Image to CSV

Andrey Kurenkov, Pavel Komarov, Ricky Liou, Hank Mccord, Ramon Sua

1 Problem Statement

Our goal was to write an application that can produce a digital spreadsheet file from a photo of a table on physical paper. More specifically, the input to our application is a smartphone-quality photo of a handwritten or printed table filled with handwritten or printed text or numbers. The output of our application is a CSV file containing the extracted content of the table.

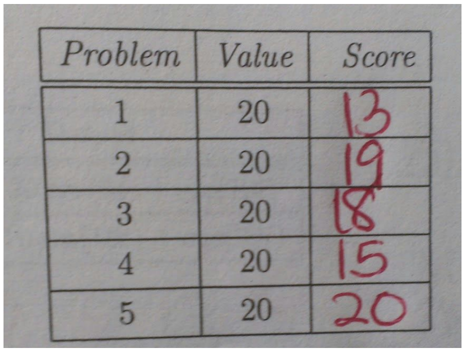
Input Image	Output File Content
 Problem Value Score 1 20 13 2 20 19 3 20 18 4 20 15 5 20 20	Problem, Value, Score 1, 20, 13 2, 20, 19 3, 20, 18 4, 20, 15 5, 20, 20

Figure 1. An example input and output.

An assumption we made concerning the input image is that it will be cropped to make the table to be extracted the only content in the image, as in Figure 1. Figure 2 is a diagram of the solution pipeline, which is quite complex; the solution requires a full OCR (Optical Character Recognition) solution implemented from scratch. The initial step for processing an input image is to simply threshold it and smooth it slightly to get a black and white image with reduced noise. The next step is to extract the table structure and find each individual cell within the table so they can be processed separately. After the image is split into the subimages of table cells, each subimage is further segmented to extract images of characters. Each of these is passed through a HOG feature extractor, the results of which are passed to a classifier that returns the latin letter or arabic numeral pictured. After all characters in a table-cell subimage are classified, they are combined into a single string using the character image locations and classifications. Character synthesis uses character locations to insert spaces. Lastly, the strings from each cell are combined based upon known table structure, and the final result is output to a file.

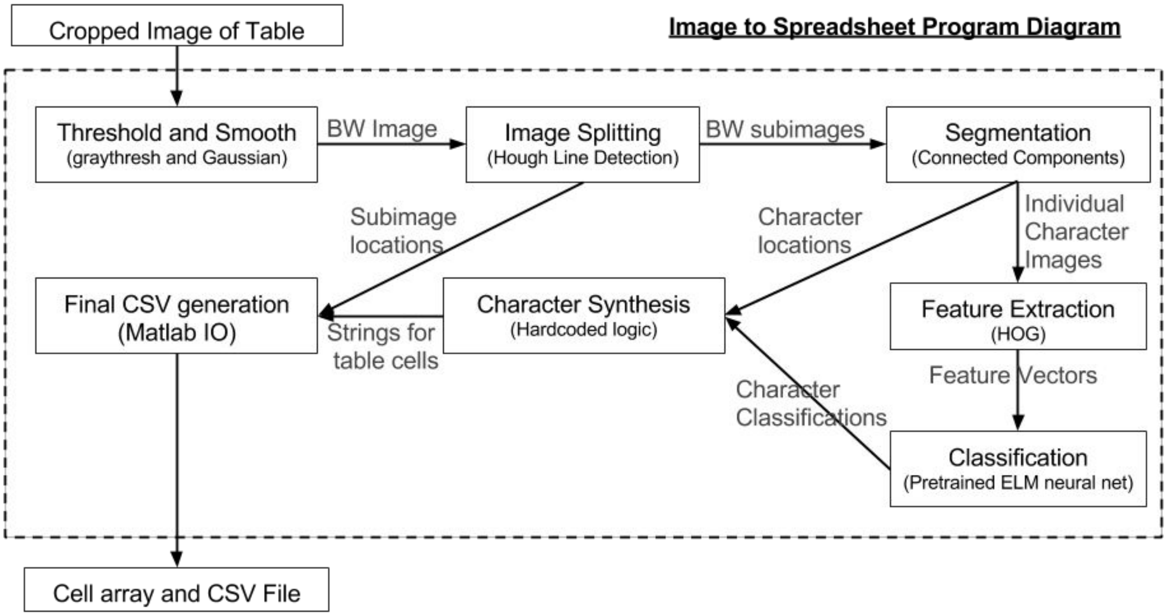


Figure 2. Diagram of the solution pipeline. Corresponding code in src/ImageToCSV.m

2 Thresholding and Smoothing

The first step in the pipeline is to create a black and white version of the image. We used the default graythresh method found in Matlab, as it intelligently picks a threshold to minimize variance between black and white pixels--a reasonable approach for handwritten text. As shown in Figure 3, the results for our first testing image were sufficiently good with graythresh. Any noise was small enough to easily filter, through some letters were very pixelated in appearance. To manage this, the image was smoothed, as we expected less-noisy images would be easier to classify.

