ATC

Evan Cohen-Doty

4/8/2021

1 Getting the Individual Characters of a Text

I think the following (unfortunately general) algorithm should do the trick, the only think is sorting out the specifics. We make the assumption that the background is uniform and contains no edges other than the characters.

1: Let E denote the set of edges
2: Create a set C of character representations (set of edges)
3: Let *threshold* denote a value representing the distance two edges must have between them in order to constitute different characters
4: **for** $e \in E$ **do**
5: $append(C, \{e\})$
6: **end for**
7: **for** i such that $0 \leq i < |C|$ **do**
8: **for** j such that $i < j \leq |C|$ **do**
9: **if** $smallest\_dist(C_i, C_j) < threshold$ **then**
10: $merge(C_i, C_j)$
11: **end if**
12: **end for**
13: **end for**
14: Return C

This is a natural analogue to the connected components algorithm, and should we choose the threshold correctly, it should yield the proper decomposition of the text into individual characters. The obvious trouble is choosing that threshold. Just playing with different values should eventually get ones that's "good enough". For truly abysmal handwriting, it will, of course, fail. But that's not really our fault, is it? We might be able to find more sophisticated methods, and I'm sure we're both thinking that a neural net would probably take this problem without a problem, but I neither feel like finding one, nor feel like making one. So, to quote Babe 2: Pig in the City, "that'll do, pig, that'll do."

1.1 Progress and Thoughts

I attempted to extract the edges and go from there, but this was very much nontrivial. Instead, I decided to implement this algorithm slightly differently. Instead of using edges, I created a binary map that separated foreground from background (i.e. writing v.s. paper). This does filter out parts of the characters, but very little (in a way it almost creates noise by adding tiny gaps in the characters which didn't *seem* to exist before. However, this isn't a problem, we can just work around this. Now, this is where it gets a little tricky. Should our distance threshold for separating characters from character components should be relative to the size of the overall image, or relative to the amount of text, or (and I think this is most probably the best idea) relative to how thick lines are?

2 Goals

First, assume that Σ is the character that the user has written. I believe that our goals should be as follows:

1. Get the typed version of Σ and make an image of it that can be compared to Σ, which we will denote as Σ'.
2. Compare Σ and Σ'.
3. Devise some kind of metric for how faithfully a written character followed the *standard*.
4. Create classes for each classification of character (single-character(上,日,水,下), left-right(现,扫,怕), top-bottom(告,笔,累), etc.).
5. Devise a weighting system (in terms of number-of-characters v. how-poorly-they-were-written) to decide which characters to report that the user needs to focus the most on.
6. Devise a feedback system which will take a character written the most poorly for their worst class of characters, and report back to the user where they went wrong with it.

3 1. Getting Σ' and Making an Image

Obtaining Σ' from Σ is (for use, at least), rather trivial. We pass Σ to the CNN, and it'll return Σ'. Now, to get it as an image which we can use for comparison is a bit trickier. I foresee a couple ways in which we could tackle this:

1. Start with a small font, print the character using it, and use the image resulting from it. If a bigger one results in the same sized bounding box (or at least one that's closer in size to the one surrounding the written

character from the user), use that one. Keep going bigger while better, bigger boxes exist.

2. Print one standard size and use an image version of that, but potentially blowing it up or shrinking the other

4 2: Comparing Σ and Σ'

When it comes to actually comparing them, we have a couple options:

1. "Overlay" them and compare them exactly. You could though it is low level, just count the number of pixels locations contained in $(\Sigma' - \Sigma) \cup (\Sigma - \Sigma')$, call it δ and then assign your "quality" rating based on the value of δ.

2. You could do the above method, however, you would make the tweak of trying to segment the character into regions and assign it some score relative to how spread out the values of δ are (where more spread is better because it implies that the misses are insignificant, but less spread is worse because it implies that there's a big portion of the image which is off.)

3.

Note that option 2 might benefit from pre-segmenting the character into more logical regions (ideally by the radicals or strokes (but this is really hard for strokes, similarly probably very hard for radicals)), so that way the regions carry more significance inherently. Consider segmenting based on the character classification, call it κ.

