CSCI 737 — Pattern Recognition, Spring, 2016
Project #3

Michael Potter, Deepali Kamat

May. 21, 2016

1 Design

This project document describes the pre-processing, feature extraction and parsing techniques used to derive symbol relationships from stroke data. The hand written recognition task is taken from the Competition on Recognition of On-line Handwritten Mathematical Expressions (CROHME)[1]. Before moving into parsing, improvements were made to the segmentation algorithm presented in Project 2. Whereas previously no assumption was made regarding the temporal order in which strokes were recorded, instead favoring a 2D clustering based approach, our new segmentation system assumes temporal ordering. While this assumption sacrifices some of the theoretical expressiveness of the system, it also allows for an easily implemented 1D dynamic programming approach to segmentation. The new system runs the classifier on each stroke together with its up-to-3 most recent strokes. Combinations are rejected if the strokes could be separated by a vertical line, unless the confidence level for one of the 3-character symbols is over 70%. The top scoring option is then recorded in a 1D array, along with a pointer back to the previous segmentation break. Since multiplying two probability scores of high confidence is often less favorable than taking a single probability at a slightly diminished confidence, this system helps to solve the problem endemic to our part two system where symbols were often left separated into component pieces rather (ex. '-' and '1') than being combined together (ex. '+'). The vertical line pruning is introduced to counteract the heavy preference of the dynamic approach for longer symbols ($0.8^4 = 0.4096$, so if the system could be even 41% confident in a symbol such as sin or log, it would choose this symbol over more probable candidates).

After segmentation is performed, a greedy recursive parser is used to determine symbol relationships. The parser operates by baseline identification, starting with the left-most symbol and attempting to find the next symbol with a 'Right' relationship to it. All symbols between the first baseline symbol and this next symbol are then split into groups based on their relationship to the baseline ('Above', 'Below', 'Sup' (Superscript), 'Sub' (Subscript), and 'Inside' (within a square root)) and recursively parsed in the same way. This design was chosen for its ease of implementation as well as its grammar independence while maintaining theoretic expressiveness.

2 Preprocessing and Features

The features used for parsing are taken form the works of Simistira et al.[2]. Our system compares two symbols by first generating tight bounding boxes around each individually. A larger bounding box is formed around their conjoined mass for normalization. The distances between various portions of the bounding boxes are then measured to generate features. In particular, the distances between the left, right, top, and bottom edges of the bounding boxes are measured and normalized by the width and height of the larger bounding box. The differences between the x and y coordinates of the centroids (naive, not center of mass) of the two bounding boxes are also recorded, again normalized by the larger box width and height respectively. Finally, the angle between the centroids of the bounding boxes are computed using standard trigonometry. The angle between centroids is broken into two separate features: sin(theta) and cos(theta). These are used to provide a continuous representation of the space (otherwise the difference between 0 degrees and 364 degrees would appear very large).

3 Parsers

As prescribed by the project, two parsing applications have been created, though both complete the actual task of parsing relationships in an identical way. The simpler system uses ground truth .lg files to extract the true symbol segmentation and classification decision before engaging in parsing. The full system segments and classifies symbols itself before handing over the results to the parsing algorithm for subsequent analysis. Neither system uses parsing to revise the earlier segmentation or classification decisions, which expedites this modular design. A more detailed explanation of the classification and segmentation tasks may be found in the Project 2 report.