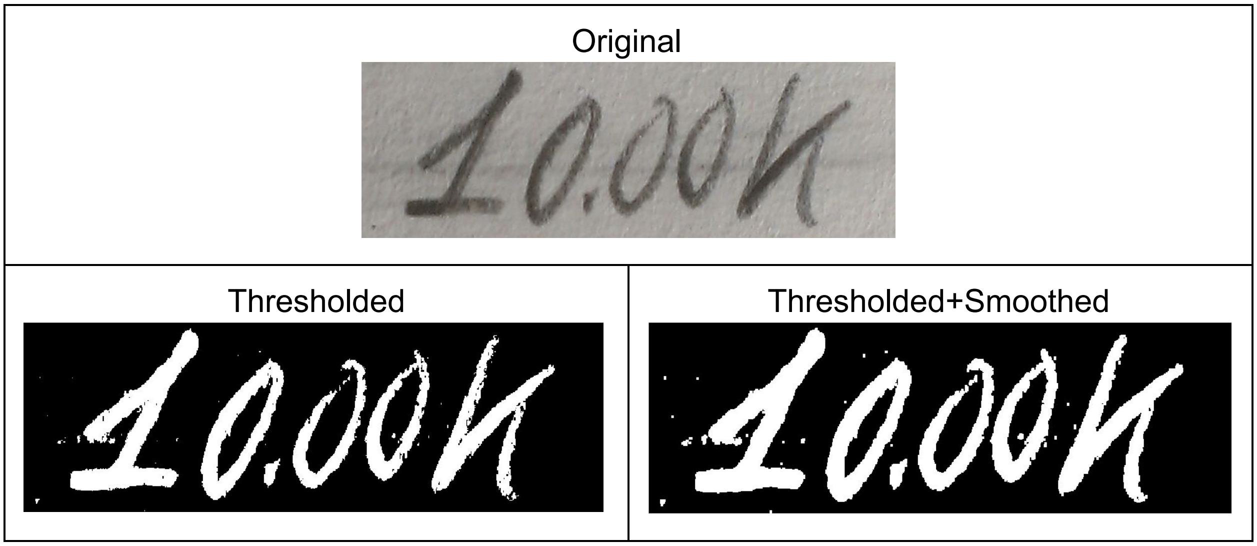


Figure 3. The first test image for thresholding, with results before and after smoothing

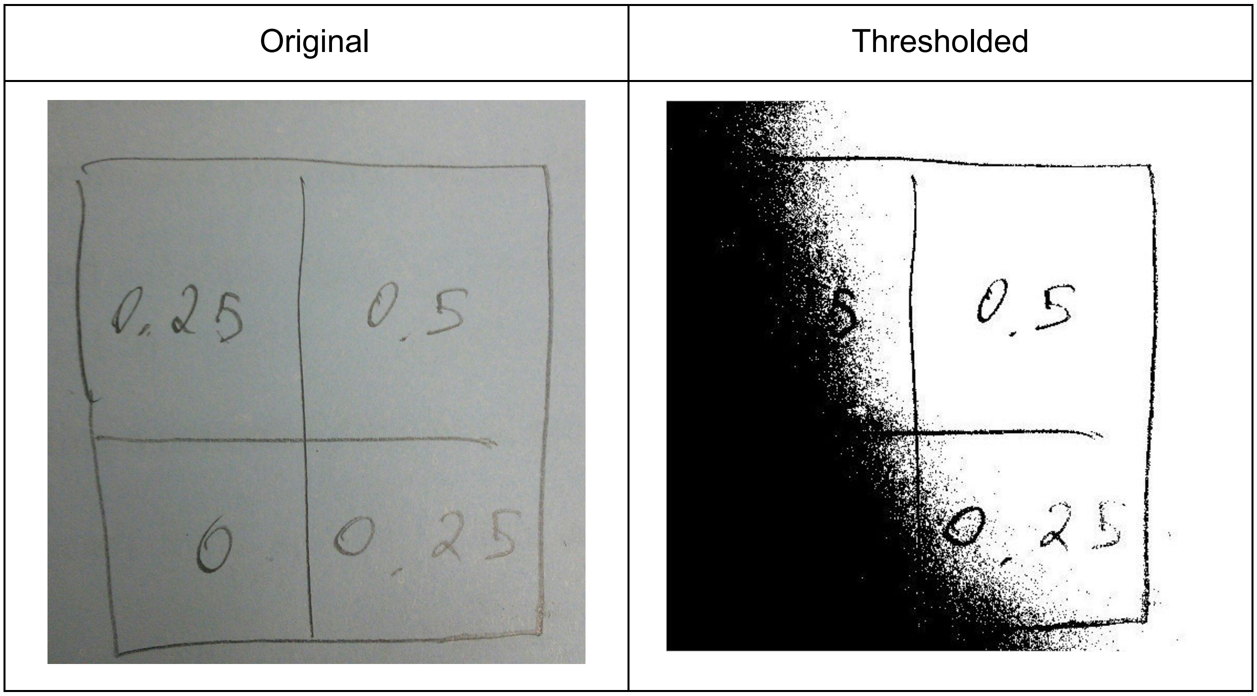


Figure 4. Example of imperfect lighting conditions leading to bad thresholding.

We later encountered problems thresholding as shown in Figure 4, where uneven lighting caused significant problems. Evening out the brightness of the image or detecting subimages first and then only thresholding those are two techniques that might handle the above case, but we thought of them too late to revise the project. (See src/Preprocessing/thresholdImage.m.) As can be seen in Figure 5, this approach is still sufficiently good for real test images.

