Automatic Evaluation of Handwriting for CJK Characters

Evan Cohen-Doty
University of Rochester
Rochester, NY
ecohendo@u.rochester.edu

Miller Hickman
University of Rochester
Rochester, NY
mhickman@u.rochester.edu

ABSTRACT

Our project was on the evaluation of the quality of a handwritten CJK (Chinese, Japanese, Korean) character. It takes in an image of a handwritten character, then processes the image, recognizes it, and compares it with a typed version of the same character in arial font. We use the typed version as our interpretation of what *perfect* handwriting is, since it is the most easily recognizable and read version for most people. Our methods allowed us to reasonably classify characters as either *well written*, *poorly written*, *very poorly written*, or *okay*. To accomplish this, we had to use a pre-trained convolutional neural network (CNN) in order to recognzie the characters, then compare an image version of the character it output for comparison with the original. Then we were able to report back a numerical assessment of the quality of the handwriting provided based on a number of criteria. After which, we could combine these numerical scores based on whether or not they passed pre-determined thresholds, and then make a judgement on whether the character was written well, okay, poorly, or very poorly.

1 INTRODUCTION

There are two major languages today which make heavy use of the CJK characters, which from here on out we will refer to as "Chinese characters", or sometimes simply as "characters", namely Mandarin Chinese and Japanese. These are both increasingly popular options for language learners, but the use of these characters in either language poses a huge challenge for anyone coming from a language in which they are not used. Learning to write them properly is a struggle and a large challenge in and of itself, however, learning to write them *well* is a whole other task to tackle. Traditionally, you need to either have a great eye, or a patient friend or mentor that is willing to give you feedback as to the quality of your handwriting of the characters. Unfortunately, not everyone is blessed with such an eye or has access to people that are willing to provide such detailed, informative feedback as to how one can improve their handwriting. This is the task which our project seeks to automate; to provide a user with feedback as to how well they're writing a character.

But this raises one big question in particular, namely: "what qualifies as good handwriting?". There are many possible definitions of "good" in this sense. It could mean aesthetically pleasing, in which case calligraphy is the domain to which one should look, as most agree that it is incredibly well written. However, when we consider that we're seeking to help those who are learning a language tied to these characters, particularly Mandarin Chinese, the most easily understood version of the characters for the largest amount of people is simply the typed version in a widely used font. For this reason, we decided to use Arial as our font and compare the handwritten characters to their equivalents in this.

Finally, a small bit of preliminary understanding of this beautifully complex writing system is necessary to understand later parts of the work in their entirety. Chinese characters often have structures with varying degrees of complexity that can be broken down to various different levels. For our purposes, we will need to overlook many of the more linguistic classifications of the different types of characters, and use a much more image-centric view of them. That is, we aren't really concerned with 日 and 上differing in that the former is a pictograph (a picture intended to resemble a real thing), whereas the latter is an ideograph (a picture describing an idea). As images, neither can be broken down into further component parts without going directly into the individual strokes, which is far too detailed to be useful feedback to a user. But there are characters which have a structure that can be meaningfully broken down and used to provide even more useful feedback to a user. For instance, 吗can be broken in half to give us 口on the left and 马on the right. We define characters which fall into the first classification, that is, characters which cannot be broken down, as *base characters*. And we define characters which fall into the latter classification as *recursive characters*.

With this in mind, we can begin examining the methods used to tackle the task at hand.

2 METHODS

2.1 Finding the Character

The first task was to actually find the character in the image submitted by the user. Because there is no general pattern in terms of shape or general form that Chinese characters *all* follow, and there are far too many to make algorithms for each individual character, this is very much a nontrivial problem. For this reason we devised a simple method for finding them. We first need to convert the image to a grayscale so we can deal directly with brightnesses rather than color chanels, which for our purposes, are entirely irrelevant. From here, we can iterate over the pixels of the image to determine the average darkness as well as the darkest value. Once these values are obtained, we can obtain a good threshold by defining it as the following:

$$thershold = darkest\_val - avg\_darkness + \mu,$$

where μ is the standard deviation of the darkness of the image.

2.2 Character Recognition

After finding the character, the next step is determining which character the writer intended to write. This is a very common task and one that many others have solved before us. Thus, we felt that finding a pre-trained Convolutional Neural Network [3] to handle the task would be more than sufficient.

