Segmentation and Classification of Symbols in Handwritten Math Expressions

Bhuta, Bhavin
bsb5375@rit.edu

Chauhan, Dhaval
dmc8686@rit.edu

1 Design

The strokes replicated in the INKML data files are not smooth. There are duplicate points in some cases and in most cases the strokes are jagged. To get cleaner stroke information, we remove consecutive duplicate points, resample the stroke points by fitting the original stroke points to a cubic spline function and then regenerate 40 new interpolated points using that function. Once the segmentation is obtained, we normalize the clean stroke points to fit them in range [-1, 1] keeping the original aspect ratio of the strokes. These pre-processing techniques have been adapted from [2].

For classification, we are using a Random Forest classifier that uses 42 distinct features of the symbols. Majority of these features are from [1] and most of these features consider the region of the bounding box in which the trace of a stroke appears. There are other features like number of traces in a symbol, that uses the fact that some symbols cannot be properly written without drawing specific number of strokes. For example, an equals symbol = cannot be written without two strokes.

Our two Segmenters assume that symbols are written from left to right, in time. This assumption keeps our segmentation process from becoming computationally expensive as navigating through every stroke in the file just once is left linear. We are not using a binary classifier to get merging decisions, but, we do have a binary function that tells whether two consecutive strokes can be merged or not based on the two features viz. Closest Two Points, and Center of Gravity or Centroid.

2 Preprocessing

As mentioned before, most of our preprocessing techniques come from the techniques explained in [2]. We start with removal of consecutive duplicate points. We linearly navigate through all the points in every stroke, and we discard the current point if the X and Y coordinates of that point is the same as that of the previous point. This is necessary because otherwise center of gravity of the strokes will be inaccurate as duplicate points get weighted greater. We then smooth the data points by best fitting them to a cubic spline function and then generating new equal-length sample points from the obtained function.

Last step we do normalization to remove variations that come from drawing the strokes in different regions of the fixed canvas. Our idea of normalization inflates or deflates every symbol such that it fits into a grid of size 2 by 2 in range [-1, 1] on both the axis, maintaining the

original aspect ratio. To maintain aspect ratio, we calculate maximum and minimum values along both the axis of the symbol and adjust the maximum and minimum values of the smaller dimension based on the larger dimension. We then use equations $x_{new} = 2\frac{x_{old}-min(x)}{max(x)-min(x)} - 1$ and $y_{new} = 2\frac{y_{old}-min(y)}{max(y)-min(y)} - 1$ for X and Y coordinates, respectively, to get 2D normalized output.

3 Symbol Classifier

We did not make any modifications to our set of features, or our classifier or its parameters. We are using a Random Forest classifier with no limit on the depth and 10 trees, and it is using 42 distinct features taken from [1]. Our list of features include number of traces in a symbol (1), mean of x coordinates (1), mean of y coordinates (1), covariance between x and y coordinates (1), aspect ratio (2), crossing features (10), and fuzzy histogram on points (25).

Crossing features divides the square symbol area horizontally and vertically, inserts 10 equidistant and parallel subcrossings or rays in each of these divisions, and then computes average number of intersections each of these rays make with the strokes.

Fuzzy histogram of points divides the square symbol area into a grid of size 4x4 and at each corner point (25) of these grids calculates the membership of the points of the strokes. Membership at each point is calculated using the formula $m_p = \frac{w-|x_p-x_c|}{w} \times \frac{h-|y_p-y_c|}{h}$ where w and h are the width and height of the grid cells, and x_p, y_p, x_c, and y_c are the x and y coordinates of the p stroke point and the c corner point.

4 Segmentation

We implemented our baseline segmenter based on our ideas in assignment-3 submission and then implemented a very similar sophisticated segmenter.

4.1 Features

We are using two features, one in each of our segmenters. We used closest two point distance from our idea in assignment-3 for baseline segmenter, and distance between averaged centers from [4] for our sophisticated segmenter. Upon research, we found that closest two point distance had already been used in [3].

4.1.1 Closest two point distance

K. Toyozumi calls it neighbor distance and they mathematically defined it as $min_{i,j}(e(p_i, p_j))$ in [3]. It is the minimum Euclidean distance e between any two points i and j of the two strokes, respectively. This is an excellent feature for those symbols that are made up of large strokes in relation with the entire expression. For example, some people write $\sqrt{root}$ symbol with two strokes, one for the check like symbol and the other for the top roof. In this case, feature like distance between averaged centers will be too big and those won't get qualified for merging. Other symbols that can benefit from closest two point distance are $\sum$, $\prod$, and extended fraction. The disadvantage of it is that it will want to merge almost every other symbols into one if they are too close in space.

4.1.2 Distance between averaged centers

This was one of the features used for segmentation in [4]. Defined as $e(\sum_{i=0}^{n} x_i, \sum_{i=0}^{n} y_i)$, it calculates the average of the points in the two strokes and uses the distance between the two centroids as feature. This feature doesn't work as well as closest two point distance when merging strokes for large symbols, but, does well for other regular sized symbols like alphabets and digits. We used this feature in our sophisticated segmenter since it works well for majority of the symbols.