The parser is designed to take expression symbols as input and then recursively identify baselines over which relationships are defined. Expressions are traversed from left to right. During traversal, the current baseline symbol is compared with subsequent symbols until three candidate symbols with a ‘Right’ relationship are found. The search is terminated early if a large horizontal gap exists between the first candidate encountered and the subsequent candidates in order to reduce the search space. If three valid candidates are encountered (potentially implying the existence of a fraction or sum), the candidates are compared to each-other. If any candidate has an ‘Above’ relation with one candidate and a ‘Below’ relation with the other, then it is in between the two and should be selected as the next baseline character. If three candidates are not found (due to the horizontal gap rule), then the candidate with the highest probability relationship is taken to be the next baseline. The symbols that fall between the current baseline character and the newly identified one are then split into groups based on their relationship with the current baseline symbol. These could therefore be either ‘Above’, ‘Below’, ‘Sup’, ‘Sub’, or ’Inside’. If any of the intermediate symbols have the ‘Right’ relationship (for example the limit of a sum, where the sum was chosen as the next baseline character but the limit was written slightly to the left of the sum), they are moved in the traversal list to follow the preferred baseline. These subgroups of intermediary symbols are then recursively analyzed in the same way. The system then moves to the newly identified baseline character and repeats the process with the remaining symbols. Pseudocode for this process is given below:

```
Parse()
Input :strokedata

medianwidth = median(strokedata)

#Sort the symbols based on their stroke coordinates from left to right based on x value:
traversalorder = []
for each j,stroke in strokedata:
        if stroke[1] in strokegroups[]:
            minX = min( x in stroke)
            traversalorder[j][1] = minX

ParseRecursive()
Input: traversalorder, strokedata
relationship = []
candidate = []
baseline = strokedata[0]
#Search for up to 3 candidates for the next baseline symbol
for val in traversalorder:
    relation = classifier.compare(baseline,strokedata[val]
    if ‘relation’ = ‘Right’:
        if already have at least 1 candidate:
            ignore candidate if x distance to existing candidates > median_width/8
        add val to candidate list
#check for fractions if left with 3 symbols
if 3 candidates:
    for every permutation (a,b,c):
        if classifier.compare (a,b) == ’Above’:
            if classifier.compare(a,c) == ’Below’:
                newbaseline = a
else
    newbaseline = candidate with max classifier score

#evaluation of symbols between baseline symbols
for symbol in traversalorder between baseline and newbaseline:
    #compare elements to next element and check for relationship class
    relation = classifier.compare(baseline,symbol)
    if relation == ‘Above’:
```

```
                above.append(symbol)
        elif relation == ‘Below:
                below.append(symbol)
        elif relation == ‘Sub’:
                sub.append(symbol)
        elif relation == ‘Sup’:
                sup.append(symbol)
        elif relation == ‘Inside’:
                inside.append(symbol)
    elif relation == ‘Right’:
        right.append(symbol)

for set in [above, below, sub, sup, inside, right]:
    relationships = ParseRecursive(set)

return relationships
```

The run-time complexity of this algorithm is $O(n^2)$. The initialization can be completed in $n * log(n)$ time for the sort. The recursive calls each require at most n steps to locate the 3 candidates for the next baseline, with each test requiring a (albeit large) constant time call to the random forest. The second loop is also worst-case $O(n)$ but is in series with the first. The recursive depth is at most n since each call removes at least one symbol from the search space. This gives the final complexity as $O(n^2)$.

This parsing system is able; in theory, to recover arbitrarily complex math expressions, but does have several limitations on the possible expression set. First, the expression must be comprehensible in a left-to-right order (not a problem for most math, but would limit its utility in parsing Chinese poetry). The parser also assumes that only 6 possible symbol relationships exist, so any expressions containing alternative relationships (such as a pre-subscript) will not be interpreted properly. There are also implementation-based limitations of this strategy which impede its expressiveness. The most obvious such limitation is that if the left-most character of an expression is not part of the primary baseline of an expression, then the entire parsing thereafter may be thrown off (though in many cases the substructure could still be correct). This could be fixed in future implementations by adding a dummy start node at some negative x coordinate and launching the recursion from there. Another limitation of the system is that, when evaluating vertical expressions such as fractions, the left-most element of the numerator and denominator will be the ones related to the baseline element. This is not always appropriate, for example if someone were to write $\frac{\sqrt[x]{e}}{n}$ with the x slightly to the left, it would become the new baseline of the numerator rather than the radical.

4 Results and Discussion

As described in Section 1, a significant overhaul of the segmentation algorithm was performed between this project and the last. The performance of this new system on both training and testing data is summarized in Figure 1.

