Live Handwriting Recognition Using Language Semantics

Nikhil Lele, Ben Mildenhall
{nlele, bmild}@cs.stanford.edu

I. Introduction

We present an unsupervised method for interpreting a user's handwriting using language semantics. We collect time series data of a user's handwriting using a tablet and create clusters of the written symbols (or "glyphs"). Then we use language semantics to determine the most likely written text.

II. Language Semantics and Substitution Ciphers

We depart from the norm by not using prior data about the absolute shapes of characters, nor comparing a user's input to previous samples of handwriting from that user or any other user. We have the user input a sentence of glyphs and spaces $S = \{G_1, G_2, \ldots, G_m\}$ that represents some natural language phrase (usually 10-15 words). Given that the user is writing with alphabet A in language L, our ultimate goal is to recover the user's intended mapping from glyphs to characters in A using only a sample corpus of text written in L.

In order to do this, we first use a comparison function to find clusters of glyphs we think should map to the same character in A. If the true mapping from these clusters to characters in the original sentence is bijective, then we have reduced the problem to solving a traditional substitution cipher. A substitution cipher is an archaic technique for encrypting messages where the key is some permutation of the characters in the alphabet. All characters in the original message are mapped to their image in the permutation. For example, the phrase *"the cat in the hat"* could be encoded as *"can iec th can aec"* using the permutation $(a, c, e, h, i, n, t) \to (e, i, n, a, t, h, c)$.

A substitution cipher retains much of the semantic information of the original phrase in the form of letter frequencies and the relative positions of characters in each word in the phrase. This information is enough to solve the cipher quickly and reliably for phrases of 100 characters or more and to provide a list of very good guesses for shorter phrases (speed and accuracy depend on the length of the words and how many times letters repeat in the phrase).

Our process for analyzing input sentences involves the following steps:

1. Record a sentence of n glyphs.
2. Evaluate the similarity of all n^2 pairs of glyphs.
3. Create k clusters of the most similar glyphs.
4. Solve the ciphertext corresponding to that clustering.

III. Recording Glyph Data

We used a Wacom tablet to collect sample data. Each data point is a single "glyph," which is a vector of n 3-tuples $\{t_i, x_i, y_i\}$ where t_i is elapsed time since the first point recorded in the glyph and $p_i = (x_i, y_i)$ is a two-dimensional point in screen space. Once the user indicates they have finished inputting the glyph, the time data is normalized so that $t_1 = 0$ and $t_n = 1$. In addition, the glyph is scaled and translated so that its centroid is at $(0.5, 0.5)$ and the point farthest from the centroid lies on the perimeter of the square ranging from $(0, 0)$ to $(1, 1)$.

We save this normalized data but run all comparison operations on smoothed versions of each glyph. Smoothing is a two-step process. First, we upsample the glyph. Each glyph has between 20-50 points when recorded, and we resample the glyph at 500 evenly spaced intervals from $t = 0$ to 1. To upsample a glyph G at time $t_i = i/500$, we find the value of j between 2 and n inclusive such that $t_{j-1} < t_i \le t_j$. We then linearly interpolate the coordinates p_{j-1} and p_j of those points using $t'_i = (t_i - t_{j-1})/(t_j - t_{j-1})$ to get the sample $(1 - t'_i)p_{j-1} + t'_i p_j$ for time t_i.

After upsampling the glyph, we downsample to 100 points by taking a weighted average of all samples in

the high-resolution version of the glyph. Our sample at $t_i = i/100$ is $\sum_{j=1}^{500} N(t_j - t_i) \cdot p_j$. The function $N(x) = \exp(-x^2/2\sigma^2)/(\sigma\sqrt{2\pi})$ is the probability density of a normal distribution with mean 0 and variance σ^2, used to weight the points closest in time to t_i most highly. We set $\sigma = .01$. Lower values of σ preserve more detail from the raw sample such as sharp turns and twists in more "pointy" letters, but higher values are more effective for smoothing out the noise in the raw data. Since we do not make note of when the pen is lifted, smoothed glyphs usually interpolate some points between pen-up and pen-down points, which connects consecutive strokes in letters like 't', 'x', 'j', etc.