There was, however, some intermediate work necessary since the CNN used has a strict format which it requires from the images passed to it. It requires that they be $96x96$ and that the pixels defining the character hold some nonzero value. This is precisely why the thresholding done earlier allowed the character pixels to retain their original grayscale values. Finally, we simply had to crop out the empty space around the character and scale it if necessary, which is precisely how our program functions.

2.3 Skeletonization

Through our initial testing, we found that the thickness of the handwritten character and the font characters, really the variability of thickness, was having a pretty large impact on the results. Characters that should have scored highly, because they were written quite neatly, seemed to be scoring negatively because the thickness of the character was either much thicker or much thinner than that of the corresponding font character. To accommodate for this, we applied a skeletonization algorithm [1] to the characters, reducing them down to a much thinner version, while retaining the original structure, which we considered to be the *essence* of the characters.

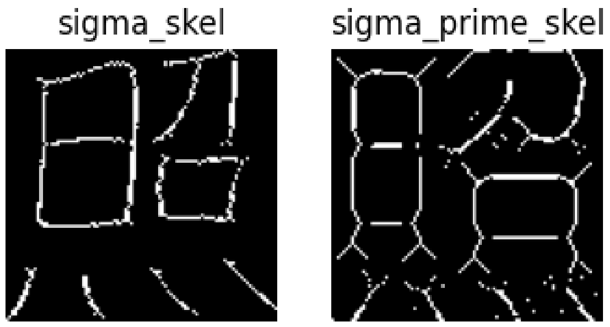

Figure 1: Skeletonization of the Zhao (照) character. Handwritten (left) referred to as sigma and Font (right) referred to as sigma prime

2.4 Radical Segmentation

As described, we aimed to not only judge the character as a whole, but to also break down each character into the radicals that make it up, if there are any. To accomplish this, we employed the use of a CJK Decomposition Database[4]. Given a character, this database returns two main radicals of a character, as well as their orientation. Many radicals are oriented horizontally or vertically, which are denoted in the database with "a" for "across" and "d" for "down", respectively. Some radicals are surrounded by another, where one radical completely encloses the other or partially surrounds it on two sides (like the shape of an "L", but it can be in any orientation). Surrounded radicals are denoted by the database with an "s", as well as additional letters like "br" (for "bottom-right") if it is not fully surrounded.

Once the orientation of the radicals has been determined, we segment them using a different function for each type of orientation, although the process for each function is quite similar. For "down" cases, the function looks at each horizontal line of the image and counts how many active pixels (non-background) are present on each line. It then finds the horizontal (or run of horizontals) with the least number of pixels and segments it there (or at the average of the run). The process for "across" cases is exactly the same, but with a vertical line instead of horizontal. The function for the "surround" cases applies both the down and across functions, then segments off the inner portion where the horizontal and vertical lines intersect.

This process works quite well for radicals that are not touching, however this is not always the case. For radicals that do touch, the algorithm still segments at the vertical or horizontal where the least pixels are present, but the result is not a perfect segmentation. For characters where there are many radicals recursively embedded i the character, the segmentation continues until it finds single components or the decomposition database returns information about individual strokes in the character, which are unnecessary for this project.

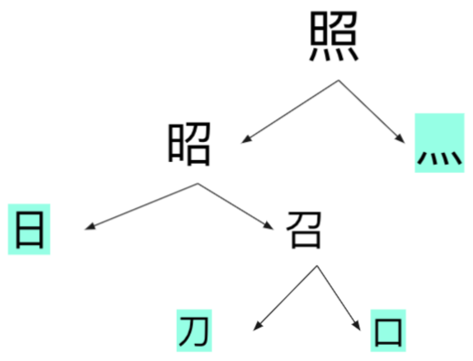

Figure 2: Concept of how Zhao (照) would be broken down into it's radical components.

2.5 Comparison

There were three main metrics used for comparison between the handwritten character submitted by the user and the typed version:

(1) p - The *proportion* of the character, or how much of the frame it occupied.
(2) g - The *location* of the character, or where the pixels were relative to the corresponding image.
(3) s - The *structure* of the character, as given by the formulation of SSIM.