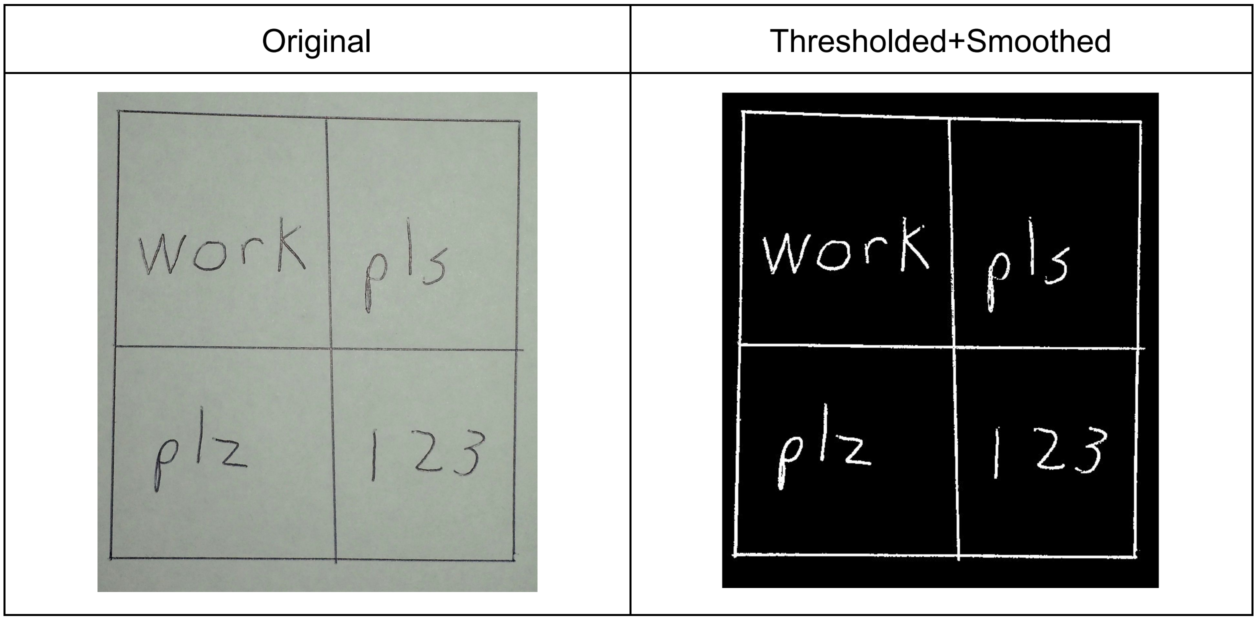
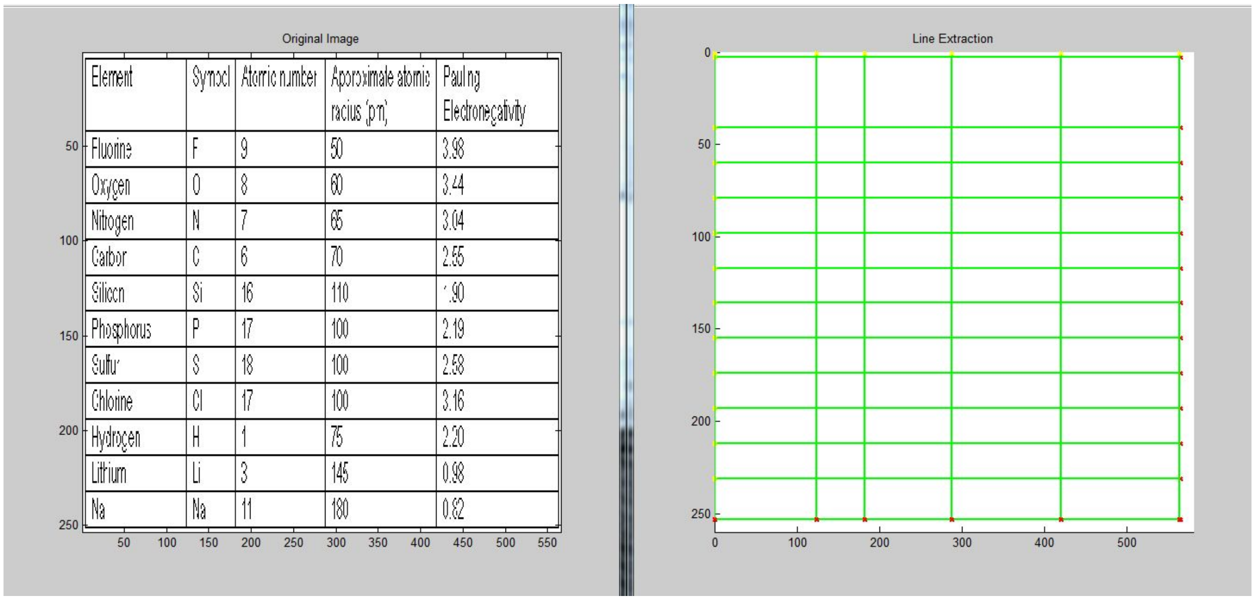


Figure 5. The example handwritten table for which we will present results.

3 Image Splitting

After thresholding and smoothing, the image needs to be split into the subimages of table cells. The code for this can be found in /src/Area Detection. We decided early that the easiest way to do this would be to detect straight lines in the image and use intersections of these to find the rectangular table-regions (cells). To find straight lines in the image, we use the Hough transform as implemented in Matlab. Additionally, we apply filtering to get rid of short lines that do not constitute table borders and crop the resulting table subimages to get rid of any extraneous line segments; some of the border is left in subimages and can interfere with character extraction.



Element	Symbol	Atomic number	Approximate atomic radius (pm)	Pauling Electronegativity
Fluorine	F	9	50	3.98
Oxygen	O	8	60	3.44
Nitrogen	N	7	65	3.04
Carbon	C	6	70	2.55
Silicon	Si	16	110	1.90
Phosphorus	P	17	100	2.19
Sulfur	S	18	100	2.58
Chlorine	Cl	17	100	3.16
Hydrogen	H	1	75	2.20
Lithium	Li	3	145	0.98
Na	Na	11	180	0.82

Figure 6. Hough lines detected for a computer-generated table.

Though Hough is a classic approach to finding straight lines, it was found not to be very robust; a lot of parameter fiddling is needed to get it work for any given image. A more basic approach of traversing the boundary and looking for corners might have worked better, but by the time we discovered Hough’s unreliability, it was too late to do anything about it.

