Recognition of Online Handwritten Mathematical Expressions

Karanveer Mohan (kvmohan) and Catherine Lu (cglu)

CS221 Project Final Report
Stanford University

December 16, 2013

1 Introduction

A fair amount of work has been done thus far towards automatic human handwriting recognition. Digit recognition has been well-studied using the MNIST dataset, and its classification approximates human performance [1]. Character recognition has been similarly well-studied [2]. In addition, a simple Google search yields results of small projects many have done with respectable outcomes close to perfect accuracy. Thus, character and digit recognition have been well-studied and achieved levels comparable to human performance.

There have been many papers published to date on general handwriting (English words and numbers) [5]. Handwriting recognition is comprised of two general parts: offline recognition, where the text was written in a non-digital medium (e.g. a handwritten letter), and online recognition, where the text was written on a special digitzer where the sensor picks up the pen-tip movements (e.g. using a stylus to take notes on a tablet). Though offline handwriting accuracy still lags behind online handwriting accuracy, it is accurate enough now to be used in real-world systems such as interpreting handwritten postal addresses or monetary values on bank checks [5]. Online handwriting has the benefit of including additional metadata such as stroke information, which can be used by the algorithm.

Less progress has been made surrounding handwritten equation recognition. The 2012 Competition on Recognition of Online Handwritten Mathematical Expressions (CROHME) [3] had 7 serious competitors. The system that was best by far, and the only system with better than 50% accuracy on all tests, was submitted by Vision Objects. Vision Objects has online demos for the handwriting and equation recognition engine [4], and there is no doubt that their technology is great. However, Vision Objects is also a privately held company with proprietary technology. Thus, we decided to tackle this problem for our 221 project.

2 Data

Our data comes from CROHME, which provides datasets from the 2011 and 2012 competition freely available for research purposes. The data is in Ink Markup Language (InkML) format, which is a type of XML data that is specifically used to describe digital writing [6]. The entire content is contained within a single `<ink>` element, and `<trace>` element elements within that represent strokes. The `<trace>` element elements themselves are comprised of points captured as X-Y coordinates.

```
<ink xmlns="http://www.w3.org/2003/InkML">
   <trace>227 50, 226 64, 225 78, 227 92, 228 106, 228 120, 229 134</trace>
   <trace>282 45, 281 59, 284 73, 285 87, 287 101, 288 115, 290 129</trace>
   <trace>366 130, 359 143, 354 157, 349 171, 352 185, 359 197</trace>
</ink>
```

Above: Sample InkML data with three strokes.

For the past two competitions, they released the following data sets:

Name	Year	Number of equations
CROHME train	2011	921
	2012	1338
CROHME test	2011	348
	2012	488

Table 1: Data set from CROHME.

Each of these contains the ground truth associated with them, with strokes corresponding to symbols along with an entire equation-level ground truth.

We went through all equations in the train data set to create a new data set of 17,000 (normalized) handwritten symbols and their corresponding truth values. The average number of symbols per equation was 11.6 symbols. We also have equation-level data, where all of the equation strokes have a corresponding equation truth value. This equation truth value also has more information at the symbol-level, such as the strokes used to make up each symbol.

3 Overview

Our pipeline has three distinct phases:

1. **Image Segmentation:** Breaking down the equation into specific characters.
2. **Classification:** Classifying each character based on the image, returning a probability likeliness. This is done first by normalizing and converting each character into pixel images, and then classifying each character.
3. **Hidden Markov Model (HMM):** Putting together characters with an HMM, using relative location of the character in the original image and possibly other signals.

4 Image Segmentation

In order to recognize an entire equation, we broke down an equation into its component symbols. Once we had these symbols, we could classify them and then either use an HMM (see Section 6) or use special factors to recognize whether a symbol is a superscript, subscript etc. (see Future Work section).

	Symbol (all)	Equation (all)	Symbol (*)	Equation (*)
Intersection	81%	17%	89%	45%
Intersection & SVM	87%	35%	92%	53%

Table 2: Summary of our segmentation accuracies.
*Ignores $=, i, j, \leq, \log, \sin, \cos, \lim, \geq, \rightarrow, \div$.

