined together give a unique pattern of combination. For example, in our figure (2) example of $\sqrt{4}$, where $\surd$ has a relation of "*Inside*" with 4, corner and midpoint vectors seem to converge to a point inside the box of the source symbol i.e. $\surd$ in this case. Whereas, vectors from $tan\alpha$, where tan has relation "*Right*" with α, tend to converge to a point that is outside both the bounding boxes and in *east* direction. Similarly, vectors in "*Above*" relation will tend to converge in the *north* direction, "*Below*" in the *south*, "*Sup*" in $north - east$, and "*Sub*" in $south - east$. It is not necessary that these vectors converge. When the sink bounding box is larger than the source bounding box, these vectors will appear to diverge from one another. However, their angles and directions will be similar. We are training a Random Forest model with 50 estimators and 50 max depth with these features. We kept estimators and depth as 50 after grid searching with different values. Although the difference in accuracies were quite insignificant at around 1%.

3 Parsers

3.1 Baseline Parsers

We developed and implemented around 4 different parsers that were built on top of one another in an attempt to accurately recognize few different types of expressions. We do not have a parser that can work on input primitives i.e. strokes. It expects symbols as input. However, we have used our segmenter and classifier from previous projects to get predicted symbols from strokes and then use them as input to our parser/s for expression recognition from strokes.

Our initial (baseline) parser was based on a simple algorithm that sorted all the symbols from minimum X values to maximum X values and then constructed a tree by connecting two consecutive symbols in the sorted list. This parser was successfully able to recognize expressions that were linear in nature, i.e. only had "*Right*" relations or had other relations only at the end as shown in example expressions in figure (3). However, this algorithm failed to correctly recognize expressions with complicated structural layout like the one shown in figure (1) earlier.

Figure 3: Example expressions that baseline parser recognized correctly.

We then implemented a stack based algorithm on top of it to try to capture linear expressions that had multiple instances of superscripts and subscripts in between. With correct prediction of spatial relationship between the symbols, this parser was able to recognize expressions like the one shown in figure (4). Just like our baseline parser, this algorithm would work on list of sorted symbols. It would check for spatial relationship between the consecutive symbols first. If the relation between two symbols was not "*Right*", then it would save the first symbol along with the relation it made, in the stack and continue its relationship prediction on the consecutive pairs until an inverse relationship of the pair is obtained. Inverse relation is opposite of a label in spatial layout. For example, inverse of "*Sup*" is "*Sub*", and "*Above*" is "*Below*"

and vice-versa. If an inverse relation was obtained, we would pop the topmost symbol from the stack and pair it with the current second symbol with "*Right*" relation.

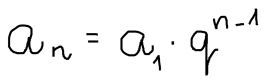

Figure 4: Example expression that stack based parser recognized correctly.

We ended up making a few different versions of the stack based approach. These parsers only had average performance, since, they were failing at generating correct layout tree for expressions that included relations from {"*Above*","*Below*","*Inside*"}.

3.2 Recursion based Parser

It seemed futile to carry on with the previous baseline approaches if our goal was to correctly recognize variety of expressions. We then decided to try and break expressions down into several small sub-expressions and then constructing a layout tree by connecting roots of the sub-expressions. Recursively breaking sub-expressions as well ensured that relationship decisions were made using local context. The algorithm uses list of symbols sorted on minimum X co-ordinate value. The first(left-most) symbol is considered the root and the following tree structure is built on top of it. The algorithm is summarized in the pseudo code shown below. Our main function in Algorithm (1), sorts the symbols (line 2) and for each pair of connected root symbols, predicts the relation, and saves the relation (line 8-12). For every root symbol and its leaf symbols, the algorithm saves the expression in the list of sub-expressions, and, after the end of the loop, sends the sub-expressions to the recursive function for further breakdown.