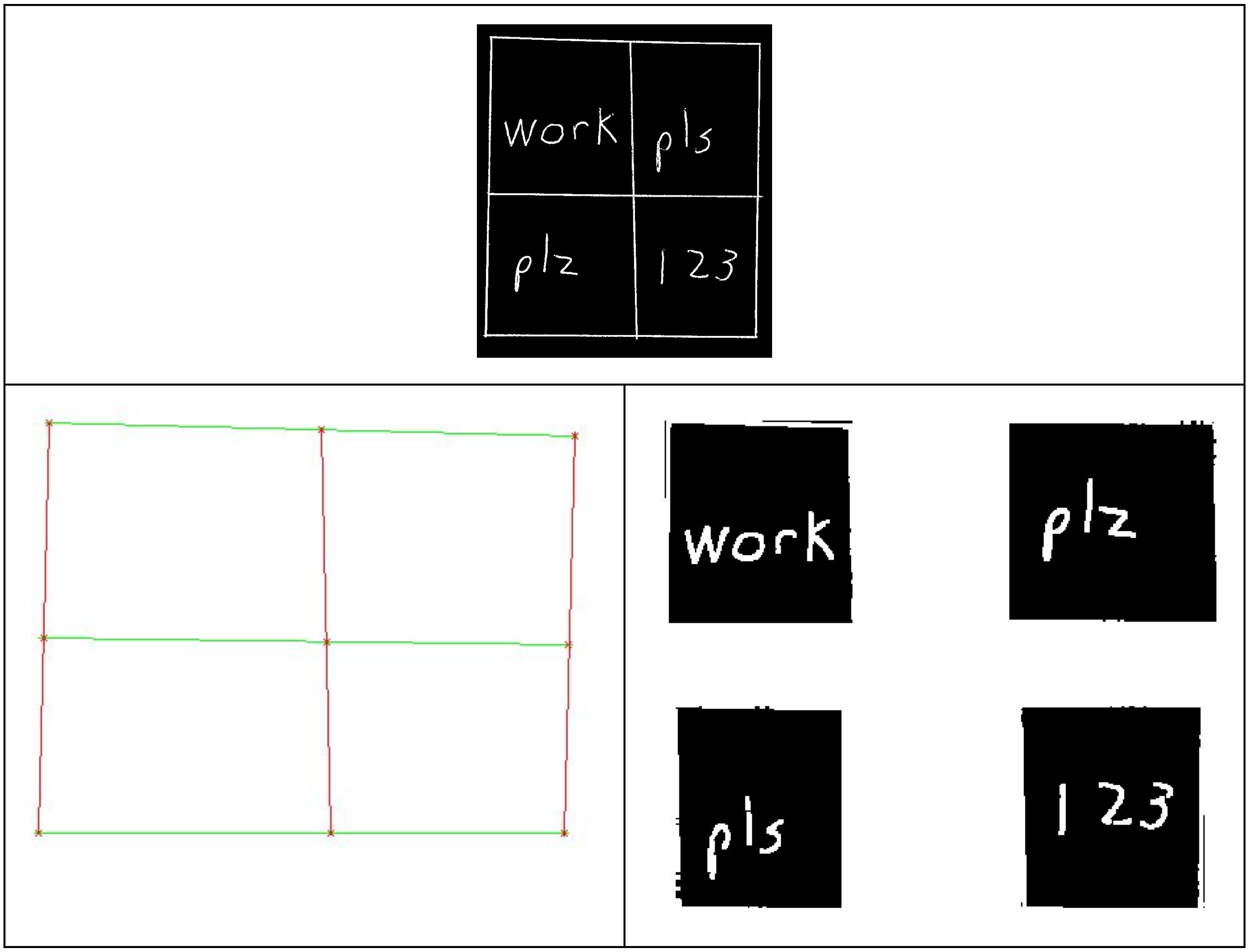


Figure 7. Hough lines detected for the test image and resulting subimages.

3 Character Segmentation

An essential step in performing text recognition in images is finding individual characters, since these most elementary objects are easiest to assign. A classic approach to this task is to find regions of contiguous white pixels on a black background and treat each such region as a character subimage. Though this is a simple algorithm, a custom implementation (in /src/Segmentation) was needed to run quickly and produce the results we wanted, as given in Figure 8.

Though connected components does perfectly for a printed image, it alone is not sufficient for noisy or handwritten images. Logic for filtering out small noise components as well as to recombine nearby components that were erroneously split is needed to deal with the imperfections in a thresholded image of handwritten characters. The additional logic is implemented in connectedComponentsSegmentation.m. As it is, the function segments the image displayed in Figure 9 perfectly. But more logic is needed to handle special cases such as general punctuation and split letters such as ‘i’.