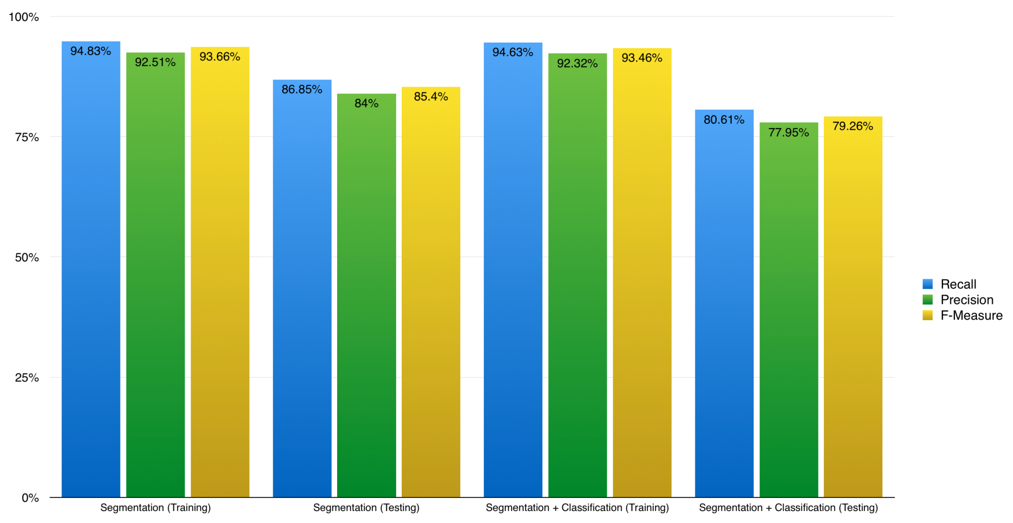

Figure 1: Symbol Segmentation + Classification Performance

For comparison, the previous system achieved an F-Measure of 75.9% on the combined task of segmenting and classifying symbols, as compared to the 79.3% achieved with the new system. Performance improvements are gained in every category (training, testing, segmenting, segmenting + classifying), with the strongest grains coming from improved precision. While recall scores increased from the previous system by around 2%, precision jumped by 4-5%. This is understandable since the new system tends to heavily favor combining strokes (multiplying fewer confidence values together tends to result in higher scores), whereas the old system required a contrived heuristic in order to privilege more complex stroke combinations. In fact, the propensity of the dynamic programming system to combine strokes into large symbols was so strong that the system actually performed worse than the greedy heuristic algorithm before the horizontal spacing constraint described in Section 1 was introduced.

The simpler of the two systems developed for part 3 of this ongoing project, however, was implemented without need for a segmentation algorithm. Basing the segmentation and classification decisions off of ground truth values, the parsing algorithm was able to achieve consistent and fairly high relationship detection rates, as summarized in Figure 2.