First, we must examine the formulation of p. It's a bit easier to first consider this conceptually. Imagine taking the images, laying them over one another, and comparing how much of the frame they take up. This is essentially how p functions, but it does a little extra work. It breaks up the image into a $3x3$ grid and does a comparison for each cell in the grid first. It them sums up and normalizes the amount taken up relative to the reduced frame. At the end, we compare the overall proportions of the frame each takes up and return how different they are as a percent (for ease of use, this is simply a floating point number between 0 and 1, inclusive). We can then, much like the other two functions, use this percentage as a rating, and with a threshold determine the quality of the written character with respect to this metric. To define p more formally, we have the following. Let π_i and π'_i denote the proportion of the frame taken up by the character in the i^{th} frame of Σ and Σ', respectively, where $1 \leq i \leq 9$. Furthermore, let c_i and c'_i denote the set of character pixels in the i^{th} frame for Σ and Σ', respectively. We then have

$$\pi_i = \frac{|c_i|}{32*32}, \text{ and}$$
$$\pi'_i = \frac{|c'_i|}{32*32}.$$

From here, to condense notation a bit, we define the following:

- $\alpha = \sum_{i=1}^{9} \frac{1}{9}\pi_i$
- $\alpha' = \sum_{i=1}^{9} \frac{1}{9}\pi'_i$.

This allows us to succinctly define p as

$$p = \frac{min(\alpha,\alpha')}{max(\alpha,\alpha')}.$$

Note that taking the minimum as the numerator and the maximum as the denominator allows us to get a percentage difference between the two that is always between 0 and 1 (inclusive).

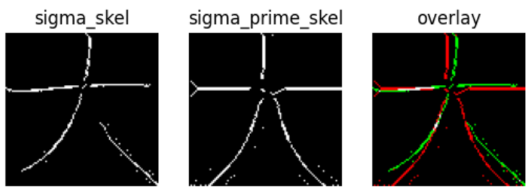

Figure 3: Example of the "Overlay" Intuition Using 大

Second, we must examine the formulation of g. Again, we will use the intuition of overlaying the two images, but now we're comparing how distant the points of the lines are from one another. In the more concrete sense, "points" are simply the pixels which define the lines. We simply add all the character pixels to arrays in row-major order, then for each pair of (more-or-less) corresponding pixels, that is pixels who share the same indices in their respective arrays, we calculate the overall sums of the x and y locations for the respective images, and similarly return the proportion of the two as before.

More formally, we can define g after some basic definitions are established:

- Let C_1 denote the set of pixel locations which define Σ.
- Let C_2 denote the set of pixel locations which define Σ'.
- Let $s_1 = \sum_{(x,y)\in C_1}(x + y)$
- Let $s_2 = \sum_{(x',y')\in C_2}(x' + y')$

We can now easily define g as follows:

$$g = \frac{min(s_1,s_2)}{max(s_1,s_2)}$$

Now, finally we can define s. There is no really intuitive definition of this, and it is not our own, we simply used the formulation of it from the definition of SSIM[2].

3 RESULTS

3.1 Radical Segmentation

As seen in Figure 4, the segmentation worked exactly as displayed in Figure 2. The original character was broken down into its 4 base radicals, which are each contained in their own 96x96 pixel image. Zhao is a pretty straightforward example, as it only consists of radicals in the "down" and "across" orientations. First, component4 is segmented from the original character in the "down" orientation. Component1 is then segmented from components 2 and 3 in the "across" orientation. Finally, components 2 and 3 are segmented from each other in the "down" orientation. This example is a very clean segmentation, as there is ample space between all the characters, making it easy to identify where the segmentation should occur. For characters where the radicals are extremely close or touching, the segmentation algorithm has no chance of a clean output, however it does segment at the point with the least pixel density. Typically, the results would be the same for both the font and handwritten versions of the character, which are being compared, so if the segmentation is equally slightly off for both it shouldn't affect the results of the comparison.

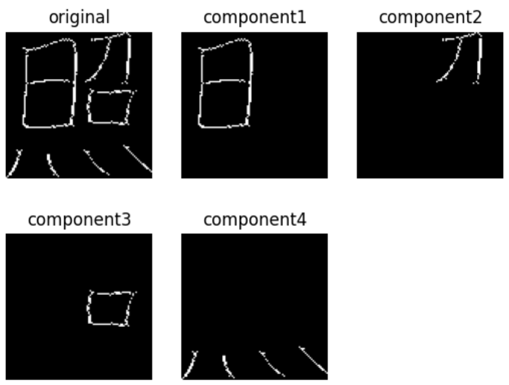

Figure 4: Segmentation of a handwritten Zhao (照) character.