Algorithm 1 Recursion based Parser - Main function

```
1:  procedure ExpressionParser(unsortedSymbols)
2:      symbols = sort_min_X(unsortedSymbols)
3:      all_relations = [ ] list of relations in OR format
4:      sub_expressions = [[ ]] list of sub-expressions
5:      for <all symbol in symbols> do
6:          symbol_1 = symbol
7:          symbol_2 = nextsymbol
8:          feature_vector = getBoundingBoxFeatures(symbol, nextsymbol)
9:          relation = predictRelation(classifierModel, feature_vector)
10:         if relation is not "Right" then
11:             sub_expressions.lastElement().add(nextsymbol)
12:         else all_relations.add(relation) and sub_expressions.add([ ])
13:     all_relations = Recursive_ExpressionParser(sub_expressions)
14:     return all_relations
```

The recursive function in Algorithm 2 does almost the same things that the main function does. The only difference between the two is lack of sorting, (since the symbols in sub-expressions are already sorted), and addition of a base case scenario to avoid infinite looping of the algorithm. Base case checks the number of symbols in the sub-expression. If number is less than 2, it considers the sub-expression as parsed and returns the input list of relations without any change.

Algorithm 2 Recursion based Parser - Recursive function

```
procedure RECURSIVE_EXPRESSIONPARSER(sub_expressions, all_relations)
    sub_exps = [[ ]] list of sub-expressions
    for <all expressions in sub_expressions> do
        if Base Case: length(expressions) < 2 then
            return all_relations
        for <all symbol in expressions> do
            symbol_1 = symbol
            symbol_2 = nextsymbol
            feature_vector = getBoundingBoxFeatures(symbol, nextsymbol)
            relation = predictRelation(classifierModel, feature_vector)
            if relation is not "Right" then
                sub_exps.lastElement().add(nextsymbol)
            else all_relations.add(relation) and sub_exps.add([ ])
    all_relations = Recursive_ExpressionParser(sub_exps)
```

4 Results and Discussion

We have used the same test-train data split from project 2 for evaluating results for our parser. For stroke level parsing, we are performing segmentation and classification from previous projects to obtain symbols, and after that we make use of our recursion based parser for getting complete expression.

Test		Train	
Symbol	Stroke	Symbol	Stroke
94.25	94.66	95.16	95.38

Table 1: Relations accuracy of Spatial Relationship Classifier using Bounding Box Vectors

Figure 5 shows the results we observed through this exercise. Almost similar corresponding values of different measures shows that our relationship classifier('Above', 'Below', etc) performs pretty good. Hence, the bounding box vector angle features proved to be quite informative feature for determining relation between a pair of symbol. This conclusion is supported with high values of overall recognition rates of spatial relationship classifier observed in Table 1.