4.1 Image Segmentation Heuristics

We wanted to start trying to segment the equations using a simple heuristic, since it would be easier to implement and lead to high accuracy. Initially, we tried to segment the equation into symbols by drawing vertical and horizontal lines between characters. The idea was if we had an equation of the form $a^2 + b^2 = c^2$, we could segment it as $|a|^2| + |b|^2| = |c|^2|$. Similarly, we could horizontally segment fractions and then vertically segment the linear subparts. However, we soon realized that segmenting characters in this manner had issues. While this would work well if there was adequate spacing between symbols, that is often not that case. $\sqrt{2}$ for example would not be split since the $\sqrt{}$ sign often goes

above 2. Moreover, handwritten equations often look like Figure 1, where it is very hard to separate the parentheses, the commas or the for all (∀) sign from the variables.

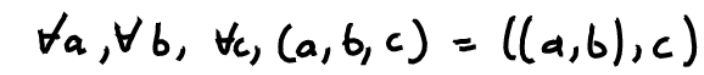

Figure 1: An equation from the dataset where horizontal and vertical splitting doesn't work.

We ended up discarding the idea and began recognizing symbols by trace intersection. Our algorithm detects whether two traces intersect and if they do, it infers them to be part of the same symbol. This ends up working reasonably well. Using intersections alone, we exceeded an 80% segmentation accuracy.

Most of the errors were in symbols such as log, cos, sin, lim, = etc. which generally have multiple strokes without intersections and hence are classified as two or more different symbols instead of one. However, symbols such as x and k were also not segmented well, with two-thirds of the x's being segmented as two symbols instead of one. Upon further examination, a lot of our InkML strokes for symbols such as x and k were not intersecting at all, but gave the appearance of touching due to the thickness given to the strokes. For instance, x was often written with two non-intersecting strokes as ')('. We added some proximity heuristics, seeing how close points from two different strokes were. Unfortunately, while these proximity heuristics helped reduce errors in segmentation for symbols like x and k, errors were added for other symbols. In particular, for the summation symbol, the text above and below the symbol are extremely close to it, and therefore were all being considered part of the same symbol.

4.2 SVM Classifier & Heuristic

Clearly, we needed something more than heuristics. We ended up using our SVM classifier (described later) to detect symbols like x, =, $\geq$, $\leq$ etc. If two consecutive strokes did not intersect, we used our SVM classifier to see if the two strokes were classified as a symbol like x, =, etc with high probability ($> 70\%$). If this were the case, we would then consider the strokes part of the same symbol.

Using intersections and the SVM classifier together, we achieved a segmentation accuracy of 87.1% with 34.9% of the equations being perfectly segmented.

Figure 2 shows that sin, lim, and cos were together responsible for about 50% of the segmentation errors, with a number of x, =, k, π, etc. still misclassified. For future work, the segmentation accuracy can be improved by extending our algorithm to handle an arbitrary number of strokes and see if lim, sin, cos etc. can be detected with high probabilities instead of merely checking if two strokes form a commonly incorrectly-segmented symbol with high probability.

A question that comes up is why we used intersection algorithms at all instead of solely using SVMs to see how many strokes per symbol would give us the best accuracies. Firstly, it was much easier to implement and much faster, thereby allowing us to make quick progress and gave us reasonably high accuracies to begin with.

Secondly, given the features we had, we would not have been able to achieve high segmentation accuracy with SVMs alone. For example, consider the plus symbol '+'. If we did not use intersection information at all, the strokes may be classified as 1 and - with higher accuracies than it being classified as +. An equation such as 2 + 3 could easily be classified as 21 - 3, even if factor graphs were used, since our features do not contain information regarding the position of the stroke on the canvas.

5 Classification

5.1 Strokes to Pixels

With the segmented symbols, we now went on to create an N x N pixel images (we experimented with 3 pixel array sizes: N = 18, 24, and 36) with each symbol taking the same size so that the longest height or width of the drawn character is set to 18 pixels (e.g. a subscript ^{2} and normal 2 will look roughly identical). These pixel images were created by converting stroke information of X-Y coordinates, drawing lines with appropriate slope between consecutive coordinates and translating that information into pixel

