Decoding reCAPTCHA

COSC 174 Term project

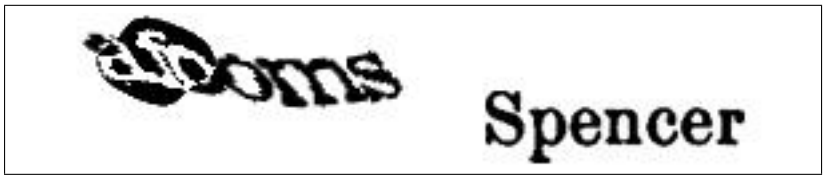
Curtis R. Jones & Jacob Russell

May 28, 2012

Project Objective

Our project is on solving reCAPTCHA puzzles. A reCAPTCHA is an image containing two words, one for which the computer knows the solution, and one which normal OCR algorithms gave a low probability of being correct. The idea behind CAPTCHAs is that they solve two problems at once, discriminating between humans and bots, and solving the most difficult word recognition problems.

So, we would like to take in an image and convert it to text:



→ **afooms Spencer**

However, the guidelines for solving reCAPTCHA puzzles allow 1 character to be incorrect, so we have a little flexibility in this regard.

State of the art

According to Strong CAPTCHA Guidelines[1] CAPTCHAs are considered broken if an automated system gets a 1% error rate. This is because spammers can still create enough accounts to be successful since they are not using their resources, rather use botnets. This leads to a high turnover rate on CAPTCHA puzzles, making it hard to define state of the art for CAPTCHA solvers.

Wilkins also writes that using the Tesseract OCR implementation, originally written by Hewlitt Packard and maintained by Google, he got a success rate of 5%. If he considered that in a CAPTCHA only one word has to be correct, and assumed that 50% of the time when he only got a single word correct it was the one he needed, he got a success rate of 17.5%.

Beede[2] found that a tool called AntiReCAPTCHA gets 2% on CAPTCHA puzzles and gets 21.5% on individual words. More importantly, on his website, Beede provides the actual CAPTCHA dataset that he used, so useful comparisons can be made with his work.

Baecher et al.[3] are the most current paper that we could find (2011) and they get a success rate of 5% on ReCAPTCHA puzzles using a holistic classifier and doing a lot of preprocessing to remove ellipses.

It is important to note that the high turnover rate on CAPTCHAs makes a single year radically different in terms of success rates (and CAPTCHAs themselves).

Bursztein[4] states that many different classifiers perform strikingly well once the CAPTCHAs have been segmented. He recommended using KNN or SVM classifiers because of their relative simplicity and not much difference in performance among classifiers.

To summarize, a number of techniques perform approximately 5% on CAPTCHAs, a holistic classifier, the AntiReCAPTCHA (which just tries do un-distort the words and then runs an OCR package on the new image) and KNN and SVM are all state of the art for CAPTCHA solving.

Dataset

Google to Old reCAPTCHA

Getting CAPTCHA data is quite easy, there are scripts available on the net to download CAPTCHA images, and one can hand label them. This is what we initially did. There were significant challenges in segmentation and ellipse removal.

Unfortunately because CAPTCHAs have a high turnover rate, getting CAPTCHAs doesn't necessarily mean that our results will be comparable to other people's results.

In order to compare with other people's results, we have to use the same dataset or at least the same type of CAPTCHA. The only dataset that we were able to find is the one from Beede[2], which contained 1000 images for testing and 1000 images for training.

myCAPTCHA

There were still substantial challenges affecting the HMM development after changing over to the re-CAPTCHA dataset. Therefore, we set out to create a second dataset that would allows us to remove effects such as noise, word-shaping, and low kerning. Most importantly, it would allow for clean segmentation of each character sequence. We called this dataset myCAPTCHA.

The dataset consists of 1200 images, each consisting of a single character sequence. Each sequence is a random arrangement of letters from a small alphabet consisting of 'a' to 'z', 'A' to 'Z', and digits '0' to '9'. Sequence lengths ranged from 3 to 15.

All sequences were generated using MATLAB and the rand command. All image files were tagged with their correct label.

