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:

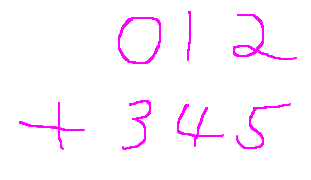
1. MyScript:
   1. symbol recognition: high 90s, structural relationship recognition: high 60s.
   2. For our needs, which are currently mufti-column arithmetic, long division, and fraction arithmetic, the accuracy is in high 90s for both. Its full potential has been explored. The 3 remaining problems 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.

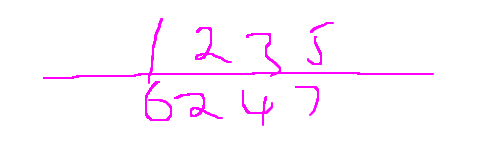
The 3 tasks it accomplishes are symbol segmentation, symbol recognition, and structural relationship recognition. Its general idea is to generate a set of possible parse trees, and get the most likely one based on a probability measure. Instead of accomplishing the 3 tasks step by step, which is what 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 arithmetics 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, silouhtte method, and gap-statistic method. Only the silouhette 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):

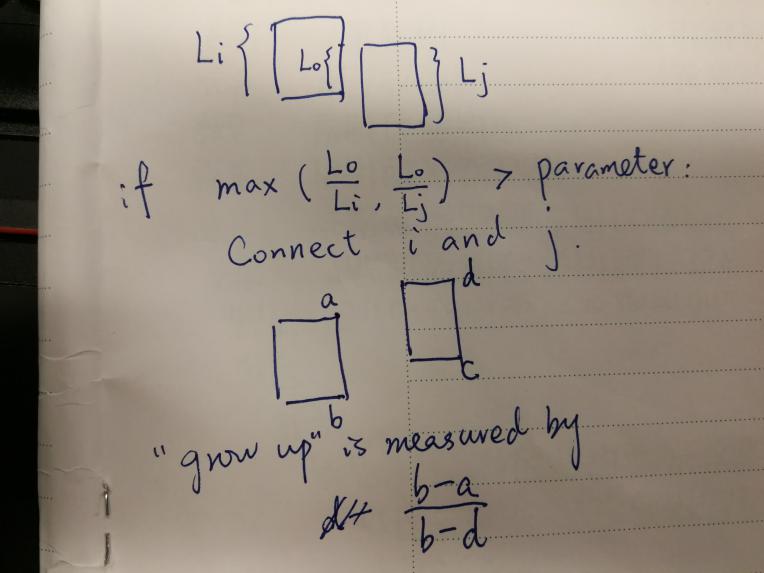
//indicate j is close to i vertically

Connect i and j in the graph

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

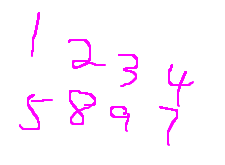
//Each connected component is a line.

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.

**Seshat usage:**

1. git clone <https://github.com/yirentang/seshat>
2. Go to the seshat folder: cd seshat
3. Download the “nlohmann” folder from <https://github.com/nlohmann/json> and add it to the “seshat” folder. This adds support for JSON in C++.
4. 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.
5. Compile seshat: make
6. Run the executable:

$ Usage: ./seshat -c config -i input [-o output] [-r render.pgm]

-c config: set the configuration file

-i input: set the input math expression file

-o output: save recognized expression to 'output' file (InkML format)

-r render: save in 'render' the image representing the input expression (PGM format)

-d graph: save in 'graph' the description of the recognized tree (DOT format)

Example:

$ ./seshat -c Config/CONFIG -i SampleMathExps/input0.scgink -o out.inkml -r render.pgm -d out.dot

You will then see the output of the program. The most important part of the output is the JSON in the end. I added some code in the vanilla Seshat to output this JSON, which will be used in the communication between the client and server. The JSON output is the essentially the parse tree of the recognized mathematical expression.

To run Seshat in the backend and run a user interface in the frontend, navigate to the server folder and run

$ python3 runserver.py

Then open the localhost [http://127.0.0.1:8000](http://127.0.0.1:8000/)

in a browser.

**Bounding Box Algorithm:**

After Seshat or some other engines 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. This is most applicable to multi-column arithmetic problems, because

Bounding box algorithm (transforming the export of MyScript into a form that AL can accept):

Core problem:

Given the positions and sizes of a set of bounding boxes, how do we get the relative positions (above, below, left, right) among these boxes as accurately as possible?

2 type of solutions: targeted for mufti-column arithmetic problems, general for any set of bboxes

Targeted: easy, solved, using horizontal range / K-means

General: hard, current promising solutions:

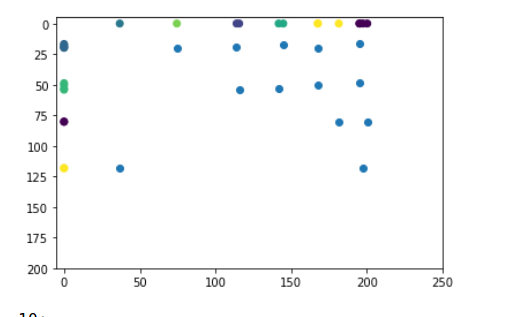
General ideas:

1. Simple rule-based algorithms:
2. a set of boxes are in the same horizontal line <=> the horizontal distances among their centers are within 0.5 times the average height of all bboxes result: good
3. From above step we get a list of horizontal lines. For center c1 in line 1, traverse all centers c2(s) in line 2 and choose the one s.t. the slope of the line c1c2 is maximized.

Danny suggested using lines next to each other and down the vertical position information to all lines. It improves accuracy a lot, and I think is the right direction to go with.

1. Clustering algorithms:

The centers of the bboxes are separated into different clusters according to their x-axes values or y-axes values. The K-Means algorithm works very well. It only remains to find the best algorithm for determining the number of clusters.



1. Combinations of the above ones