IV. Comparing Glyphs

We tested multiple features for measuring the similarity between glyphs. Since we collected live data, all of our features are time series. This allows for easy comparison between glyphs, since we avoid the expensive and tricky process of trying to match spatial points in different glyphs.

This assumption means that any feature we use will not correctly match two examples of the same character drawn in different ways; if a user draws some glyphs representing the character 'd' starting from the loop and some from the top of the stem, the matching in time will not be the best possible matching in space. We will address this problem in section VI.

A. Shape Contexts

The richest and most successful feature we use is the shape context for each point in each glyph, as presented by Belongie, et al. [1]. The shape context of a particular point p is a rough description of how the shape "looks" when viewed from p. It is a representation of the polar coordinates (r_i, θ_i) of the vector $p_i - p$, i.e. the distance and angle from p to p_i for all points p_i in the glyph. We store discretized histograms of these polar coordinates as viewed from every point in the glyph, splitting the data uniformly into 5 bins based on $\log r$ and 12 bins based on θ (the values used in [1]).

To compare the shape contexts of two glyphs, we match the spatial points corresponding to times $t_1, t_2, \ldots, t_{100}$ in the smoothed versions of those glyphs. As in [1], we use the χ^2-test statistic as a measure of similarity between the i-th histogram in the two glyphs. The overall similarity of the glyphs is taken to be the sum of the histogram comparisons at each time step.

B. Spatial Comparison

The spacial feature is simply the list of screen space coordinates in a glyph. Spatial comparison has meaning since we normalize the glyphs to all have the same centroid and diameter. In order to compare two glyphs spatially, we sum together the squared distances between points corresponding to the same time value in each glyph. This feature works well to distinguish glyphs with gross spatial differences, such as 't' and 's', but is not as successful at differentiating more spatially similar letters, such as 't' and 'f'.

C. Curvature

The idea of matching glyphs using curvature was taken from Zitnick [2]. The curvature feature is a list of the change in direction at each point in the glyph. More specifically, for some index i, it is the angle between the displacement vectors $p_{i+1} - p_i$ and $p_i - p_{i-1}$. To compare two curvature lists, we add up the squared difference of each pair of curvature values. We also compared the "integrated" curvatures of glyphs, created by accumulating the partial sums of the curvature lists. The noisiness of the data and unpredictable behavior at "cusps" in characters (like the point or turn at the top of the stem of a 'd') made comparisons using curvature more unreliable than the other two features.

V. Minimum Spanning Tree Clustering

We use a linear combination of the features from the previous section to create a symmetric cost function $C(G_i, G_j)$ that takes in two glyphs G_i and G_j and outputs a real number in the range $[0, \infty)$. $C(G_i, G_i) = 0$ for all G_i, and $C(G_i, G_j) < C(G_i, G_k)$ if G_i is more similar to G_j than it is to G_k, as measured by our features.

Given this cost function, we can treat the sentence as a fully connected graph with one glyph at each vertex, where the edge between glyphs G_i and G_j has weight $C(G_i, G_j)$. We then find the vector of edges in the minimum spanning tree of this graph, sorted in ascending order. To find k clusters of glyphs, we simply remove the $k - 1$ most expensive edges from the tree and take the remaining connected components of the graph as our clusters.

We prefer using the minimum spanning tree over k-means for clustering for two reasons. First, averaging feature vectors from multiple glyphs does not always make sense, depending on the features. It certainly

does not always produce a meaningful visual result. In addition to averaging feature vectors, we tried lumping together the space-time points from all glyphs in a cluster and using the weighted average downsampling technique from section III to produce a cluster average. This method produces something that usually looks like the glyphs in the cluster but with its extremal points smoothed out. With either scheme of averaging, the cost score of any glyph in a cluster with the cluster average is disappointingly high.