Preprocessing

Ellipse detection and removal

Modern reCAPTCHAs now include shapes, which are best described as "ellipses", which are arbitrarily placed in the reCAPTCHA. Where ever this ellipse occurs the pixels are inverted. This compounds the problem of edge detection since a letter and an ellipse may share an edge and the letter's edge profile may change at some point in the letter. Ultimately, any ellipse should be removed to try and improve the overall image so that proper observations can be taken.

Our application uses a very simple density search to determine if an ellipse is present, and then basic neighborhood search to remove it.

The density search is done using a sliding window which takes a snapshot of the pixel density at that location. It then compares that density to that of the entire image.

Figure 1: Depiction of the sliding window used to scan the density of the image.

If the ratio of comparison is over 1.5 then we assume that an ellipse is present at that location and we proceed to try and remove it. Below is an example of an ellipse removal.

Figure 2: An example of a CAPTCHA before and after ellipse removal.

This algorithm has approximately 75% accuracy using our training images.

Segmentation

Segmentation is the process of splitting up the words into letters. This is one of the most important parts of word recognition, and also where we ran into real problems.

The literature recommended Color Filling Segmentation[5] to segment the images. This approach, which was previously used with CAPTCHAs, did not work for us to segment our reCAPTCHAs because the letters are too close together (and CAPTCHAs have been updated, so they're not the same type of images).

MATLAB has a built in function to segment letters based on connected components. This segmentation approach didn't work for us either because of the same reason.

What we actually ended up doing is choosing fixed window sizes and using the fixed window sizes for the moment because it's not the focus of the project.

Figure 3: An example of a CAPTCHA segmented using uniform window sizes.

Unfortunately, this leads to not so great training examples, which in turn leads to bad classification.

Descriptors

Descriptors are needed to describe what our image looks like to the HMM. The HMM itself is supposed to learn the patterns in the sequence of the descriptor, and then associate that pattern with a classification (or classification sequence). Pixel Sums

Pixel Sum

We sum the black pixels in the column for each letter and use the column sum vector itself to describe our image.

Note: important to this descriptor is that we can't have much noise in the image. This makes preprocessing (especially ellipse detection and removal) particularly important.

The idea behind this descriptor is that (since we are using a sequence classifier) we are capturing a simple relationship between sequences of columns. This relationship is then assumed to be representative of the image itself.

This descriptor is weak but it enabled us to begin HMM implementation with some form of data without having to sort through implementation issues possibly caused by more advanced descriptors. Histogram of Oriented Gradients (HOG)

HOG

Once our training implementation was complete we moved toward a more advanced image descriptor. We decided to use an implementation of Histogram of Oriented Gradients (HOG) [6] since it relies on grayscale images and a large majority of our reCAPTCHA images are also grayscale.

Furthermore, regardless of image size the HOG algorithm would return an observation vector that was 1-by-81 which would work as a form of dimensional reduction. It would also let us use dynamic window sizes over the images when making observations instead of limiting ourselves to a fixed observation window width.

Algorithms

Unsupervised

The classic algorithm for Hidden Markov Model training is the Forward-Backward/Baum-Welch algorithm. It's an expectation maximization algorithm that maximizes the likelihood of the states given the data. Usually it takes as input a vector of observations (O), a transition matrix (A), and a likelihood matrix (B). We still have all of these, but we have a vector of vectors (or a matrix) for our observations. We also let S represent our N states and let V_i represent our output (vocabulary) corresponding to state S_i.

Algorithm 1 Forward-Backward

Initialize A uniformly $\ni \sum_{j=1}^{N} A(i,j) = 1 \forall i \in [1, N]$

Initialize B and π randomly $\ni \sum_{j=1}^{N} B(i,j) = 1 \forall i \in [1, N], \sum_{i=1}^{N} \pi_i = 1$

converged $\leftarrow$ false

while converged = false **do**

$\xi_t(i,j) \leftarrow$ Expected # of transitions from $S_i \rightarrow S_j$ at $t \in [1, |O|]$ ▷ E-Step