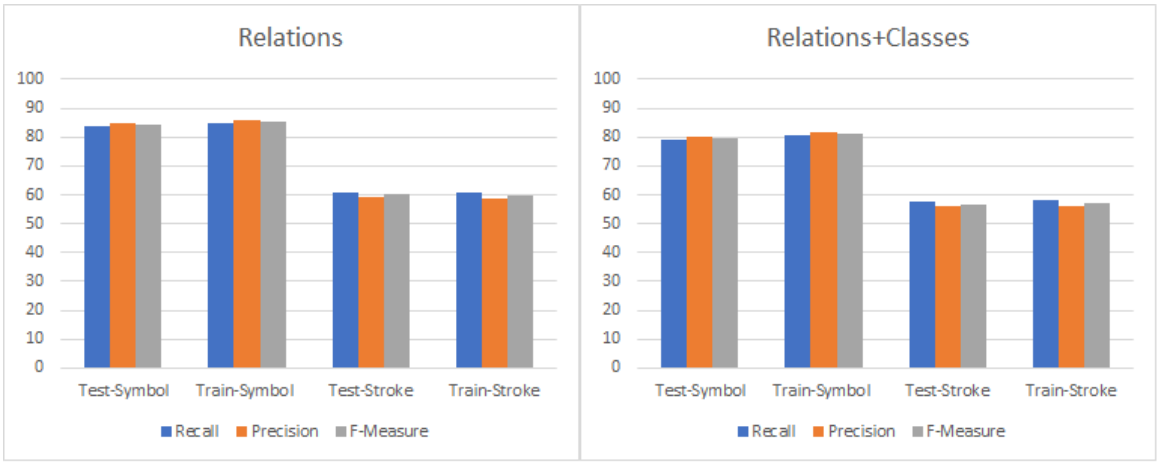

Figure 5: Recall, Precision, F-measure for Relations and Relations+Classifier

Figure 6 shows that on having segmented symbols with us, our parser does a fine job. Table 2 shows the actual parsing rate of symbols we observed which is relatively higher as compared to parsing rate over strokes. The low expression rate using stroke data is low due to cascading errors. We guess our classifier did a fair job in classifying the symbols, though lower-case, upper-case letters were major errors we observed. Then coming to segmenter, we think that this is the weakest part of our Expression Recognizer. Table 2 supports this claim of ours. Finally coming to our parser, we believe bounding box vector would fail in cases particularly when you would have ascender with subscript relation or when descender is with superscript relation. Also the general nature of writing and superscript and subscript is different as in, when a person writes subscript, he/she will write it quite close to the parent symbol whereas for superscript the blank space will be greater. We suspect this could have been one of the reasons why we observed more subscript relationship errors. Also, punctuation hasn't been friendly with us. Visually, punctuations like '.' and ',' appear in subscript, but, the label that it gets with its preceding symbol is the "*Right*" in our dataset. This creates a huge overlap between the data points of the two labels of our relationship classifier's feature space. Maybe using grammar-based context could have had a positive impact on our parser.

Symbol level	Stroke level
45.72	10.03

Table 2: Overall Recognition rates over Test Set Expressions

Figure 6: Average expression rates over structure and relations

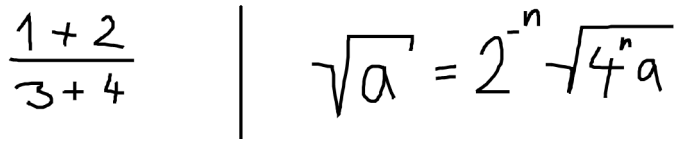

Figure 7: Correctly recognized expressions, using ground truth symbols

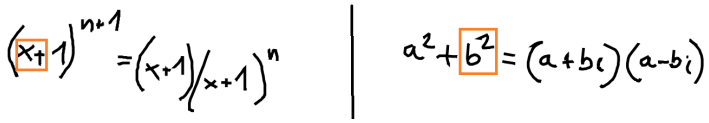

Figure 8: Error in comprehending Subscript And Superscript Relation

Figure 7 is an example of positive cases when complex inside, above/below were deciphered correctly by our design. Figure 8 contains a family of errors we observed by using the confHist tool. In figure 7, 'x' is the source symbol and '+' is the sink symbol, we think the reason for it to give subscript relation is due to the reason if we observe carefully the + sign has a part of it which is below as compared to 'x'. Inorder for parser to comprehend it as a Right relationship the lower angular vectors had to correspond to close to 180 degrees. Hence cases particularly when you have an operand and operator which are not close to each others horizontal plane, we would have negative cases. In figure 8, we see errors which were predominantly observed for superscript relationship errors. Here, b has been related to 2 as Right relationship instead of superscript relation. The reason we suspect would be due to the midpoint angular vectors confusing particularly for ascender and symbols lying in superscript space.

5 References

[1] H. Mouchère, R. Zanibbi, U. Garain, C. Viard-Gaudin, "Advancing the state of the art for handwritten math recognition: the crohme competitions 2011–2014", IJDAR, pp. 1-17, 2016.

[2] B. Q. Huang, Y. Zhang, and M.-T. Kechadi, "Preprocessing techniques for online handwriting recognition," in Intelligent Text Categorization and Clustering. Springer, 2009, pp. 25–45.

[3] Davila, K., Ludi, S., and Zanibbi, R. Using Off-line features and synthetic data for online handwritten math symbol recognition. In Frontiers in Handwriting Recognition (ICFHR), 2014 14th International Conference on, IEEE (2014), 323–328.

[4] Hu, L., Zanibbi, R.: Segmenting handwritten math symbols using AdaBoost and multi-scale shape context features. In: Proceedings of International Conference Document Analysis and Recognition, pp. 1180–1184.