3.2 Comparison

After running our program for about 34 characters, each with four versions representing each quality (bad, okay, good, perfect), we were able to average the values and plot the data seen in Figure 5. All three metrics trend downward as quality decreases, which is excellent. However, the range of values within each metric are somewhat close in proximity, making it difficult to set rating thresholds.

We ultimately decided to set the thresholds at the average of each metric for the "bad" quality. Now, the ratings can be assigned

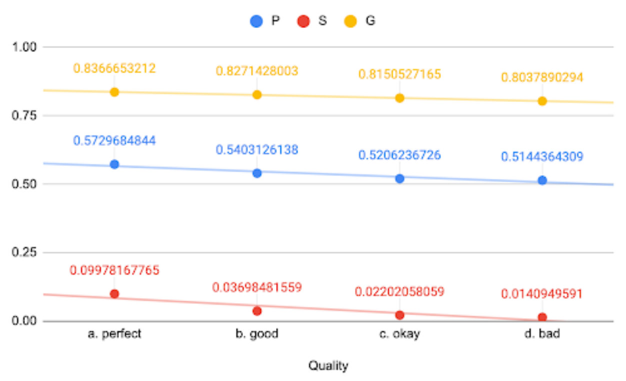

Figure 5: Plot of inital results. Each point represents the average score of each metric (P, S, or G) for the given Quality.

for a given handwritten character. If the character surpasses the threshold for all three metrics, it is rated "good." If it passes only two thresholds, it is rated "okay." A character is rated "bad" if it passes only one metric and "really bad" if it does not pass any of the metrics. The results of the ratings on our test data set can be seen in Table 1. The results from a set of multi-component characters can be seen in Table 2.

	Rating			
Quality	Really Bad	Bad	Okay	Good
Bad	2	18	14	0
Okay	2	15	15	2
Good	1	14	19	0
Perfect	0	7	21	6

Table 1: Results from initial data set. Displays that the program returns appropriate ratings given our subjective quality assignments of test images.

	Rating			
Quality	Really Bad	Bad	Okay	Good
Bad	1	6	8	3
Okay	0	9	12	1
Good	0	10	8	4

Table 2: Results from data set of assorted multi-component characters. There were no "perfect" quality characters tested in this data set.

We also tested the program with a data set written by a complete Chinese novice, who had never written Chinese characters before. Out of 71 inputs, the CNN [3] we used to recognize the characters only recognized 22 of them correctly, about 31%. This is an excellent display of how this CNN is somewhat flawed and can very easily mistake one character for another. There are some negative implications for the program as a whole because of this, as a student with very poor handwriting would be unable to use the program. Although, the individual could also take that as a sign that their handwriting is very poor.

4 DISCUSSION

4.1 Possible Improvement

The results obtained by our program are a great step in the right direction toward giving automated feedback for the quality of a person's handwriting. While they may be far from perfect, the trend that's followed is a good one. And when it does misclassify input, it almost always classifies it as worse than what we would consider it. That is, if it was what we would consider to be an "okay" character, it's far more likely that the program would classify it as "bad" than "good", if it incorrectly classifies it at all. We consider this to be a much better alternative because if you're mislead by the program, it misleads you in the direction of doing more to improve rather than doing less, which is a much better alternative.

But the question remains as to why exactly it has classifies so many as being worse than they actually are? We strongly believe that this is simply a question of not *what* metrics we used for comparison, but rather *how many* we used. There are a number of other means of comparison which could have also been used to take more things into consideration which would have gotten more accurate results, and a few will be presented here with reasons as to why we believe they would help.

Principally, a method by which we could compare the general $form$ of a line would be incredibly useful. This way, even if the lines don't match up perfectly, which becomes overwhelmingly likely for later strokes if early strokes in the character were a bit off to start with, we can still know if they generally follow the same form as the printed version. That's to say, if a line should be straight, how straight is it written? Or if a line should be curved in a particular manner, how closely does it follow that curvature? We imagine that such a procedure could be developed by finding corners in the image, or even just points of significant enough changes in direction, and then grouping the character pixels between such points into lines. Then we could find equations to model the lines in each image and compare them from there. Had we more time, we absolutely would have developed such a method of comparison.

