Breaking CAPTCHAs with Statistics and Wavelet Transforms

CAPTCHA is an acronym for “Completely Automated Public Turing test to tell Computers and Humans Apart.” It is a type of challenge-response test in which distorted letters and numbers must be accurately identified. These tests are set up so that most people can pass, but current computer programs cannot, and so they are helpful in determining whether a user on a website is human. Many websites have them in place because they do not want their system to be broken into by a bot. For example, a website that sells concert tickets would not want a program to automatically purchase all of them, instead of letting individuals purchase them on their own. Many blogs and forums also require users to identify a CAPTCHA correctly in order to reduce comment spam but still allow legitimate comments. Similarly, free email services use CAPTCHAs to prevent bots from signing up for thousands of email accounts and sending spam.

Figure 1. Examples of CAPTCHAs
When CAPTCHA codes first appeared, many were very simple. In many cases, they were either sets of words or consistently the same length so that it was not impossible for a computer program to deconstruct the code into characters and identify them. Students and faculty at Grand Valley State University in Michigan already figured out how to break the CAPTCHA codes used by the Holiday Inn Priority Club in 2005. They developed a method for creating an algorithm that isolates the letters and then recognizes them. However, modern CAPTCHAs are generally much more complex. They may have characters touching, lines drawn through the code, and many different ways of distorting the characters or images. This makes segmentation and identification very difficult.

Our goal is to use wavelets, geometry, and statistics to break CAPTCHAs that are slightly more sophisticated than the ones used by the Holiday Inn. Our intent is not to be malicious in doing so, but rather to help companies and websites test the rigor of the CAPTCHA system they might implement. We will use randomly chosen uppercase letters, lowercase letters, and numbers to create our CAPTCHAs in Mathematica. The characters will be assigned random colors, and the CAPTCHA will be made up of three to five of the characters to create a “word”. This word will then be either rotated by a random angle, or each character within the CAPTCHA will be randomly rotated at different angles.

Figure 2. Examples of CAPTCHAs for this project
A directory is created that contains each colored number, uppercase letter, and lowercase letter. These are then stored in Mathematica as colored images, and as matrices in greyscale. The directory is used for generating CAPTCHAs, and is used as a resource for the program to compare characters of the CAPTCHA to when making a guess. The computer program is set up to follow a basic algorithm. First, if the entire word has been rotated, the code must estimate this angle and rotate the word back to horizontal. Next, it will split the word up into individual characters. If these characters have been individually rotated, it will then estimate those angles, and undo those rotations. When the characters have been split up, there are still blank rows and columns around the actual letter or number. Next these edges will be trimmed off. The edges will also be trimmed off of the characters in the directory, and the list of possible matches will be pared down using the dimensions of each character. Lastly, the final conclusion will be made using wavelet transforms. A wavelet transform will be performed on each character of the CAPTCHA, and each of the final matches from the directory. Then each final match transform will be subtracted from the CAPTCHA character’s transform, and the result that has the lowest norm will be the best guess.

If the entire word has been rotated together by a random angle, linear regression can be used to find the angle of rotation. As shown in Figure 3, the CAPTCHA can be graphed using Mathematica, along with a best fit line \(y = mx + b\). The angle of rotation can then be calculated simply by taking the arctangent of the slope, \(m\). Next, the CAPTCHA is rotated by the negative of this angle in order to bring it back to a more horizontal line. While the resulting, altered CAPTCHA is now much less rotated, it may still not quite seem horizontal. Thus, multiple iterations of this can be performed. The altered, rotated-back CAPTCHA can be graphed with a best fit line and un-rotated again, and then that result can have the same steps performed on it as
well. The program is written to repeat the process until the resulting word is horizontal within a
certain tolerance level (we chose 0.05 radians).

Once the word has been brought back to horizontal, the characters within it can be
identified and separated. Even though each character is colored, they are easily converted to
greyscale. On a computer, images are stores as matrices with each pixel represented by a
number. White is 255 and black is 0, with shades of grey being values in between. Thus, the
matrix for the CAPTCHA has several columns and rows containing only 255 in between and
around each character. The program for separating the characters first transposes the un-rotated
CAPTCHA matrix and then finds groups of rows that are all white, and groups of rows that have
nonwhite pixels. Then the rows with nonwhite pixels can be singled out, and transposed back to
normal again. Thus, the white on the sides of each character has been stripped away, and the
single CAPTCHA is now broken up into the three to five characters it was made up of. There are
still rows of white above and below each character. In order to trim these off, the program finds
the rows that contain only the value 255 above and below the character, and deletes them from
the matrix.
As shown in Figure 4, once each character has been trimmed, it has somewhat defining dimensions. While S, P, and B all have somewhat similar rectangular shapes, the letter “I” has a very long, skinny shape, and X would have more square dimensions. Thus, using the lengths of columns and rows for each character, its ratio of dimensions can be calculated and used for comparison with the directory after those characters have also been trimmed. Since the next step, taking the wavelet transforms, takes a bit more time, it makes sense to only compare each CAPTCHA character with a pared down list of matches instead of the entire directory. This reduced list of matches is found by taking the top characters that have the most similar ratios of dimensions. After performing quality tests and using trial and error, I found that taking the top 14 matches from a directory of 62 possibilities was not only sufficient, but over 90% accurate.