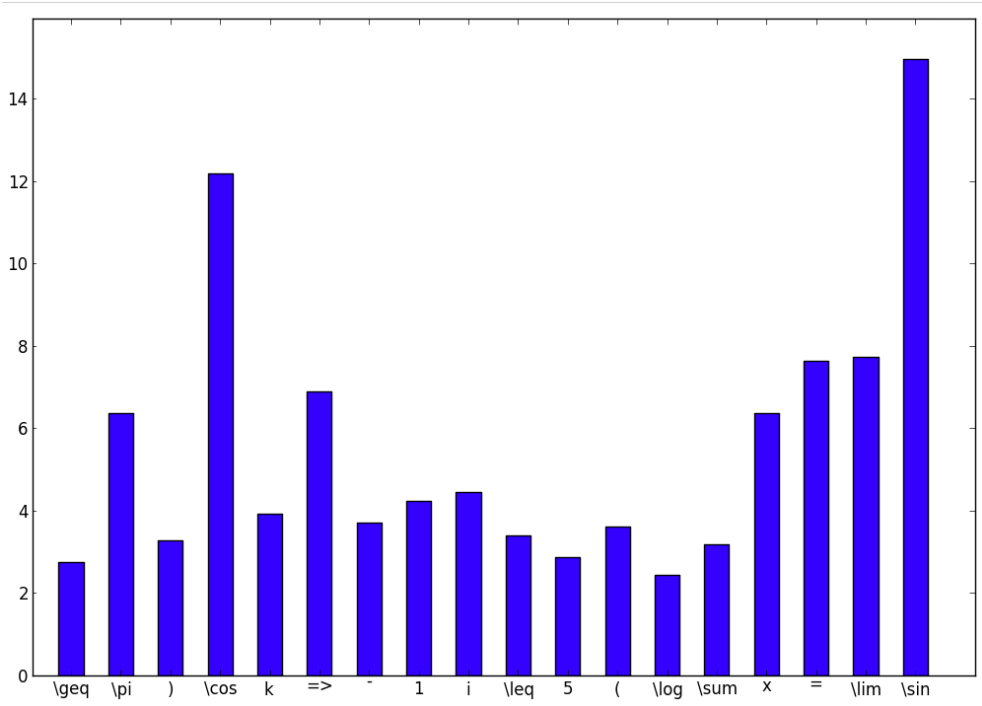

Figure 2: Symbols and their percent contribution to overall segmentation errors. Symbols with low error contributions have been removed for clarity.

representations of the points being added. So a symbol like Figure 3a that is represented in InkML as strokes with many points (Figure 3b) would be translated to an 18 x 18 pixel image (Figure 3c):

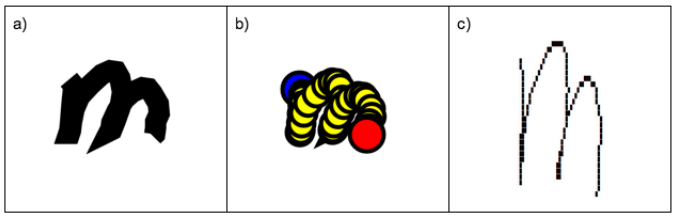

Figure 3: a) Symbol b) InkML representation c) Normalization and pixel representation.

The reason we normalized and converted INKML strokes into pixel images was to gain more descriptive features from the stroke information. For example, a symbol drawn larger would have more points than a symbol drawn smaller in InkML. Having a fixed-size pixel array would be a better means of classifying symbols.

We used k-nearest neighbors (KNN), a support vector machine (SVM) with a Gaussian kernel and logistic regression on these pixel images and managed to reach an accuracy of slightly less than 70% using SVMs and logistic regression, and less than 40% for KNNs (see Table 3).

Since KNN performed so poorly in all categories, we discarded it as an algorithm of choice moving forward with additional features. Upon adding additional features, SVM did much better than logistic regression. As table 3 shows, logistic regression was highly overfitting the training set whereas the SVM did not seem to overfit and performed better than logistic regression upon adding further features.

5.2 Improving Features

We improved the features with the following:

1. **Number of strokes** was added to the feature vector. The motivation behind adding number of strokes as a feature was that the number of strokes in most symbols is almost always the same. = is always two strokes, and 'a' is always one stroke. Using the number of strokes, we hoped to improve accuracy by reducing misclassifications between symbols such as 1 and i.

2. **Pixel array blurring** was done by changing the pixel information from 0's and 1's to a distribution with mean 1 on the pixel and noise with 0.2 decreasing information in the surrounding pixels for every pixel distance away (defined by euclidean distance). We did this because we noticed that symbols were being classified differently, when placed slightly to the left or to the right. For example, = was classified differently depending on the distance between the two parallel lines. The pixel array blurring added thickness to the strokes, making them more robust to minor variations.

3. **Pixel array size reduction** was done on the pixel array, changing its size from 36x36 to 18x18 ultimately to decrease the number of features in our feature vector and reduce computation time. We thought that decreasing the number of features for our pixel array could increase accuracy, since a lot of the information becomes redundant as the size of the pixel array increases. Dealing with a smaller pixel array also reduces computation time.

