**Handwritten math recognition:**

OCR engines: (optical character recognition)

1. Google OCR: works OK for neat and tidy handwritten numbers, simple operators/symbols
2. Amazon textract: doesn’t work
3. Tesseract (open source):
   1. works OK for GOOD handwritten numbers, simple operators/symbols
   2. can be trained for individual style/font to work VERY WELL with GOOD handwritten numbers, simple operators/symbols, but the documentation on training is very hard to understand
   3. The newest version uses LSTM, which implies that it’ll not work fine with training general handwritten math. Previous versions
4. Other less well-known services: Mathpix (doesn’t export bounding box information), InftyReader, etc. None works for our needs.

Conclusion: These OCR engines, no matter what algorithms and ML techniques they use, all suffer from the fact that they are not designed to work with general HANDWRITTEN MATH.

ICR engine: (intelligent character recognition)

The state of art techniques in general handwritten math recognition have to use STROKES data. In OCR, only PIXELS data are provided. The two best ICR engines from an international competition are:

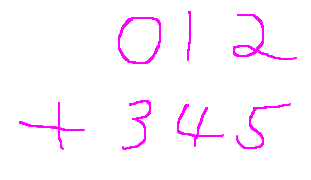
1. MyScript:
   1. For our needs, which are currently mufti-column arithmetic, long division, and fraction arithmetic, the accuracy is in high 90s for both. Its full potential has been explored. The 3 remaining problems that only changing the backend code will solve are:
2. doesn’t recognize long division symbol
3. export format may affect bbox parsing accuracy for different strokes sequences
4. not free, actually expensive
5. seshat(open source):

Seshat was designed to recognize handwritten math expression by using strokes data.

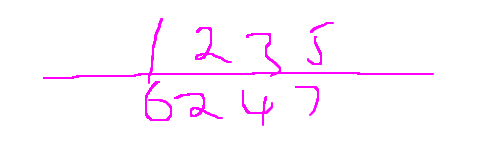
The 3 tasks it accomplishes are symbol segmentation, symbol recognition, and structural relationship recognition. Its general idea is to generate a set of possible parse trees, and get the most likely one based on a probability measure. Instead of accomplishing the 3 tasks step by step, which is what some other systems do, seshat analyzes segmentation and recognition together. The reason is that the hypotheses for different segmentations can be affected by structural relationships, and vice versa. Refer to the paper for details on the parse trees, probability equations, and ML methods.

In order to use seshat for our needs, 3 major challenges exist:

1. The system can only recognize 1 line of math expression(s) at a time, mainly because the model for the parse tree does not have heuristics about multi-line structures. We can partially fix this problem for multi-column arithmetic inputs. Similar to text lines, each line in such an input is a well-formed horizontal/slanted line. Some existing ideas and algorithms in the text lines segmentation research field are potentially useful. My 2 algorithms are as follows:
2. Cluster all of the (x,y) points of the strokes data based on their y coordinates. Each cluster indicates a horizontal region where a line should reside in. Multiple clustering methods have been tested, including K-Means, affinity propagation, and the kernel density method. They work pretty well with good inputs such as in the following figure. To be more precise, a good input is one in which the bounding box for each line is a horizontal rectangular box and all the bounding boxes do not intersect. However, such good inputs are too restrictive for the users to have a good experience. Bad inputs are fed into the methods, and only K-Means works fine, but it



requires knowing K, the number of clusters, before running the algorithm. All major methods for computing the optimal K have been tested too, including the elbow method, silhouette method, and gap-statistic method. Only the silhouette method shows OK results, but still far from satisfying for bad inputs, i.e. multi-column inputs in which numbers in different lines overlap to some degree. Another serious problem with clustering the strokes data based on Y coordinates is that the clusters may not reflect a ground-truth division of the actual lines. In the following example, there are 2 clusters, but the lower one contains some noise data points from 1 and 3 in the upper cluster, and this may affect the recognized results by seshat.



Conclusion: Perhaps the mainstream clustering methods, although sound fancy, do not fit the nature of our unsupervised clustering task on the strokes data.

1. Instead of clustering the (x,y) points, can we cluster the strokes instead? Each stroke is a set of (x,y) points. By looking at a level up, we may achieve more accurate results. The general idea is to first get the bounding boxes of the strokes, and divide these boxes into groups. Each group is a set of strokes residing in the same line.

Initialization:

Get the list of bounding boxes of the strokes.

Create an undirected graph, each node representing one box.

Algorithm:

getLines(List, parameter):

for box i in List:  
 for box j in List:

Li = height of box i

Lj = height of box j

Lo = overlapping height

if (max (Lo / Li, Lo / Lj) > parameter):

Connect i and j in the graph

box k = minimal enclosing box of box i and j

if (box k doesn’t “grow up” vertically much from box i):

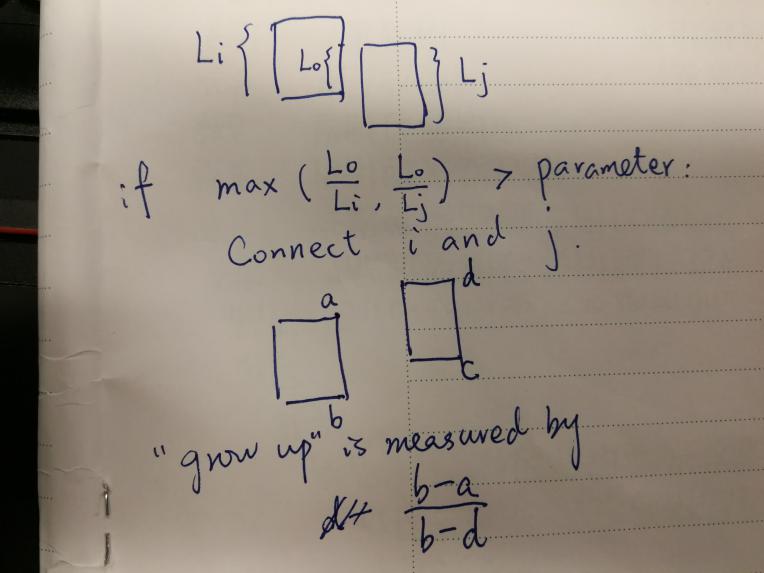
//indicate j is close to i vertically

Connect i and j in the graph

Run DFS(depth first search) to get the connected components in the graph.