When the characters of the CAPTCHA have been individually rotated, a slightly different process is used for paring down the list of possible matches and undoing the rotation. Using a built in function in Mathematica, the list of matches from the directory can be reduced while the characters are still rotated. ImageCorrespondingPoints is able to analyze two images, and find matching points within them. Thus, a simple function can use ImageCorrespondingPoints to compare a CAPTCHA character to each character in the directory and then return a list of the directory characters that have the most matches with the CAPTCHA character. I found that taking top dozen with the most matches from the directory was generally sufficient. This method
may have worked for finding top matches previously, instead of using the ratios of dimensions. However the previous method is quite accurate for the straightened out characters. This way of paring down matches was not necessary until we looked into individually rotated characters.

Since each character is rotated differently, and a best fit line may not help with only a single character, linear regression is no longer an easy way to calculate the rotation angle. However, there is another function built into Mathematica that is useful here. ImageCorners finds important corners in an image. Our function uses ImageCorners to find a certain number of corners (I use 6) in the CAPTCHA character, and in each of the top matches from the directory.

![ImageCorners on rotated A and directory A](image)

**Figure 5. ImageCorners on rotated A and directory A**

Then, the function centers both images. It finds where the important corners within the image are in relation to the origin, for both the CAPTCHA character, and each of the directory characters. Then it calculates how much the character from the CAPTCHA has been rotated, if the character from the directory is the original. If A has been rotated, for example, by $\pi/2$ (Figure 5), then when comparing this character to A in the directory, the resulting angle of rotation will be about $\pi/2$. If the rotated A is compared to B in the directory, the resulting angle will probably not be correct. The different corners are just too dissimilar. Thus, when the CAPTCHA character is then un-rotated by all of these different angles, only rarely will it appear to be right-side up and
horizontal. This will make it fairly easy to compare to the directory when using wavelet transforms and get the correct result.

Figure 6. Wavelet transform of “A” at 1 and 2 iterations

Once all characters have been separated and un-rotated and there remain 12-14 possible matches from the directory, wavelet transforms are used to make the final guess. A wavelet transform takes an original image matrix and creates a new image matrix, as illustrated in Figure 6. The top right corner is a compressed version of the original. It is created by taking the averages of every group of four values from the original matrix. For example, if the top left corner of a matrix has the values \[
\begin{bmatrix}
110 \\ 100 \\
90 \\
80
\end{bmatrix}
\]
the upper left corner of the wavelet transform will be the average of these values, 95. The upper right corner shows the vertical differences within the image matrix. Using the values from the previous example, the upper left corner of this section will be calculated by taking the average of the left column minus the average of the right column, which is 10. The bottom left corner illustrates horizontal differences, and is calculated by subtracting the average of the bottom row from the average of the top row, which is also 10 in this example. Finally, the bottom right corner reveals the diagonal differences of the matrix. This is calculated by taking the average of the main diagonal, and subtracting the average of the opposite diagonal. Thus the upper left corner of the bottom right section would have the value 0,
using the examples numbers. All values of the wavelet transform are calculated using each group of four values from the original matrix like so. As shown in Figure 6, multiple iterations can be done. I found that the best results came from using two iterations. More iterations than that did not significantly improve the accuracy of the program.

Once the wavelet transforms have been done for both the CAPTCHA character and each of its top matches, they all will be run through a function that subtracts each final match transform from the CAPTCHA character’s transform. Next it takes the norms, and returns the result from the directory that has the lowest norm. This is repeated for all characters in the CAPTCHA in order to find the best guess for each. This program, after all steps have been completed, is about 90% accurate.

Figure 7. Success!
The most exciting part of this project is that new, more complicated CAPTCHAs are constantly being created. There will always be more challenges added to this problem. In the future it would be interesting to add noise to the CAPTCHAs, and use denoising methods to solve them. Characters could also follow a polynomial instead of a straight line so that they are bending. It would also be very important and relevant to find ways to solve CAPTCHAs that have characters touching or even overlapping slightly, or have lines through them. This would create problems in separating characters, but would be incredibly useful for solving many of the CAPTCHAs used today.
Bibliography:


https://en.wikipedia.org/wiki/CAPTCHA

http://www.extremetech.com/computing/103283-breaking-and-fixing-captcha