By comparison, using min-spanning trees for clustering preserves more information about the individual glyphs. If any two glyphs are a strong match (have an extremely low cost), then we probably want to pair them in the same cluster. The min-spanning tree approach supports this intuition, since any edge of the graph with an extremely low weight will appear in the min-spanning tree with high probability.

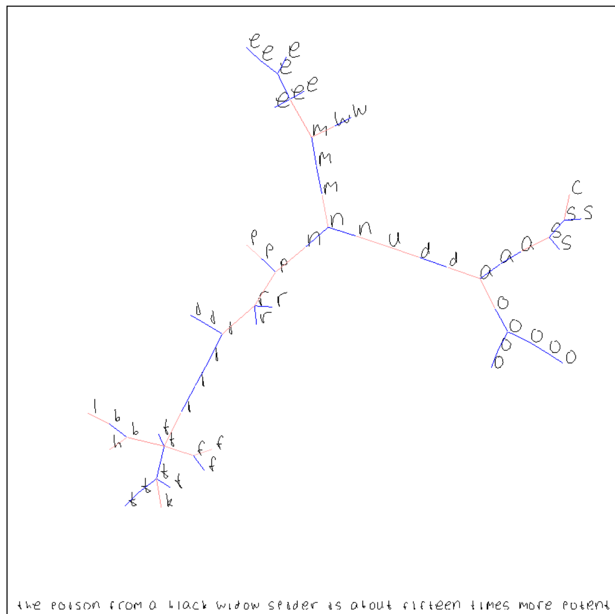

Fig. 1: Visualization of a minimum spanning tree for a sample sentence. The 20 pink edges are the most expensive, so they are cut to create 21 clusters. One of these edges is between two “f” glyphs and another is between two “p”-s so there is slight over-segmentation.

VI. Addressing Clustering Error

Regardless of the clustering method, problems arise if the calculated costs between glyphs are not completely accurate. If glyphs representing different characters have a low cost score, then they may end up in the same cluster (we call this “under-segmenting” the sentence). In this case, solving the resulting ciphertext is guaranteed to give an incorrect answer, because two characters that should be different will remain the same in the solution. It is also possible that the ciphertext will have no solution at all in the language L. We wish to avoid under-segmentation at all costs.

If glyphs representing the same character have a high cost score, then they may end up in different clusters (we call this “over-segmenting” the sentence). Luckily, over-segmentation does not necessarily mean that the resulting ciphertext will be incorrect or unsolvable. To remedy this issue, we relax the assumption that the mapping from clusters back to original characters needs to be a bijection. This means we allow multiple clusters to map back to the same character (for example, the ‘e’ glyphs might end up in two clusters instead of just one).

The calculated costs may be inaccurate either because the cost function is bad or because the user’s handwriting is inconsistent. Consistency is not the same as legibility; it is possible for a user to have illegible but consistent writing. As mentioned in section IV, if a user draws a character in two different ways, our cost function will not cluster those two glyphs together. Allowing for over-segmentation can allow these two glyphs to still map back to the same character, at which point the cipher solver can figure out that they should map back to the same character.

The more we over-segment, the longer it takes to solve the cipher. For this reason, we start with a low number of clusters (15-18), then try to solve the cipher. We increase the number of clusters only if the cipher is unsolvable, since this implies under-segmentation. Typically the ciphertext is only solvable if we have at most 3-4 more clusters than there are unique characters in the original sentence. Beyond that point, too much semantic detail is lost to recover the original sentence in a feasible amount of time.

VII. Solving Ciphertext

We initially based our ciphertext solving method on Hasinoff’s paper [3]. This process involves performing a stochastic search over the space of all permutations of the alphabet. The value of a particular permutation key is the product of the probabilities of all resulting 3-grams of letters in the ciphertext translated with that key.