$\gamma_t(i) \leftarrow$ Expected # of times in S_j at $t \in [1, |O|]$ ▷ E-Step

$$\bar{A}(i,j) = \frac{\sum_{t=1}^{|O|-1} \xi_t(i,j)}{\sum_{t=1}^{|O|-1} \gamma_t(i)} \forall i, j$$ ▷ M-Step

$$\bar{B}_t(j) = \frac{\sum_{t=1}^{|O|} 1\{O_t = V_k\} \gamma_t(j)}{\sum_{t=1}^{|O|} \gamma_t(j)} \forall j$$ ▷ M-Step

$$\bar{\pi}_i = \frac{\gamma_1(i)}{\sum_{t=1}^{|O|} \gamma_1(i)} \forall i$$ ▷ M-Step

if $\|\bar{B} - B\| \leq$ tol **then**

converged$\leftarrow$ true

end if

$A \leftarrow \bar{A}$

$B \leftarrow \bar{B}$

$\pi \leftarrow \bar{\pi}$

end while

After running this algorithm, we have a transition matrix that describes the probability of transitioning to the next state given the current state, and a likelihood matrix that has the probability of ending in a state given the current observation, but our states don't necessarily correspond with the values that we would like. Therefore, we must *decode* them with the Viterbi algorithm. We won't describe decoding in this section because it's the same as the supervised algorithm.

For our specific implementation, the observations are a Histogram of Oriented Gradients vector split into parts with K-means run over each part. This allows us to reduce dimensionality from the width*height of the image to an arbitrary choice of dimensions. We do this over each image of a letter.

Supervised

Since we have the labels for our training set we can leverage them to help train the HMM. We did this using simple counting to determine the probabilities in the prior, transition, and emission matrices based on the observations provided by the HOG algorithm. Unfortunately, this is not an HMM actually being trained, but it will be an HMM to the observer during testing.

In this case we started by segmenting each image based on the number of letters in the label. This results in a very rough segmentation of the word. Each segmentation is fed into the HOG algorithm to retrieve a description of that segment. Each HOG description is mapped to a cluster index using K-means, and that cluster index is used when incrementing the emissions matrix.

Training begins by reading each CAPTCHA label and calculating the transition matrix (A) using

$$P(s_j|s_i) = a_{ij} = \frac{C(i_t, j_{t+1})}{\sum_{j=1}^{N} a_{ij}}$$

Where a_{ij} represents the transition probability from state s_i to state s_j. The function $C(i_t, j_{t+1})$ will count the number of times state s_i is followed by state s_j in the label. The probability is then normalized using the sum over all outgoing transitions in that state s_i where N is the number of states. As a result, every row in the transition matrix must adhere to the following constraint

$$\sum_{j=1}^{N} a_{ij} = 1 \quad \forall i$$

As the transitions are being tallied, the prior probability vector is also built using the following

$$P(s_i|t=0) = \pi_i = \frac{C(i_{t_0})}{\sum_{i=1}^{N} \pi_i}$$

Where π_i represents the probability of seeing state s_i first in the label. Like the transitions, the prior probability vector must adhere to the following constraint

$$\sum_{i=1}^{N} \pi_i = 1 \quad \forall i$$

Finally, the emission matrix (B) is built using a combination of the current state and the observation vector created by using the HOG algorithm. Each image is segmented where each segment represents a single letter. That segment is analyzed using HOG and a 1-by-81 observation vector is returned which describes that segment's gradients.

Now, the observation vector has a dimensionality of 81 while the emission matrix requires a integer value to reference the appropriate column. In order to map the observation vectors to an integer, K-means is used to

cluster all observation vectors into k clusters. Using the centroids provided by K-means, each observation vector is classified to an integer value allowing us to use the following to calculate B

$$P(s_i|O_k) = b_{ik} = \frac{C(s_i, O_k)}{\sum_{k=1}^{K} b_{ik}}$$

Where O_k is the k-th observation cluster. Like the transition and prior probabilities, the emission probabilities must adhere to the following constraint

$$\sum_{k=1}^{K} b_{ik} = 1 \quad \forall i$$

Modified Supervised

Like the traditional supervised method this algorithm segments the images according to the number of letters present in the label. However, the observations are based again on the colum-sum of pixel values used in unsupervised training.

The modification in this method is how the transition matrix is initialized. The emission matrix provides the best possible state the HMM will be in based on an observation. Therefore, that state is used to determine what the next state will be that the HMM transitions to. This is counted in the transition matrix and the probabilities are found based on the observations used to build the emission matrix.