With these feature improvements, we achieved a maximum symbol classification accuracy of 87% on the test data set (using ground truth). Considering that the classification is with 75 different symbols, the accuracy seems decent.

	Additional Features	Train Precision	Test Precision	Test Recall
Logistic Regression	None	98%	64%	64%
	1	99%	67%	67%
	1, 2	97%	71%	71%
SVM	None	98%	64%	64%
	1	70%	66%	70%
	1, 2	83%	84%	83%
	1, 2, 3 (24 x 24)	87%	87%	87%
	1, 2, 3 (18 x 18)	87%	86%	87%
KNN	None	51%	58%	34%

Table 3: Per symbol classification accuracy using Logistic Regression, SVM and KNN and additional features. Logistic Regression overfit the data with 99% train precision and only 71% test precision.

5.3 Segmentation & Classification Symbol Accuracies

We then switched from using ground truth in the test data set to use our own segmentation algorithm to get the symbols. Considering that 13% of the symbols were segmented incorrectly to begin with, we classified 87% of our symbols using the SVM. Among the symbols correctly segmented, we achieved a precision of 88% thereby resulting in the overall precision for segmented symbols to be 77%.

	Additional Features	Train Precision	Test Precision (*)	Test Precision (†)
SVM	1, 2, 3 (18 x 18)	87%	88%	77%

Table 4: Per symbol classification accuracy using symbols segmented by our segmentation algorithms.
*Segmentation was done using ground truth segmentation.
†Segmentation was done using our heuristic and SVM model.

5.4 Segmentation & Classification Equation Accuracies

At an equation level, of 864 equations in the test data set, 294, i.e. 35% were classified completely correctly (using ground truth). 64% of the equations were misclassified at most on one symbol and 80%

of the equations were misclassified at most on 2 symbols. The average length of an equation was 11.6 symbols, so accuracy was quite good. Using our own segmentation algorithm, where only 35% of the equations were perfectly segmented to begin with, we reached a 16% accuracy. 37% of the equations had at most one symbol incorrectly segmented or misclassified and 56% had at most two.

	Perfect Classification	Missed $\leq$ 1 Symbol	Missed $\leq$ 2 Symbol
Test set using ground truth	35%	64%	80%
Test set without ground truth	16%	37%	56%

Table 5: Classification accuracies on a per equation level.

6 HMM: From Pixels to Equations

Using segmentation followed by using an SVM to classify this feature vector allowed us to classify strokes into symbols. However, we also wanted to explore how to better improve our accuracy. So, the final step was an effort to boost symbol classification by creating a factor graph with potentials between subsequent symbols. The binary potential was a measure of the likeliness that the symbol being labeled would follow a previously labeled symbol. For the first symbol in the equation, a special BEGIN label was assigned. This was very similar to the NER assignment done in class. The unary potentials were the probabilities from training the SVM. The factor graph was solved using Gibbs sampling.

Some results below are as follows. Note that these accuracies are from the ground truth, so that segmentation is perfect. Also, the number of burn ins for Gibbs sampling was also changed (between 100 and 1000) with no noticeable change in accuracy, so it has been excluded from the table.

Pixel array size	No. of samples	Other characteristics	Test Precision
24x24	500	None	77%
24x24	2000	None	78%
24x24	2000	Laplace Smoothing	84%
18x18	500	None	78%
18x18	2000	None	79%
18x18	1000	Laplace Smoothing	85%
18x18	2000	Laplace Smoothing	84%
18x18	5000	Laplace Smoothing	84%
18x18	50000	Laplace Smoothing	85%
18x18	100000	Laplace Smoothing	85%

Table 6: Precision of HMMs.

With 75 possible labels and the average length of equations is 11.6, it seems obvious why 10,000 samples performed much better than 500. However, we were puzzled about why we did so poorly, even worse than the SVM alone although we are using additional information. For instance, classifying t with SVM alone used to have 97% precision, but adding HMMs dropped its value down to 11%. Upon taking a closer look, 't' was classified as '+' 20 times and 'x' 22 times, but only as 't' 7 times.

Table 7 provides the conditional probabilities of some common symbols. Overall, we found that HMMs increased the accuracy of common symbols even more but decreased the accuracy of less common symbols, resulting in no net benefit. Our data is skewed in the sense that the distribution of symbols is far from uniform – we have a lot of y's, but very few log symbols. Hence, the probability that a symbol will be followed by a y ends up being much higher than the probability it is followed by a log. It seems that adding in the conditional probabilities only favors the common symbols even more, and this hurts the overall accuracy even though it also adds some valuable information. Given less skewed data, we believe we would find overall accuracy improvement. The Future Work section also describes how the factor graph can be made better to improve accuracies.