5 3: Rating Σ

After we've compared Σ with Σ' and calculated some value δ to quantify how accurate it was. Now it should follow rather naturally given the method we use for calculating δ and this section becomes kinda pointless (hindsight).

6 4: Character Classifications

This we can steal directly from one of the databases we're using, which when queried with a specific character, returns a string we just have to interpret for the pattern. So essentially this is a trivial parsing task. We can then just create some dictionary which hashes on the character class (recall, this is like left-right, single-character, etc.), and then for values, we can just hash directly to the list of individual Σ's that fit the class, along with their δ values (so the list would be of tuples of the form (Σ, δ)).

7 5: Weighting System

This is where things get tricky because we need to determine what's more important, how many were missed or how egregiously they were missed. I propose the following assumptions. First, we assume that $0 \leq \delta \leq 1$, where $\delta = 0$ denotes a quite literally $flawlessly$ written character, whereas $\delta = 1$ denotes an abysmal character that is recognizable, but pitiful. We will also assume that the cases of $\delta = 0$ and $\delta = 1$ are handled as edge cases, because we don't *really* need to do anything with said characters. Either the user entered it perfectly, so no feedback is possible beyond "good job, keep doing that", or it was so bad that they should perhaps focus and put more effort in.

With these assumptions, we can devise a weighting scheme, but must take the following problems into consideration.

1. There may be a disproportionate amount more of one character pattern than the others

2. It may be the case that most characters are written really well (say $\delta = 0.05$), but we have two distinct groups of poorly written characters. Call the first group Γ_1 and the second group Γ_2. Suppose that Γ_1 contains a number of characters which, relative to Γ_2 is a lot (but relative to *most* characters written, it few), and they're done semi-poorly (let's suppose $\delta = 0.5$). However, Γ_2 contains very few characters, but all of them are written terribly (let's say $\delta = 0.8$. If we prioritize sets like Γ_2, the user will be able to focus on the more egregiously poorly written characters, but maybe miss out

8 A Semi-Formal Description of Evaluation

The first critical step in our procedure will be gathering the *critical* information we need in order to evaluate. But what is actually critical? I propose the following.
Suppose we're given a handwritten character as an image, we will denote this as Σ. Now suppose we've run Σ through our CNN and we've selected the most likely of the three characters the CNN will return (and in the case of ties, suppose for now that we choose the *correct* one (that is, whichever one the user intended to write)), we will denote this in an equivalent image format as Σ'. Now, here's the key part, we take Σ''s character value and query our database for the character pattern, we will refer to this as $\Sigma'.\pi$. So far, the work we've done is more or less trivial, we're just utilizing the tools we've found. We have one more step which is more-or-less trivial. We need to create a collection of "bounding boxes" which can almost be thought of as filters, for each type of π. That is, we'll have one which allows us to select just the bottom component of a top-bottom type, or just the right of a left-right type. So, in essence we now have a vector for each component of a given value of π, and we will denote the i^th component of π as π_i. If we think a bit more in terms of implementation,

this means that π_i is actually just a set of valid pixel locations in which the component can lie.
So, before moving on to more detailed and interesting analysis, let's lay out everything before ourselves so we have a nice, concise and easy reference for later:

1. Σ - The image of the original character
2. Σ' - The image of the printed character
3. $\Sigma'.\pi$ - The character type for Σ'
4. $\Sigma'.\pi_i$ - The i^th component (or radical) of $\Sigma'.\pi$

Now, we get into the interesting stuff. We must define some way of meaningfully quantifying the accuracy of a character. For now, we have to make one assumption which we can deal with more concretely later. Suppose we define some value Δ to denote the accuracy of the user's Σ relative to Σ'. We define Δ more formally as follows: $\Delta = \sum_i \delta_i$, where δ_i denotes the accuracy of the i^th component (or radical) of our current $\Sigma.\pi$. Our assumption is simply that we know how to compute a *good* value of δ, and we will address this problem later. Now, as it stands, we have to consider that as it's currently defined, Δ is going to be an *overall* judgement of the quality of the character, and thus it will treat two important cases as being equivalent. The two cases are as follows:

1. $\Delta = \sum_i \delta_i = (\delta_1) = x$ (and keep in mind we're guaranteed that there will always be at least one component because even in our most basic characters, we have at least one radical (such as 日,大, and 小))
2. $\Delta = \sum_i \delta_i = (\delta_1 + \cdots + \delta_k) = x$, where $k > 1$.