Viterbi

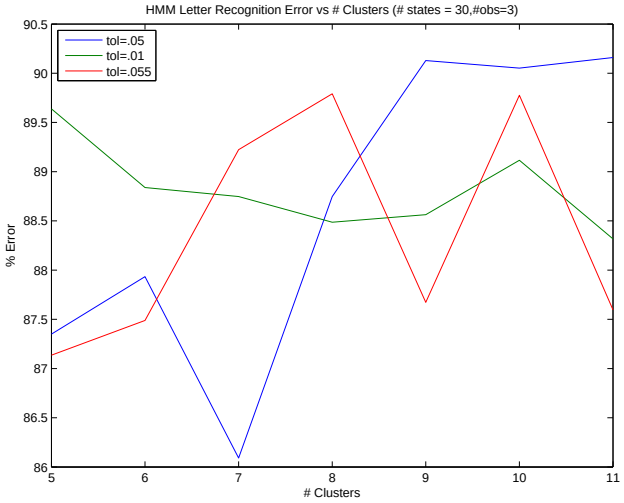
After training is complete the testing images are evaluated against the HMM using a Viterbi algorithm. The algorithm works by choosing the most probable path through the HMM at each observation point. The result is a probability that the input observation vector corresponds to a given path through the HMM. This path result is used to determine the letter, or letters, that the observation vector described.

In the case of the unsupervised HMM, the viterbi output is used to generate a state which is then used as input to the supervised HMM. The supervised HMM then uses viterbi to generate a path through the HMM. The entire output path is treated as a word.

Results

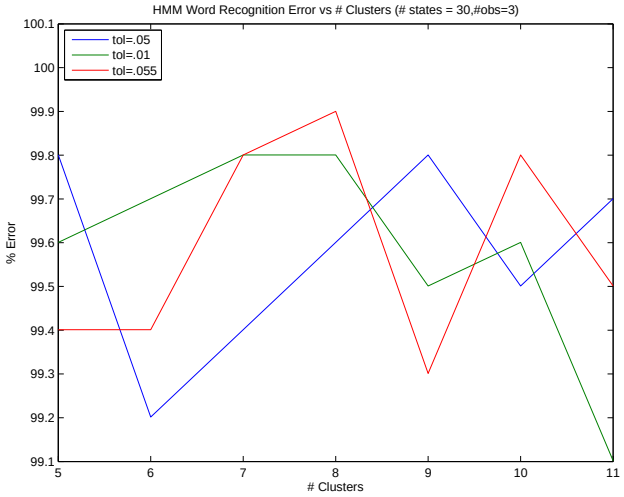
Unsupervised

Unfortunately, the unsupervised algorithm performed very poorly. On individual letters, it gets approximately 12% of letters correct on average.



It's important to note that the initializations are random, so the values will be different each time we run and the values for different tolerances aren't directly comparable, rather in the average case of all, they are comparable.

To make things worse, at the word level, it does even worse, getting less than 1% correct on average.



We believe that this is occurring because of a couple of reasons. First, the images are not segmented correctly. This wouldn't necessarily be too bad but there are also too few examples to be approximately correct with a guess at segmenting the image. Which leads us to the second problem, the number of examples. Training our HMM takes $O(KNM^2)$ computations to compute just the likelihood over all of our examples (where K is the number of letters, N is the number of states, and M is the number of possible values for observations). This still has to converge and the number of steps to converge is proportional to the size of the likelihood matrix and the number of examples. In order to converge with $K = 500, N = 30, M = 15, T = 3$ (where T is the number of observations) takes about 30-45 minutes to converge only to an accuracy of ΔLikelihood= .01. This is first of all very inaccurate and the amount of time to train over larger datasets makes tuning our parameters (the number of clusters, number of observations per letter, and the number of states) difficult.

Supervised

myCAPTCHA

First, we tested the myCAPTCHA dataset. The goal with this test was to ensure the supervised HMM was training correctly, and to try and glean any insight that may aid us in addressing the more-challenging reCAPTCHA dataset. It was also our primary dataset used to debug numerous issues, performance, and algorithm flow.

Each test consisted of randomly selecting 90%, or 1080 images, of the dataset for training, and then using the remaining 10% as testing. In the case that we're presenting today, we conducted this test three times for each value of k, and then averaged the error over each test. Only three tests were used to keep the runtime within reasonable limits. The results below took approximately 8 hours to collect using an 8-core Intel i7-based computer.