For	Followed By	Conditional Probability
=	-	14%
	x	11%
	0	9%
	1	9%
	2	9%
1	-	29%
	+	24%
	0	4%
	6	4%
	1	3%

Table 7: Classification accuracies on a per equation level.

7 Web Application

In order to see our algorithms in action, we built a web application using Flask running on: `http://cs221.herokuapp.com`.

We felt that building a web application serves many roles. It helps us experiment with the algorithms and see which symbols work well and which don't. For example, the idea of using pixel blurring came when we noticed that '=' signs were being misclassified depending on the distance between the two parallel lines. It also gave us better intuition for when HMM was working and when it wasn't. Finally, by deploying this, we can potentially get more data and perhaps use online machine learning algorithms to further improve our accuracy.

8 Future Work

8.1 Classifying Additional Information

The major improvement left in our work is to not only classify symbols but also classify additional information about them. For instance, symbols could be classified as superscript or subscript. They could also be classified as a numerator or denominator for a fraction, or as being located above or below a summation or other mathematical symbol. This information is lost in our current algorithms. Given more time, we would have approached this problem with the following modifications and feature additions:

1. **Positional information** where the top of the symbol is in relation to the canvas, and where the bottom of the symbol is in relation to the canvas.

2. **Resize factor** included in the feature vector. The resize factor is the amount the symbol is resized to fill the pixel array, and it is a good way to capture the size of the user's input. For instance, a superscript or subscript would generally be smaller than other characters, so the resize factor may prove to be a good indicator for whether a character is special.

Assuming we are only dealing with a superscript, subscript, or normal classification, this increases the possible classifications three-fold. This would likely lead to much lower accuracy and require more data.

8.2 Improving Segmentation

Segmentation accuracies can also be improved by properly segmenting word symbols such as log, lim, sin, cos, and tan. A challenge that comes from these symbols is that cursive writing may not guarantee the number of strokes for each of these. One such way to properly segment these symbols is by creating special classifiers for each of log, lim, sin, cos, and tan. Then, at the beginning of segmentation, try classifying into each of these special cases one stroke at a time. If any of these reaches a probability above a certain threshold, take the classification with the highest. Otherwise, feed it into the normal segmentation algorithm.

Also, the HMM could be used in order to perform segmentation and classification simultaneously instead of treating them as discrete steps. Right now, Figure 4 would be segmented incorrectly since A and x intersect. However, if additional segmentation is done during classification and it becomes clear Ax isn't one symbol, we could try and separate A and x to get better segmentation and classification accuracies.

Figure 4: A case where our current segmentation fails. The strokes for A and x intersect.

8.3 Improving Classification

We could also improve the features we use in classification. The feature vector could be enriched with more information such as:

- The number of non-zero pixels.
- Edge information. For instance, add the following features:
  - Indicator variables for vertical lines of at least length 5, 10, 15, and 20.
  - Indicator variables for horizontal lines of at least length 5, 10, 15, and 20.
- Use a normal distribution for the pixel array blurring.
- The direction a stroke was drawn e.g. left to right or up to down.

Finally, by deploying our webapp, we can collect more data, reduced skewness and improve classification accuracies further.

References

[1] Dan Cireşan, Ueli Meier, and Juergen Schmidhuber. *Multi-column Deep Neural Networks for Image Classification*. URL: `http://arxiv.org/abs/1202.2745`.

[2] Øivind Due Trier, Anil K. Jain, and Torfinn Taxt. *Feature extraction methods for character recognition-A survey*. URL: `http://www.sciencedirect.com/science/article/pii/0031320395001182`.

[3] Harold Mouchère et al. "ICFHR 2012 Competition on Recognition of On-Line Mathematical Expressions (CROHME 2012)". In: *Frontiers in Handwriting Recognition (ICFHR), 2012 International Conference on*. IEEE. 2012, pp. 811–816.

[4] Vision Objects. *Demonstration Portal*. URL: `http://webdemo.visionobjects.com/home.html#equation`.

[5] R. Plamondon and S.N. Srihari. "Online and off-line handwriting recognition: a comprehensive survey". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 22.1 (2000), pp. 63–84.

[6] W3C. *Ink Markup Language (InkML)*. URL: `http://www.w3.org/TR/InkML/`.