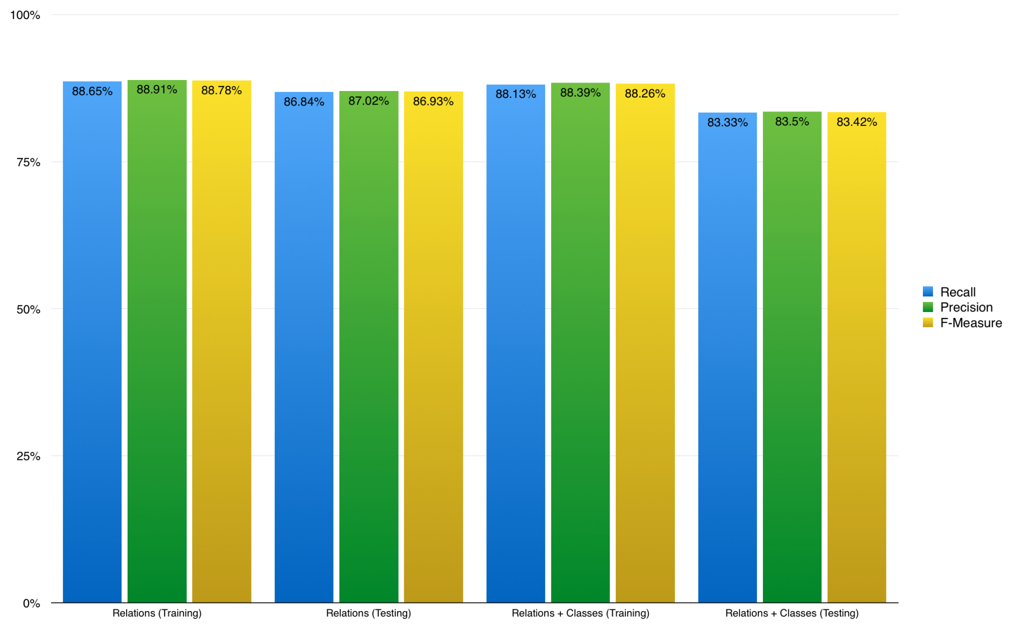

Figure 2: Relationship Parsing Given Perfect Segmentation

As Figure 2 shows, recall and precision rates for the parser are virtually identical, with a slight edge towards precision. This

is in stark contrast to the segmenter from above. The difference in performance between training and testing data is also much less pronounced for parsing than it is for segmentation. While the segmenter's F-measure lost over 14 percent off of its performance score going from training to testing data lost less than 5 percent. Although the smaller performance differential between training and testing may indicate that the parsing system does a good job in not over-fitting the data, it is also disappointing that higher scores were not achieved given that there are only 6 different relationship classes compared to the over 100 character classes which must be handled by segmentation.
As expected, when the system must segment and classify symbols before parsing relationships, the performance of the relationship parsing drops. This is shown in Figure 3.

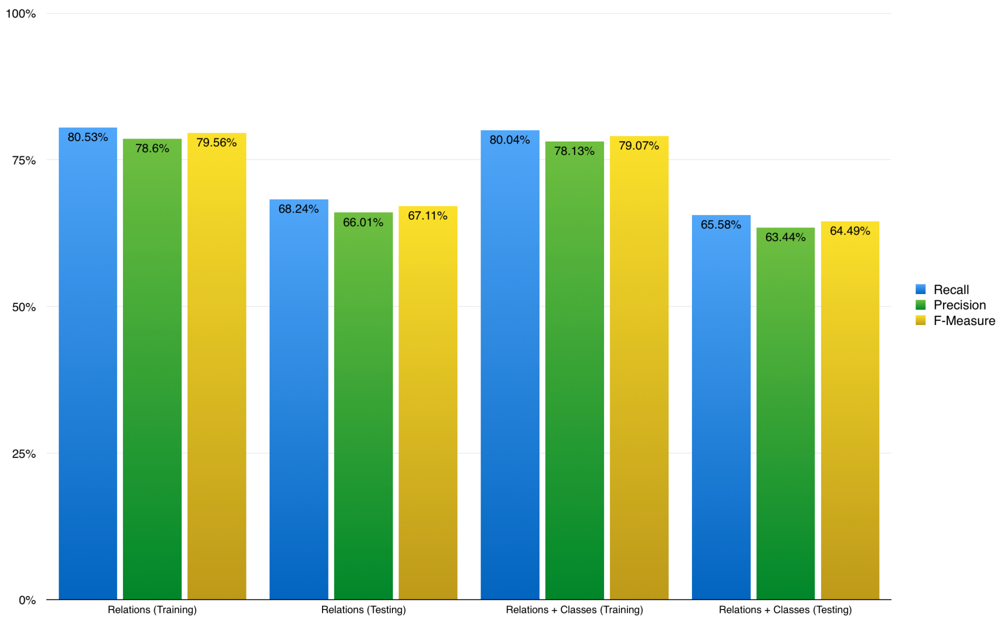

Figure 3: Relationship Parsing After Imperfect Segmentation

In contrast to the performance of the pure parser, the full system has lower precision than recall when identifying relationships. This is likely a bi-product of the segmenter having lower precision than recall (roughly 2% lower, which happens to also be the magnitude of the performance difference appearing in the parser following up on that system). The testing performance of the full system when parsing is fairly low (at 64.5%). This is understandable, however, by considering the compound effect of the segmentation and pure parsing errors. In fact, if modeled as independent error sources, the predicted F-measure when combining the segmentation and parsing systems on the test set would be 0.79*0.83=0.66, which is quite close to the observed value. This relationship holds for the other data combinations as well (testing/training and relations/classes), with the true value being slightly worse than what independent error estimates would predict in the various cases. It therefore seems likely that further improvements to the segmenter would proportionally improve the results of the final parsing system.