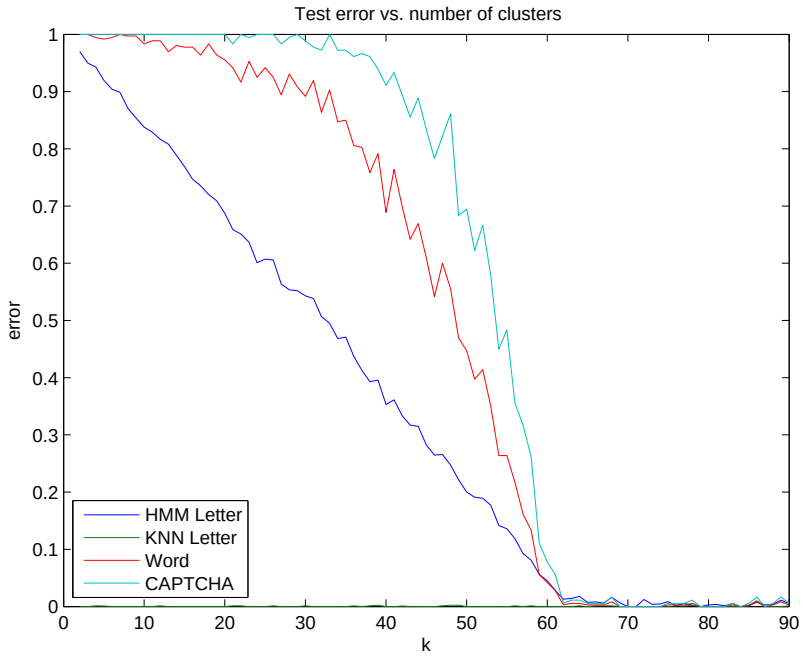


Figure 4: Testing results using the myCAPTCHA dataset.

As Figure 4 illustrates, the HMM's performance is linearly related to the number of clusters being used by K-means in the training algorithm. However, once we reach a cluster count of 62, the HMM does extremely well in identifying letters, and thus words and complete CAPTCHAs. We believe this is due to over-fitting, since there are exactly 62 possible states, or letters, in this dataset. We can get 0% error once the cluster count reaches 70.

As a comparison, we ran K-nearest neighbors (KNN) alongside the HMM using $K = 1$. As you can see, the KNN algorithm gets nearly 0% error in every test over the number of clusters. This was expected, since the dataset contains no deformation in the letters, and there are no shapes to confuse the KNN comparisons.

reCAPTCHA

With the results from the myCAPTCHA testing we set out to address the reCAPTCHA dataset using the same series of experiments. Again, we used 90%, or 900 images, from the dataset for training and 10% for

testing.

We made one modification for this test. Instead of using the KNN results directly as output, we decided to test what would occur if the KNN output was fed into the HMM as the observation vector.

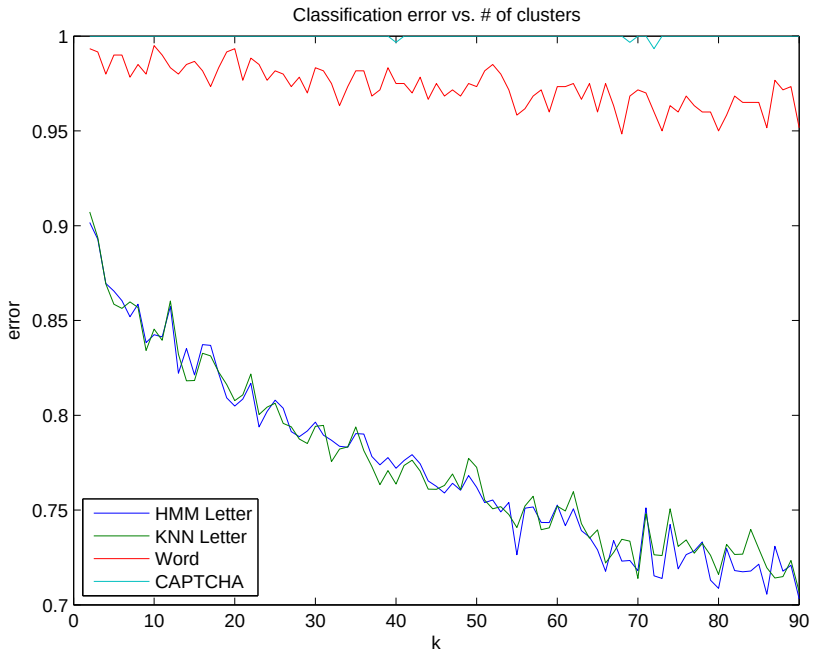


Figure 5: Testing results using the myCAPTCHA dataset.