4.2 Segmentation Algorithm

4.2.1 Baseline Segmenter

Apart from the fact that our baseline segmenter uses closest two point distance as the feature for deciding whether to merge two strokes or not, it also applies a maximum 3 strokes in a symbol constraint on itself while merging. This merge is transitive too. For example, stroke 2 is to be merged with stroke 3, and stroke 3 and stroke 4 can be merged as well, then all three strokes get merged together to form one symbol. So, our algorithm is pretty straight forward (see Algorithm 1). We linearly iterate through every trace drawn, assuming that symbols are drawn from left to right in space, and in time. we pass current stroke and the next stroke to our binary classifier, or in our case, a function, along with a maximum merge threshold value ($mthresh$). This function and the threshold value will explained in next few sub sections. This function returns $true$ or $false$ to indicate whether the strokes in picture should be merged or not. We keep on merging such consecutive strokes such that not more than 3 strokes get merged together. Our symbol classifier then iterates through *symbol* lists in *expression* list and predicts their classes based on the same 42 classification features explained earlier.

Algorithm 1 Baseline

```
1:  procedure PERFORMBASELINESEGMENTATION(traces)
2:      expression = []
3:      for <all traces> do
4:          symbol = []
5:          if min_{i,i+1}(eucdist(trace(i), trace(i+1))) < mthresh then
6:              if expression(lastindex).contains(trace(i)) then
7:                  symbol = expression(last index)
8:                  if len(symbol) <= 3 then
9:                      merge symbol = [trace(i), trace(i+1)]
10:         else symbol = [trace(i)]
11:         expression.append(symbol)
12:     return expression
```

4.2.2 Sophisticated Segmenter

Our sophisticated segmenter is not that sophisticated due to similarities with the baseline segmenter. The only two things that are different in this segmenter from the baseline segmenter is the lack of maximum 3 strokes per symbol constraint, the use of distance between averaged centers in place of closest two points as the feature for segmentation. We had initially put the maximum 3 strokes per symbol constraint thinking that there won't be any case in which a symbol will require more than 3 strokes. Then we stumbled upon an expression in the dataset that contained a sine function symbol written with 6 strokes, 1 for s, 3 for i, and 2 for n. We

found many such datafiles for a few other symbols as well (cos(), limit, summation, etc.) which made us realize that, perhaps, putting this contraint is not a good idea. Hence, we got rid of this constraint. Also, distance between averaged centers seemed like a better choice, since, it doesn't undersegment much by merging separate symbols whose parts of traces are very close in space.The flow of the algorithm is the same as that of baseline segmenter. See Algorithm 2.

Algorithm 2 Sophisticated

```
1: procedure PERFORMNONBASELINESEGMENTATION(traces)
2:     expression = []
3:     for <all traces> do
4:         symbol = []
5:         if eucdist(Σ_{i=0}^{n} x_i, Σ_{i=0}^{n} y_i) < mthresh then
6:             if expression(lastindex).contains(trace(i)) then
7:                 symbol = expression(last index)
8:                 merge symbol = [trace(i), trace(i + 1)]
9:         else symbol = [trace(i)]
10:        expression.append(symbol)
11:    return expression
```

4.2.3 Binary classifier

The purpose of the binary classifier is to make a decision whether the given strokes can be merged or not. In our program, instead of using a classifier model, we have used a simple function that doesn't need training. The function is simple and are included in the two algorithms shown above. If the distance from the feature is less than a pre-calculated merge threshold $mthresh$, it returns $true$ indicating we need to merge, otherwise $false$.

Maximum merge threshold value is calculated for every expression. That is, every expression will have a different $mthresh$ value based on the bounding box dimension of the expression (all strokes combined) and the number of input primitives i.e. the traces or strokes. we find height and width of the bounding box of the whole expression, and divide the greater of the two dimensions by the total number strokes in the expression. Mathematically, formula for mthresh can put be put as below.

$$mthresh_{expr} = \frac{max(height_{expr}, width_{expr})}{n+1} \tag{1}$$

The idea behind the use and formulation of this threshold was that it would give us a decent estimate of how far apart the centers of the symbols might be in the expression. Inferring that strokes whose distance between the centers were less than this threshold could be considered too close to regard them as separate symbols. In our implementation we have an additional constant value α (common across all expression) with which we multiply the $mthresh$ value to scale it or fine tune the calculation of it to give us good results overall. Keeping α too high would result in under-segmentation and keeping it too low over-segmentation. We kept $\alpha = 1.0$ as default to have less bias towards any of these two segmentation errors. We found that segmentation with $\alpha = 0.8$ improved our baseline segmenter's performance that used closest two point distance as the feature. For our sophisticated segmenter $\alpha = 1.0$ did well. $\alpha = 1.1$ did better segmentation in lengthier or bigger expressions. Our numbers in the results section are all on default $\alpha = 1.0$.