The confhist utility was used to investigate specific parsing errors in more detail. The five most common error cases for parsing given perfect segmentation are given in Table 1.

Table 1: Common Parse Errors (Perfect Segmentation)

Target Relation	**Total Errors**	**Most Frequent Erroneous Guess (Occurrence Count)**
+ *right* 1	58	no relation (57)
- *above* 1	61	no relation (44)
= *right* 1	56	no relation (50)
- *right* =	54	no relation (51)
x *right* ,	48	x *Sub* , (39)

It appears that in the majority of cases, parsing errors are due to a failure to identify the relationships between two objects. The most likely cause of these errors is that the system is picking the incorrect baseline hypothesis when finding the next symbol to come to the right of a current baseline node. A notable exception to the more common error pattern is in the case of commas, which are sometimes interpreted as subscripts rather than as right relations. This error is understandable since commas are often small, immediately adjacent to the character they are paired with, and in the location where one might expect to find a subscript character. This could potentially be fixed using a post-processing rule if the maximum possible accuracy is desired.

To see if the most common parsing error characteristics change as a result of segmentation failures, a table of common relational errors was generated for the full system. This data is given in Table 2.

Table 2: Common Parse Errors (Imperfect Segmentation)

Target Relation	Total Errors	Most Frequent Erroneous Guess (Occurrence Count)
- *above* 1	126	no relation (34)
+ *right* 1	117	no relation (51)
= *right* 1	97	no relation (31)
x *right* +	96	no relation (31)
= *right* 1	87	no relation (26)

In keeping with the earlier hypothesis that the errors of the two system components are largely independent, the top three parsing error targets are conserved between the systems regardless of segmentation performance. The key difference contributing to the roughly doubling of the error rates in each category moving from perfect segmentation to imperfect segmentation is that in the latter case there are a large set of errors due to incorrect interpretation of symbols rather than incorrect relationships. Looking at the '- *above* 1' relation, for example, the 2 next most common errors after 'no relation' both involve the 'above' relationship, but between improperly segmented/classified symbols.

If this system were to be improved in the future, transitioning the parser to a dynamic programming algorithm or spanning tree rather than the current greedy implementation would probably yield superior results, as was found with the segmentation system. There are, however, problems with the dataset which need to be addressed in order to achieve optimal performance. When investigating error cases, it was discovered that at least one ground truth file (101_Nina) improperly labeled the above and below relationships as superscript and subscript. It is unclear how many other data instances are compromised, but if the problem is widespread this inconsistent labeling could be preventing the relationship classifier from developing good rules for differentiating between these cases. This in turn could be responsible for some of the structure-level parsing failures observed, since our algorithm runs recursively on sections of data defined by their relationship to a baseline character. If these characters become artificially divided into two sets split between - for example - below and subscript, then many structural errors will inevitably follow.

References

[1] Harold Mouchère et al. "Advancing the state of the art for handwritten math recognition: the CROHME competitions, 2011–2014". In: *International Journal on Document Analysis and Recognition (IJDAR)* (2016), pp. 1–17. ISSN: 1433-2825. DOI: 10.1007/s10032-016-0263-5. URL: http://dx.doi.org/10.1007/s10032-016-0263-5.

[2] Fotini Simistira, Vassilis Katsouros, and George Carayannis. "Recognition of online handwritten mathematical formulas using probabilistic SVMs and stochastic context free grammars". In: *Pattern Recognition Letters* 53 (2015), pp. 85–92.