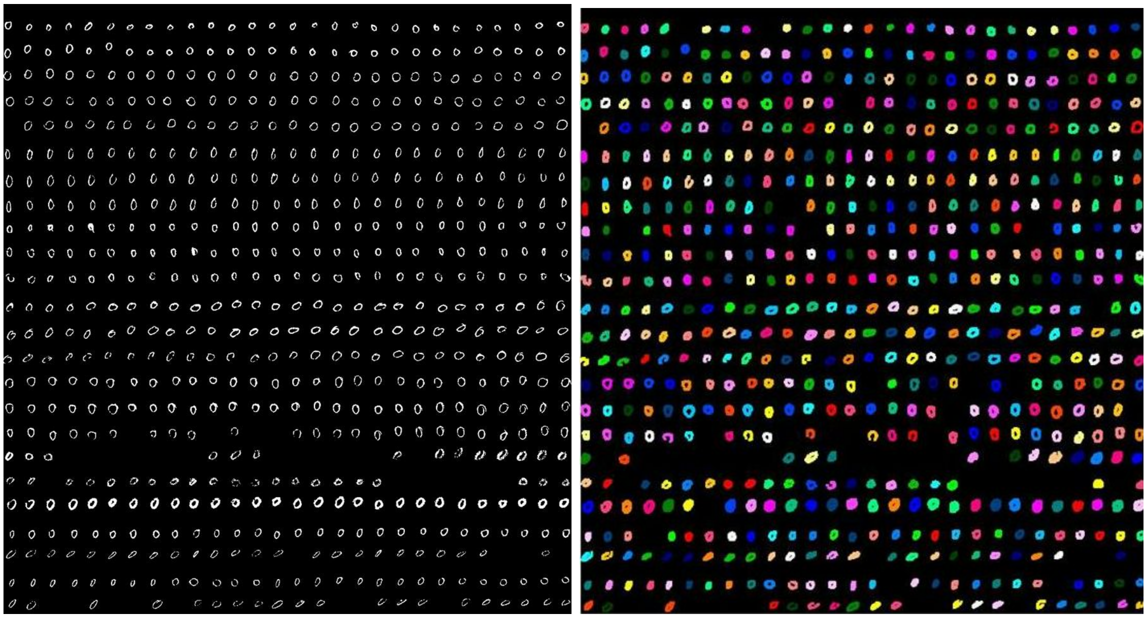


Figure 8. Results for connected components over a noisy image with many characters

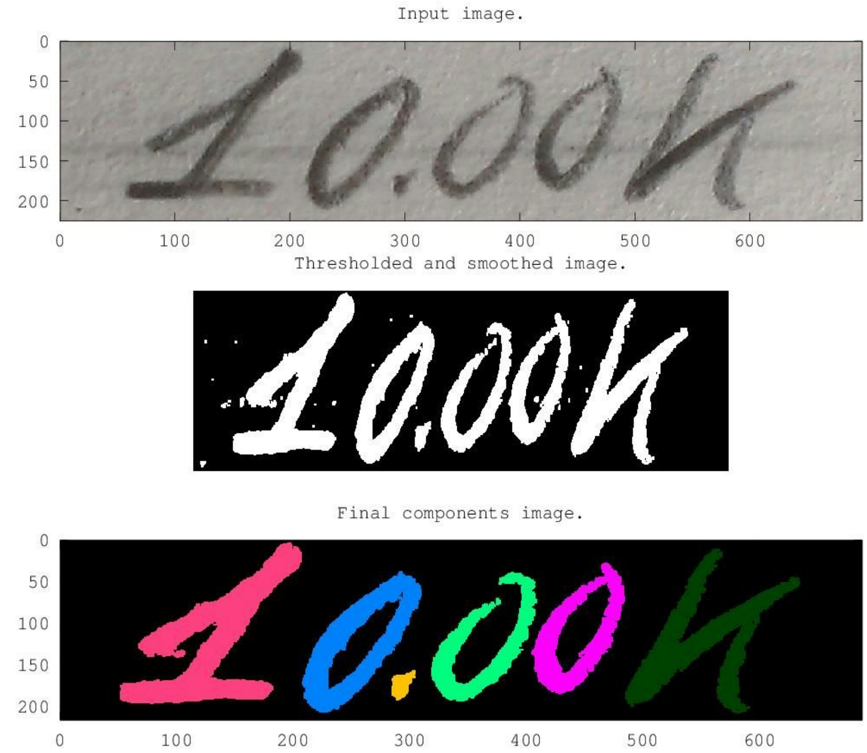


Figure 9. Segmentation results for the initial test image of the algorithm.

Character segmentation is executed separately for each subimage found in the initial image-splitting step (see Figure 7), so a set of character images is generated for each subimage (see Figure 10) . Each character image is then classified, and the classifications and character image locations are used to recombine the individual characters into one overall string for that subimage.

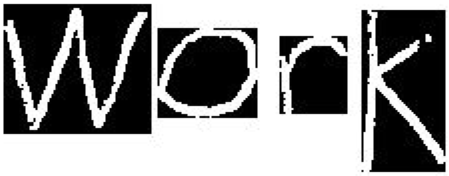


Figure 10. Segmentation results for the initial test image of the algorithm.

4 HOG Feature Extraction

To classify images of characters requires a supervised machine learner. All of these work by taking in examples, where each has a list of features and some sort of answer, and attempting to find a function that best maps from features (inputs) to given answers (outputs). Future examples are then classified based upon extractable features.

The first step of the process, then, is to extract features. There are many possible kinds of features that can be extracted from an image: color data, intensity, contrast, texture, et cetera. But of them, histograms of oriented gradients are among the most useful and consequently common.

A single histogram is found for a small region of an image called a “cell” (a few pixels by a few pixels--6x6 for us) by finding the magnitudes and directions of image-gradients in the cell and incrementing histogram “buckets” based upon these properties. The number of buckets in a histogram corresponds to a number of angular divisions of the unit-circle (9 in our implementation). That is, each bucket has a defining central-angle. A given gradient vector will contribute to filling the two buckets with central angles closest to its direction. How much these buckets is filled by this vector depends upon its magnitude and how nearly it points to the central angle of each. The update looks like

bucket += magnitude*(angle - bucket central angle)/(360/number of buckets).

Once histograms are found for all cells, groups of neighboring histograms are summed over “blocks” (2x2 cells for this project), and the normalized result is stored as part of a feature-vector defining the image. The block-window is moved by over one cell and the process repeated until normalized histograms for all possible blocks have been found.

Though the graphical meaning of HOG features is lost on a machine learner, it is possible to use knowledge of bucket central-angles and image dimensions to rearrange these features to fit over the image from which they were generated. This is a useful sanity-check for humans. Figure 10 shows an example, an output of the testHog.m script.