5 Results and Discussion

We are splitting the dataset containing 8834 working INKML files into 2/3 and 1/3 using a naive approach. Our programs reads the names of the data files linearly, as it appears on the disk of the machine, and then shuffles this list of filenames randomly. The ground truth and the trace data are read altogether from these shuffled files and are saved in lists. We create a hashmap containing symbol labels as keys and their counts as their corresponding values from the ground truth. We then iterate through every file again and see if adding this file to the training set would increase the count of any of the symbol label it contains by roughly 0.67 times the total appearance of that label in the whole dataset. If it does, we add this file to our test set instead of train set.

Figure 1 summarizes results observed for recall, precision and f-measure, we obtained from the supplied evaluation tool. We can see that differences in F-measures from our baseline and non-baseline segmenters, both, are substantial. We did expect our results of baseline to be poor as compared to non-baseline because of the choice of feature used in it. The closest two point distance criteria of the baseline segmenter is quite likely to confuse the binary function into merging strokes from two different symbols that are close to each other into one legible symbol. For example, we found a case $\sqrt{x}$ where in the roof of the root symbol and the top of the x were so close that they were merged to give us a false segmentation and classification of π. Similar thing happened in another case when the a and the c of $4ac$ got merged into one symbol to give an output of w.

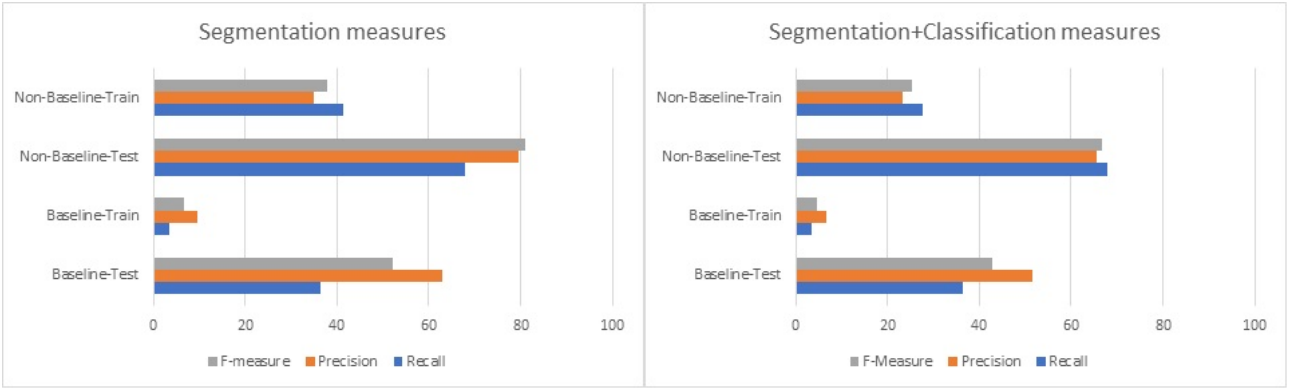

Figure 1: Precision, Recall and F-Measure for Segmentation and Segmentation + Classification Results.

The First bar graph shows results of object segmentation of both of our segmenters when tested with test and train split. Our non-baseline segmenter gave us an F-measure accuracy of 80.97% for segmentation when tested with test split and 37.90% with train split. This difference is huge, and we think that its because of the difference in types of expressions that randomly got put in train and test split. Our merge threshold is working well on the test split of the data where number of lengthy and hefty expressions are less. A similar pattern is seen in baseline results and in segmentation + classification bar graph as well for the same reason.

The second bar graph summarizes the results observed when the segmentation is combined with classification. Here in addition to observing segmentation errors we observe classification errors too. Hence, the measures are bound to take a dip as compared to just segmentation results because of the added classification errors. There were cases when the obtained segmen-

tation from the segmenter is correct, but the classifier fails to classify the symbol correctly. There were multiple cases of ',' commas getting classified as 'j' or '1'.

On making use of confHist tool to detect node level errors, the common pattern we observed particularly for segmentation was of when a single symbol is composed of more than one stroke, and they are distant is space from each other. For example, if you take a symbol square root and it has two strokes for it, one an inclined stroke and other horizontal stroke the centroids will be quite far, and our threshold wouldn't cover such cases to merge. Similar such cases observed were for sin, cos, tan, summation symbols. For such cases the proposed system shows over-segmentation which can possibly be fixed by the use of closest two point distance feature or using some sort of contextual feature. There are multiple places where our system can be improved. We introduced an additional constant parameter 'α' that could work along with our *mthresh* value to scale and adjust it to bring as many expressions as possible to better segmentation. Had we used an actual binary classifier and trained with some distance and contextual features, we would have obtained an optimal mthresh value for every situation to give us best results overall. Also, considering re-segmenting if the probability obtained from the symbol recognizer is below a threshold might fix a few incorrect segmentations.

Table 1: Training and Running times For Segmenter and Classifier(including features)

Segmenter		Classifier	
Training (seconds)	Execution (seconds)	Training (seconds)	Execution (seconds)
na	<1 sec	2,143	847

6 References

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

[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] K. Toyozumi, N. Yamada, T. Kitasaka, K. Mori, Y. Suenaga, K. Mase, and T. Takahashi, "A study of symbol segmentation method for hand-written mathematical formula recognition using mathematical structureinformation," inProc. Int'l Conf. on Pattern Recognition, Aug. 2004, pp. 630–633.

[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.