But, critically we must note that there are two subcases of case 2, namely:

1. $(\exists_j)(\forall_{k \neq j})[(\delta_j >_n 0) \wedge (\delta_k \simeq 0)]$
2. $(\forall_j)[\delta_j >_n 0]$

Note that we use the symbols $>_n$ and $\simeq$ to means "non-trivially greater than" and "similar or equal to", respectively. This essentially just means that we'll be treating cases in which a strong error in one component with little to no errors in other components, and smaller non-trivial errors in multiple components result in the same overall error, as being equivalent. Now, this isn't inherently terrible, but we have to consider how we want to advise the user.
Fear not, I also hate such open ended things like what I just stated above when we have a task at hand which requires very concrete solutions! I propose the following way to interpret not just our Δ values, but our even each of our δ_i values as well. First, we must "zoom out", and think more big picture. Suppose we have a vector of all the characters in a text, denote this vector as $\mathbf{\Sigma}$. Denote the i^th character of the text (so of $\mathbf{\Sigma}$) as Σ_i, it's printed image as Σ'_i, and the value denoting their similarity as Δ_i. We denote the text itself as T. If we

create vectors $\mathbf{\Sigma}'$ and $\mathbf{\Delta}$ whose elements correspond directly with $\mathbf{\Sigma}$, we can thus represent T as the tuple $< \mathbf{\Sigma}, \mathbf{\Sigma}', \mathbf{\Delta} >$, which is actually just a matrix with shape $(i, 3)$, where $|\mathbf{\Sigma}| = |\mathbf{\Sigma}'| = |\mathbf{\Delta}| = i$! So, now for the solution. We start by placing each character into a "bucket" based on its character type, $\Sigma.\pi$. We then order these buckets based on a normalized average of their collective Δ values. If we let B denote an arbitrary bucket, then this collective normalized average, denoted α is $\alpha = \frac{C \sum_i \Delta_i}{|B|}$, where C is our normalization constant.
Now, we have the buckets sorted in terms of non-increasing order on α. So, we can already report the most critical piece of information to the user: what type of characters they tend to write the most poorly. But, of course we want to give more! We want to show them which ones they missed the worst. We can easily figure this out if we simply recall our δ values which we calculated earlier! We can give them the additional feedback by telling them which components (in this sense "component" in the more geographical sense of the term, rather than semantic) of the given patterns they messed up the worst. We do this simply by examining each component's δ score and seeing which one indicates the worst component accuracy and report that with perhaps a comparison with Σ' which highlights the error.

8.1 The Remaining Question: How Do We Compute δ?

In short:

¯_(ツ)_/¯

Okay, kidding aside, I have an intuition, but I would grain of salt this. You probably have a better idea, I am not a clever man. But, I feel like I haven't done much as of yet and I owe at least tossing out the idea.

I propose that we try to think of overlaying the images for comparison as creating a "mapping to equivalent parts". That is, we map from where the pixel *is* in Σ to what's in the same places in Σ'. We then give a base score to this. Next, we see if one or many of a given finite number of transformations would result in a net improvement of the score . If that's the case, we adjust the score appropriately because it falls in the class of "*easily* improvable examples" (which we can report as such to the user and give our "easy" solution for improvement). If that's not the case, however, then it's too poor for an easy fix (assuming we filter out ones that are too good in advance to avoid unnecessary computation). In this case, because we're most interested in these ones, we can just note whether their inherent awfulness is the result of one or many components, and use that information later.