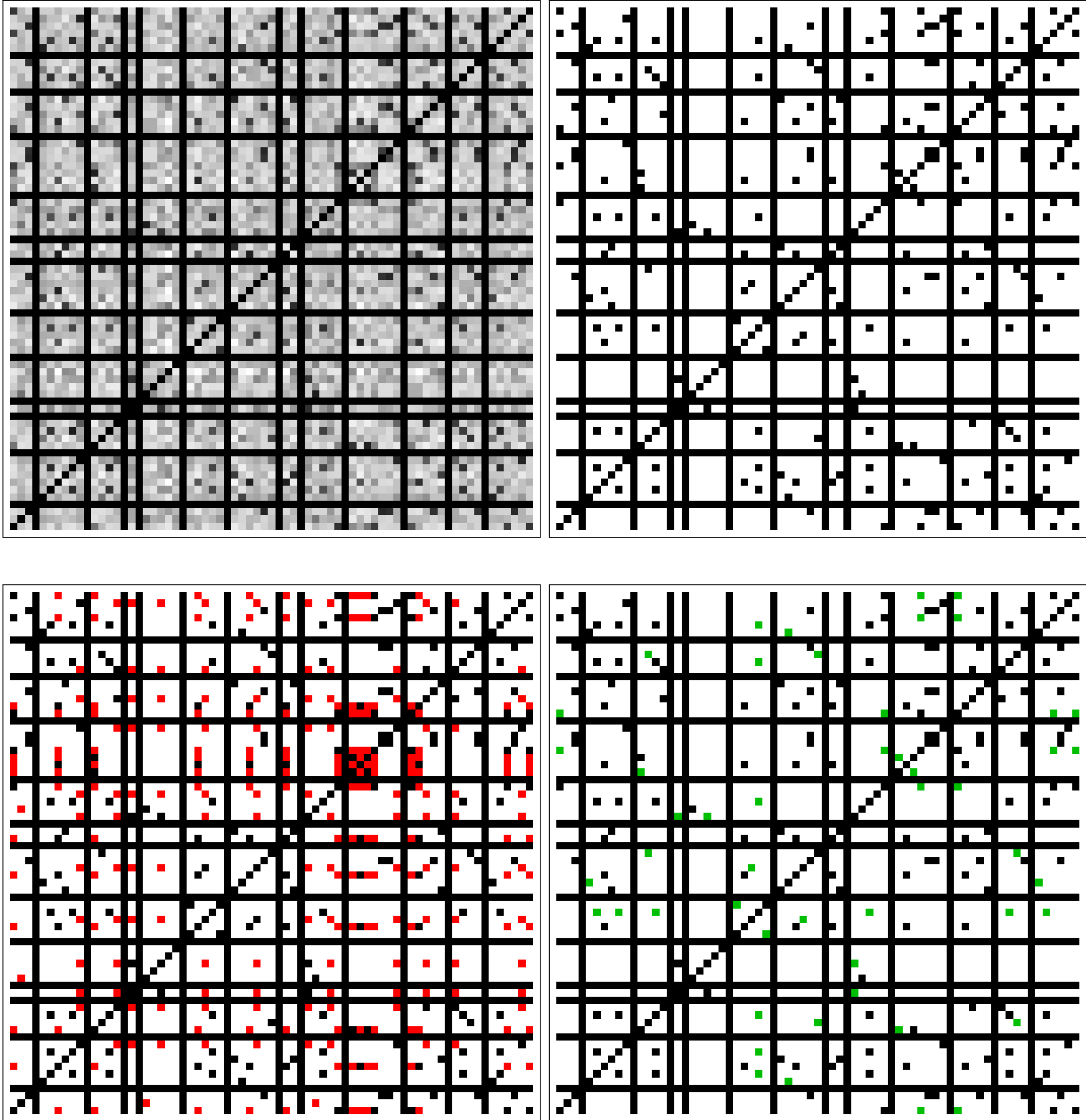

Fig. 2: Symmetric cost matrices for "the poison from a black widow spider is about fifteen times more potent" sample data. Upper left: the raw costs of matching glyphs (i, j) using shape contexts, plotted with (1,1) in the lower left corner. Colors are scaled so that black is zero (spaces matched with any glyph are also black) and white is the maximum cost of matching any two of the glyphs. Upper right: the correct matching function, with black where glyphs are the same and white where they are not. This clustering is achieved by our system using MST clustering with 19 clusters. Lower left: MST clustering using 12 clusters, demonstrating under-segmentation. Red entries where different glyphs are incorrectly clustered together. Lower right: MST clustering using 28 clusters, demonstrating over-segmentation. Green entries are where glyphs representing the same character were not clustered together.