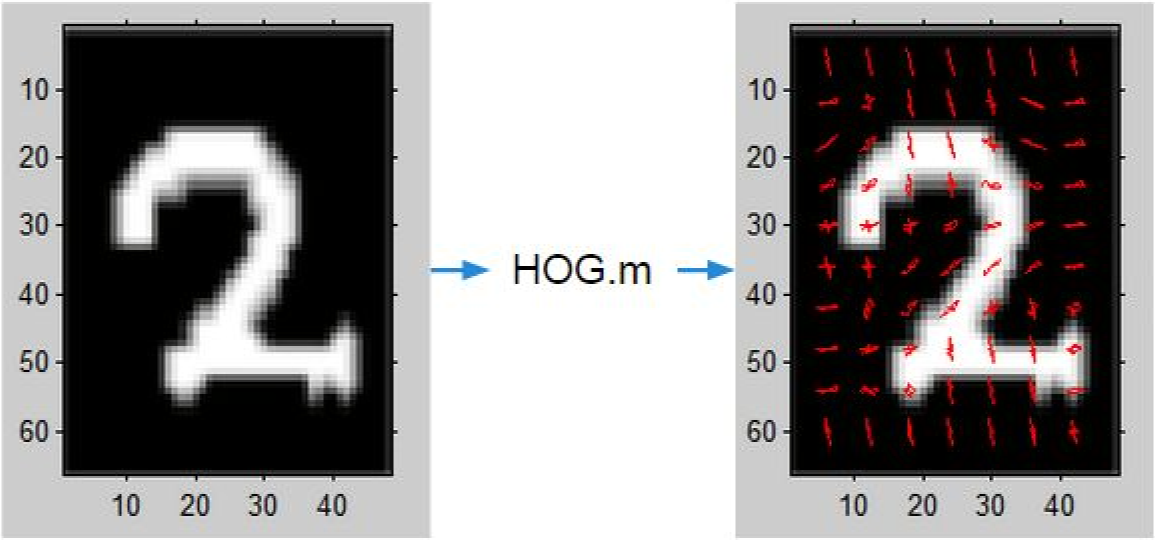


Figure 11. A visualization of HOG features superimposed upon the source image.

For code relating to HOG feature extraction, see src/Feature Extraction.

5 Classification

Ferns

To classify characters, we tried two approaches.The first of these, called Random Ferns, did not rely on HOG or any other systematically-generated vector of features. Instead, Ferns rely upon random features and the fact that noise tend to cancel over enough samples.

Ferns work by randomly selecting pairs of points in a binary image. The value of the image at these two points is XORed, and the result stored as a bit in a number. This is repeated many times for many images of different classifications. A histogram of the numbers returned after each set of point-picking is generated for each class, and later examples are classified based upon which histogram is closest to theirs.

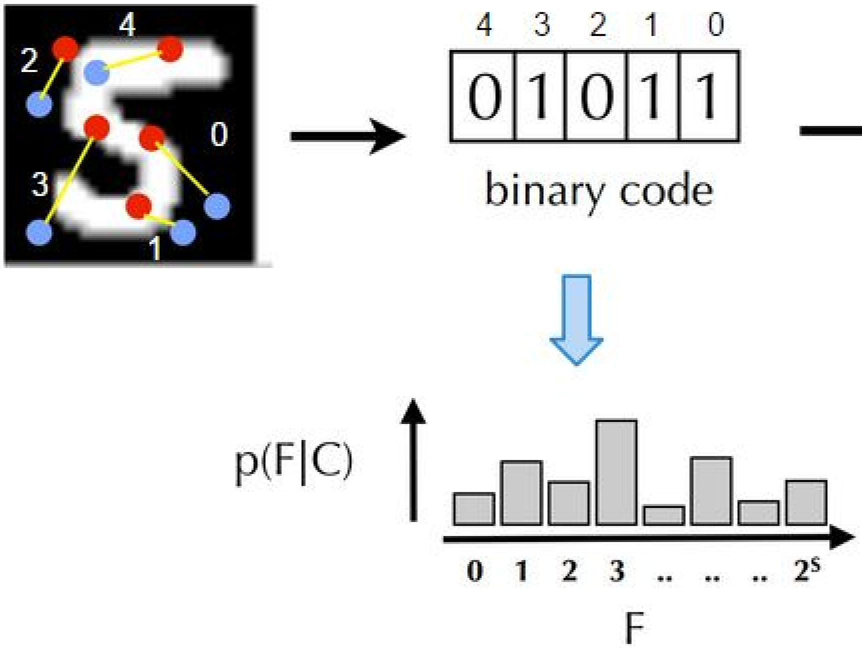


Figure 12. An illustration of how random ferns work.

But in the end, the best results we ever got with Ferns, on a digits-alone, dataset, were around 83%, not great for machine learners. So we decided to try something else.

ELMs

Extreme Learning Machines are a special kind of Artificial Neural Network in which there is only one hidden layer, the weights and biases of which are random.

There are two kinds of ELM: single-output node and multi-output node. The first we ever created was single-output and classified small numbers of examples very well. But with more training data it become necessary to create a system of multiple, binary output nodes. In this scheme, each output node corresponds to a particular classification and is trained to return a value proportional to its confidence that the input is of that class. This proved much more reliable.

In the end we have two ELM functions, one to take in features and classifications and produce and save weights, biases, and a beta matrix, the other to take in these parameters with new features and return F, a matrix containing confidence information from every output node for every example. The classification is taken to be that corresponding to the output node with highest confidence. For an extremely detailed description of how our ELMs work, see the comments at the top of src/ELMClassification/ELMtrain.m.

The code for an ELM is incredibly simple, but the math can be difficult to understand.