The other big metric which we believe could be a very valuable method of comparison is, similarly, to consider how many corners exist in the image, or (as described above) significant enough changes in direction on a line. We believe that this should be used in conjunction with the aforementioned method above because it helps to verify the results there. Perhaps the lines follow generally a very similar shape, but they aren't making the points clear enough or aren't always connecting the lines as they should (that is, they leave a bit of a gap between the lines at a point where two strokes should be connected). Since the connection of strokes can actually have an impact on the meaning of a character, this can prove to be a critically useful function. For an example of when that becomes critically important, recall that 巳, 已, and 己are all different characters with entirely different meanings.

4.2 Possible Expansions

Moving beyond the ways in which improvement could have been done, we believe it's worth nothing the very realistic expansions that could be done to this program. We believe that there are three major ones. The first extension that could be done with this is almost trivial because our decomposition methods can already handle the most daunting part of the task. That is, to extend this to traditional characters as well. This would allow users who are learning perhaps Cantonese, which still uses the set of traditional characters, or either Mandarin spoken in Taiwan or Hokkien. This would be a rather easy extension, but could likely benefit from additional metric, or perhaps even different metrics, as the characters for which tradition characters differ tend to be much more intricate, and good handwriting might need to be more strict.

The next big extension would be to one particular language, Japanese. Since Japanese mostly uses tradition characters, with certain exceptions like 国and 来, it would be best to just keep the decomposition dictionary the same, as it can handle both sets. The only *real* expansion that would be necessary is to change the CNN to recognize both sets of characters, as well as the Kana. This isn't terribly difficult, but is certainly nontrivial work. Considering the increasing popularity of Japanese as a language, this could be quite the profitable expansion, too.

Finally, the last big extension would be to an arbitrary writing system. With very particular exceptions like the script used for Nahuatl, this could be extended with some work to fit most scripts. Korean Hangul would be a particularly interesting one, as they have an alphabet, however they form their letters into "syllable blocks" of roughly equal size, which gives a very distinct form rather different from most scripts which employ an alphabet. Obviously segmentation would likely have to be removed for most other systems (except perhaps Thai with its interesting use of diacritics, but even then it would need to be meaningfully changed), but all of the comparison metrics could remain exactly as they are, since they're independent of the system used. As they simply compare images, they would work just fine for any other system.

4.3 The Data Used

As our final piece of the discussion, we believe the data we have used here should be discussed briefly. Our data was obtained simply by writing many different characters by hand to varying degrees of quality. We decided to go this route rather than searching out some massive treasure trove of handwritten characters, then spending the time to label each one in terms of its quality, simply because we could have more control over things this way. We could test all different forms of characters (that is, characters following different structure patterns like top-bottom, left-right, etc.) and have a uniform number too. In a perfect world, we could collect a bunch of data from people learning to write the characters and have them all be labeled in terms of their quality (as we have defined quality, of course), and use that data. But, being rather tight on time, we felt that this was the best course of action.

REFERENCES

[1] N. Reddy, "Skeletonization in Python using OpenCV," 10-Nov-2020. [Online]. Available: https://medium.com/analytics-vidhya/skeletonization-in-python-using-opencv-b7fa16867331. [Accessed: 13-May-2021].

[2] P. Datta, "All about Structural Similarity Index (SSIM): Theory + Code in PyTorch," Medium, 04-Mar-2021. [Online]. Available: https://medium.com/srm-mic/all-about-structural-similarity-index-ssim-theory-code-in-pytorch-6551b455541e. [Accessed: 13-May-2021].

[3] Angzhou, "angzhou/anchor," GitHub. [Online]. Available: https://github.com/angzhou/anchor. [Accessed: 13-May-2021].

[4] Amake, "amake/cjk-decomp," GitHub. [Online]. Available: https://github.com/amake/cjk-decomp. [Accessed: 13-May-2021].

[5] chineselexicaldatabase.com. [Online]. Available: http://www.chineselexicaldatabase.com/. [Accessed: 13-May-2021].