As Figure 5 illustrates, the HMM's success is no longer linearly related to cluster count, and may actually converge as the number of clusters increases. Furthermore, even with 72 possible states in this dataset, the HMM performance does quite poorly at that point. It isn't until the cluster count reaches 90 that we get the best results based on these tests. We feel we could get better results with higher cluster counts, but that would be severely over-fitting the data.

We did find it interesting that the KNN observation vectors could improve the HMM's performance in select cases. It's not significant, but it is worth noting for further research. Due to computation time, we did not test higher values of K for KNN.

When we test using only the KNN classifier, the error rate is 91% over individual words, and 58.9% over letters. The HMM at that point gets 95% error over words and 70% over letters.

We feel that our success in this dataset hinges primarily on being able to properly segment the character sequences so that a single letter can be observed at one time. However, that was not the focus of this project so further study would be required.

Discussion

While our success rate appears to be low, comparison to previous work indicates that our results may be acceptable. Comparing to Beede's work [2], we're not close since he obtained 2% success on complete CAPTCHAs while we had virtually 0% success. Furthermore, he had 21.5% success on individual words, while our best success using the supervised HMM was roughly 5% success depending on cluster count.

However, according to Wilkins [1] anything over 1% success for CAPTCHAs considers the CAPTCHA implementation "broken". They obtain two measures for this percentage. The first, obviously being, raw success rate on complete CAPTCHAs. The second being "partial CAPTCHAs" based on a calculation using individual word success rates by taking the rate of success on individual words and dividing by two. The reason being that one must only have the "control word" correct to complete the CAPTCHA, and since there are two words you can ignore half of the CAPTCHA. Our individual success rate for words was roughly 5% depending on cluster count, which gives approximately 2.5% success rate for partial CAPTCHAs according to Wilkins. Therefore, our algorithm could potentially "break" the CAPTCHA implementation provided enough resources are available to bombard a website.

There is still a great deal of potential future work. To begin with, segmentation of low-kerning words is a significant challenge, and solving that problem would greatly improve classification results. Furthermore, shape detection and removal (e.g. ellipses) would also greatly improve word segmentation and overall classification. Finally, algorithm optimization could be improved to bring training and testing times down from our current order of hours, or even days, depending on the size of our dataset.

Bibliography

[1] J. Wilkins, "Strong captcha guidelines v1.2." `http://www.bitland.net/captcha.pdf`, 2009.

[2] R. Beede, "Analysis of ReCAPTCHA Effectiveness." `http://www.rodneybeede.com/downloads/CSCI5722\%20-\%20Computer\%20Vision,\%20Final\%20Paper,\%20Rodney\%20Beede,\%20Fall\%202010.pdf`, 2010.

[3] P. Baecher, N. Bscher, M. Fischlin, and B. Milde, "Breaking recaptcha: A holistic approach via shape recognition," in *Future Challenges in Security and Privacy for Academia and Industry* (J. Camenisch, S. Fischer-Hbner, Y. Murayama, A. Portmann, and C. Rieder, eds.), vol. 354 of *IFIP Advances in Information and Communication Technology*, pp. 56–67, Springer Boston, 2011.

[4] E. Bursztein, M. Martin, and J. C. Mitchell, "Text-based CAPTCHA Strengths and Weaknesses," *ACM Computer and Communication security 2011*.

[5] S.-Y. Huang, Y.-K. Lee, G. Bell, and Z.-h. Ou, "An efficient segmentation algorithm for CAPTCHAs with line cluttering and character warping.," *Multimedia Tools and Applications*, vol. 48, pp. 267–289, August 2009.

[6] O. Ludwig Jr., D. Delgado, V. Goncalves, and U. Nunes, "Trainable Classifier-Fusion Schemes: an Application to Pedestrian Detection," in *Proceedings of the 12th International IEEE Conferernce on Intelligent Transportation Systems*, 2009.