Figure 12 shows all the most important equations. H is a matrix generated by multiplying input features (x) by random weights, adding random biases, and passing the results through a sigmoid function. The result represents the output of the hidden layer of nodes. T is a matrix of answers, of desired outputs. Each row is a binary string where a 1 appears at the location corresponding to the correct classification. Beta is generated by multiplying the pseudoinverse of H with T. Output node confidences (F) for new examples are generated by finding a new H matrix for that data and multiplying by beta.

$$\mathbf{H} = \begin{bmatrix} \mathbf{h}(\mathbf{x}_1) \\ \vdots \\ \mathbf{h}(\mathbf{x}_N) \end{bmatrix} = \begin{bmatrix} h_1(\mathbf{x}_1) & \cdots & h_L(\mathbf{x}_1) \\ \vdots & \vdots & \vdots \\ h_1(\mathbf{x}_N) & \vdots & h_L(\mathbf{x}_N) \end{bmatrix}$$

$$\mathbf{T} = \begin{bmatrix} \mathbf{t}_1^T \\ \vdots \\ \mathbf{t}_N^T \end{bmatrix} = \begin{bmatrix} t_{11} & \cdots & t_{1m} \\ \vdots & \vdots & \vdots \\ t_{N1} & \cdots & t_{Nm} \end{bmatrix}$$

$$h_i(\mathbf{x}) = G(\mathbf{a}_i, b_i, \mathbf{x}), \mathbf{a}_i \in \mathbf{R}^d, b_i \in R$$

$$G(\mathbf{a}, b, \mathbf{x}) = \frac{1}{1 + \exp(-(\mathbf{a} \cdot \mathbf{x} + b))}$$

$$\boldsymbol{\beta} = \mathbf{H}^\dagger \mathbf{T}$$

$$\boldsymbol{\beta} = \mathbf{H}^T \left(\frac{\mathbf{I}}{C} + \mathbf{H}\mathbf{H}^T \right)^{-1} \mathbf{T}$$

Figure 13. Essential mathematics for training an ELM.

Increasing the number of hidden nodes increases the order of the ELM’s classification function. In other words, the ELM can classify better with more nodes. (See Figure 13.) It’s still only single-layer, though, so adding more nodes is not very beneficial past a certain point. More nodes unfortunately also means geometrically-more computations in the process of inverting H, so training an ELM with more than a few thousand nodes may be impractical. However, feed-forward is fast no matter the ELM size, so using as many nodes as possible became our goal. Some groups have successfully trained ELMs with many tens of thousands of nodes on extraordinarily powerful computers. With the best computer available to us, we could train a learner with, at most, 19000 hidden nodes. This is the classifier in use today.

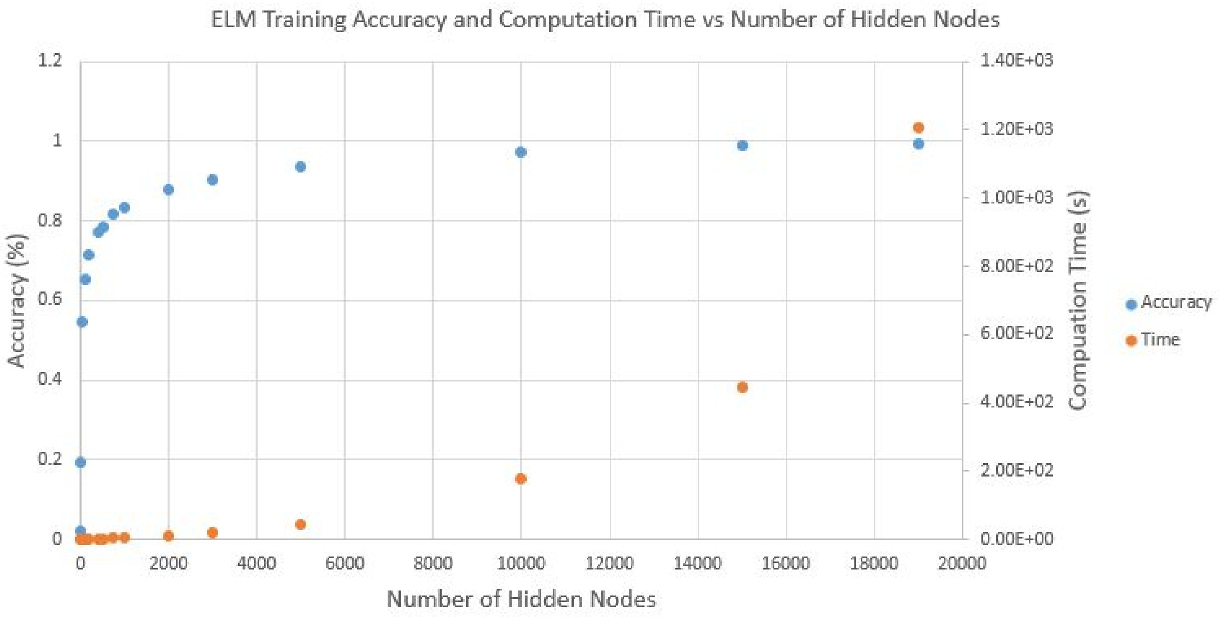


Figure 14. ELM training accuracy and computation time vs number of hidden nodes.

Datasets

It would be remiss not to mention the importance of good datasets in the process of training a learner. We ran into problems with some earlier versions of our classifier because the training dataset contained many more numerals than alphabetic characters. As such, the classifier was far more likely to think a character was a number than a letter, and outputs were horribly misclassified.