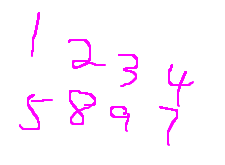
//Each connected component is a line.

Illustration:



Some explanations:

1. The measure of overlapping length is max (Lo / Li, Lo / Lj). It allows worse inputs than taking 2\*Lo / (Li + Lj), or taking the vertical distance between the centers.
2. We use the “grow up” measure because in some situations such as the number 5, the above stroke typically doesn’t overlap with the lower stroke, or because, in general, strokes don’t align as well as terminal symbols.
3. We use a graph and get the connected components. This way allows more kinds of inputs too, such as slanted but still acceptable lines.



Conclusion: The accuracy is above 85% on my dataset. If the input doens’t have too many lines overlapping with each other, the algorithm should work satisfactorily in production uses.

1. The recognition accuray needs improvement.

Symbol recognition and especially structural relationship recognition are not satisfying when it comes to bad handwriting. Limiting the list of symbols and structural relationships improves the accuracy a little, but the core algorithm of seshat needs change if we aim higher. The core algorithm is a combination of probability models and machine learning.

1. The core algorithm is not meant to recognize symbols such as the long line in multiple-column arithmetic problems and the long division symbol. We can probably solve this problem by training the RNN to recognize them.

**Run seshat:**

Get a feeling about seshat:

1. git clone <https://github.com/yirentang/seshat>
2. Go to the seshat folder: cd seshat
3. Seshat uses the C++ boost library. Typically it is included with Ubuntu system, and installed in the /usr/include/boost directory. Otherwise, change the path to boost in a -I flag in Makefile.
4. Compile seshat: make
5. Refer to README.md in “seshat” on the structure of the command.

Run a user interface:

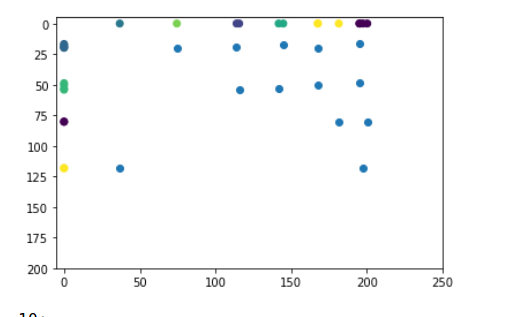
Refer to README in “seshat/server”

Collect strokes data from users and run algorithms on the data:

Refer to README in “seshat/server/testing”

**Bounding Box Algorithm:**

After seshat recognizes the mathematical expression, we want to get the relative positions among the terminal symbols in the expression. Getting relative positions means getting the above, below, left, and right boxes of all boxes. Of course, the relative positions are subjective to a large extent for very cursive or ill-aligned mathematical expressions, and so the goal of designing such an algorithm is to allow expressions as worse as possible while also getting the correct relative positions. Two kinds of algorithms are proposed. The first one is clustering algorithm(s). The idea is to cluster the centers of the bounding boxes of the terminal symbols by their x coordinates or y coordinates. Clustering by x coordinates will get the columns and clustering by y coordinates will get the rows, as illustrated.



What has been said about clustering algorithm(s) in the context of dividing strokes into rows is still true in this context of dividing terminal symbols into rows and columns. Put it in one sentence, clustering algorithm(s) only work OK on good inputs.

The second one is a simple rule-based algorithm that a CS undergraduate can easily understand, but works well enough, at least to my expectations. This algorithm is 100% original.

The idea is to first get the rows by evaluating the vertical overlap between pairs of bounding boxes, and then use [2-candidate matching] and [network propagation] to get the columns from the rows.

Get rows:

Initialize: Create an undirected graph, each node representing one bbox.

for each pair of bboxes i and j:

Li = height of bbox i

Lj = height of bbox j

Lo = overlapping height

if (max (Lo / Li, Lo / Lj) > parameter):

Connect i and j in the graph

Run DFS to get connected components

// each connected componenet is a row

Get columns:

Initialize: Create an undirected graph, each node representing one bbox.

for each pair of rows i and j: (i is above j)

for each pair of bboxes m and n: (m in i, n in j)

Cm = center of m

Cn = center of n

S = abs(slope of the line connecting Cm, Cn)

for each bbox m in i and for each bbox n in j:

Maintain a preference list =

[candidate1, candidate2]

Candidate1 and candidate2 are the 2 bboxes whose S values are the highest 2.

for each bbox m in i:

if the top candidate of the top candidate of m is also m, then connect m and the top candidate

Disable the candidates that are already matched. Promote the 2nd candidate to the 1st candidate if the 1st is disabled.

In the graph, compute the change in the row number for each edge of each node.

For each node, only keep the edges with the least change in row number. Delete other edges.

Run DFS to get connected components.

//Each connected component is a column.

Things to add:

1. Better pseudo code
2. Add the subtleties behind the algorithms. For example, why use [2-candidate matching] and [network propagation]
3. Related papers