We implemented a simple stochastic search to solve substitution ciphers that worked but was slow, inconsistent and unable to solve over-segmented ciphers. We wanted a procedure guaranteed to find a solution to the cipher if it existed, so instead we used a deterministic brute force search over all possible solutions.

We use a recursive procedure that takes a cipher with n words already fixed and loops over all possibilities for the $(n+1)$-th word. The procedure chooses which word to loop over next based on which unsolved word in the ciphertext has the fewest matches in the vocabulary of language L. We sort the possible matches in descending order by frequency, so the most common words are always tried first. For each word in the list of possible matches, we fix that word in place and modify the lists of possible matches for all other unsolved words to reflect the updated cluster to character mapping, then recursively call the procedure again. If some word in the ciphertext has no possible matches in L, the procedure recognizes this partial solution as a dead end and terminates. If all words in the cipher have been fixed, we score the solution and save it.

To score solutions, we use the sum of the frequencies of single words and pairs of words (1- and 2-word grams) in the solution. Frequency data is extracted from a corpus of sample text written using language L. This helps the cipher solver return the most likely solution, assuming the user wrote a sentence with some semantic meaning rather than a list of unrelated words.

VIII. Results, Issues, and Future Work

We implemented all parts of this system from scratch in C++, using OpenGL to collect the data and visualize the intermediate results. We used the American National Corpus to collect data about the frequency of words and word pairs in English.

The system as a whole is not yet amenable to mass testing. First and foremost, we did not manage to acquire any dataset appropriate to the particular problem we were trying to solve. Also, since the system relies on three components (comparing, clustering, and cipher solving), it will break if any one of them fails. So far, it has succeeded in reading the five data sets we have made (all similar in length to “the poison from a black widow spider...”).

We know that the cost function using shape contexts is very accurate (90-95% testing error on large sets of glyphs). The clustering depends on the cost function, so it is also quite good.

There is the most room for improvement in the cipher solver. The solver should be probabilistic rather than deterministic, and should not need clusters as input. Instead, it should use the cost matrix to probabilistically guess which glyphs should correspond to the same character. Then the cipher solver itself could adjust when it thinks some pair of glyphs has been incorrectly matched, rather than having to wait for over-segmentation to cut the incorrect edge. This would make the cipher solver robust against an imperfect cost function. Paragraphs of handwriting could be solved extremely reliably because of the high amount of semantic information they contain.

Another step to make our system more generally applicable would be to segment entire words into characters. Currently, the user indicates the end of each glyph. This is an artificial constraint, since many people connect some of the letters in their handwriting. The ability to segment written words would make the system much more useful.

It may seem somewhat contrived to perform handwriting recognition while completely ignoring the significance of the absolute shapes of characters. Our goal was to demonstrate the power of semantic analysis alone. One advantage of using semantic analysis only is that it is entirely independent of language. Our method would work for any language whose alphabet has a number of characters similar to English. It would not work for Chinese or Japanese since there would not be enough repetition of characters in a single sentence. More generally, semantic analysis could be added to existing handwriting recognition systems to increase their accuracy when recognizing semantically meaningful passages of writing.

References

[1] S. Belongie, J. Malik, and J. Puzicha, “Shape context: A new descriptor for shape matching and object recognition,” in *Advances in Neural Information Processing Systems*, vol. 13. MIT Press, 2001, Proceedings Paper, pp. 831–837.

[2] C. L. Zitnick, “Handwriting beautification using token means,” *ACM Transactions on Graphics*, vol. 32, no. 4, 2013.

[3] S. Hasinoff, “Solving substitution ciphers,” Department of Computer Science, University of Toronto, Tech. Rep., 2003.