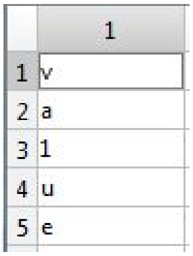
In addition, we decided that since many characters appear similar, especially when hand-written and resized to fit through our feature extraction/classification pipe, it made sense to actually combine some into larger classes and separate at a later stage with some logic. The classes for the current system are: (0,O,o), (1,I,l), 2, 3, 4, 5, 6, 7, 8, 9, A, B, D, E, F, G, H, J, K, L, N, Q, R, T, Y, a, b, (C,c), d, e, f, g, h, i, j, k, (M,m), n, (P,p), q, r, (S,s), t, (U,u), (V,v), (W,w), (X,x), y, (Z,z). Ambiguous number/letter combinations are set by default to their numerical form. All lowercase Ls appear as 1s after the classifier, for example. And all ambiguous letters are returned in their lowercase form.

6 Character Synthesis

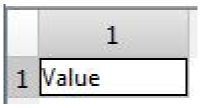
Once individual characters of the table data are classified through the use of HOG and ELM, the next step is to reconnect the individual characters back into their original form. In our case, the classifications are passed, cell by cell, to our synthesis function (in /src/synthesizer.m) as a cell array of characters. The characters are extracted from the cell array and concatenated to form a string that represents the content of the cell (see

Figure 15). Synthesis is ran on all cells with the output string of each cell of the table placed in a cell of a cell array mirroring its location in the table. (See Figure 16.)

.



	1
1	v
2	a
3	1
4	u
5	e

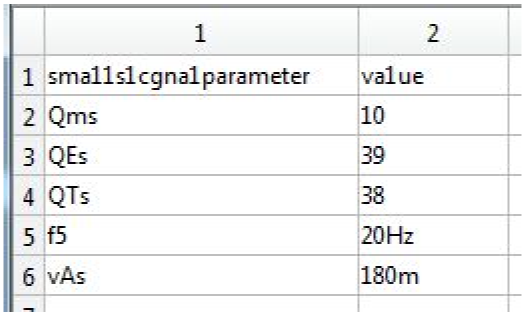


	1
1	Value

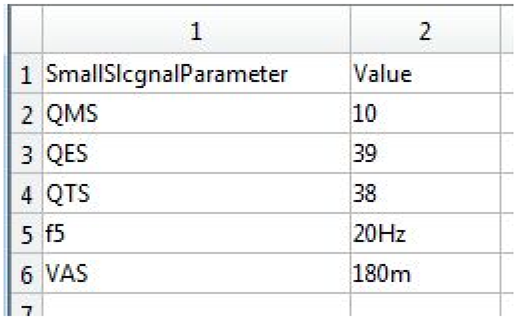
Figure 15. Input and output of character synthesis.

Character synthesis also handles the classification of letters whose lower/upper cases have similar structure as mentioned in the classification section. In order to determine if these letters are upper/lower case, the height of the corresponding letter's bounding box is used. A threshold of a fraction of the max bounding box height is applied. Those whose bounding box height were subthreshold were changed to their corresponding lower case form while those that exceed threshold were not changed.

Classification of letters which have similar shape as a number (O and 0, 1 and I) are also distinguished between each other during synthesis. Since the letters does not have a significant difference in height to their number "counterpart", their height cannot be used to distinguish between the two. Instead, we used the tendency that there is a higher chance of a certain type being followed by its own type (i.e. letter followed by a letter) as oppose to being followed by a different type (i.e. letter followed by a digit) in order to distinguish the type of the character in question. Figure 12 shows the effect of the implementation of these logic.



	1	2
1	sma11s1cgna1parameter	va1ue
2	Qms	10
3	QEs	39
4	QTs	38
5	f5	20Hz
6	vAs	180m
7		

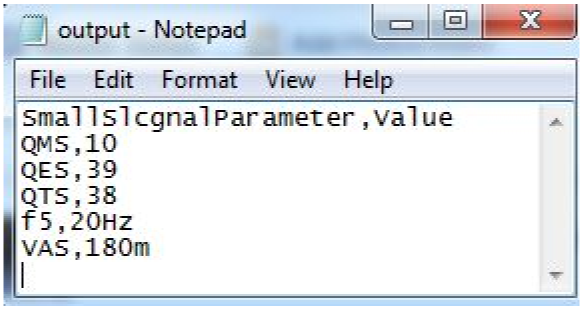


	1	2
1	SmallSlcgnalParameter	Value
2	QMS	10
3	QES	39
4	QTS	38
5	f5	20Hz
6	VAS	180m
7		

Figure 16. Left image shows result without intelligence implemented while right image shows result with intelligence implemented.

7 CSV Output

Having the contents of each cell of the table in a cell array with the same dimension, the cell array contents are simply accessed one by one and written into a csv file. Commas are inserted in between cell contents and the new line character “\n” is inserted at the end of each table row. A function was written (in /src/saveTableToCSV) in order to accomplish this. Figure 13 shows the input and output of the function.



```
SmallSlcgnalParameter,Value
QMS,10
QES,39
QTS,38
f5,20Hz
VAS,180m
```

Figure 17. Output of saveTableToCSV.m

8 Learning Outcomes

We learned first hand how difficult it is to deal with real world conditions.We finished the elements of the pipeline for printed tables relatively early in the semester. Every step worked as it was supposed to, so we assumed that the next step of running our code on real world images would not be a big jump. We were wrong. There are so many external factors; it exposes how difficult it is to perform threshold finding. Uneven lighting severely confuses the binarization threshold. Even the slightest curvy lines cause Hough transform to not detect table lines. Slight gaps in solid black letters cause segmentation to break components erroneously into smaller components.

However, in the end, we were successful in running the code on a handwritten table. While constructing this table, special attention had to be put on even lighting and straight lines. Also, the lines had to be clearly black and sharply defined. Careful care was made to not overlap any letters so that segmentation would not segment multiple letters into one component.