The Captchacker Project

March 2009.

Jean-Baptiste Fiot
Ecole Centrale Paris
Ecole Normale Supérieure de Cachan
jean-baptiste.fiot@student.ecp.fr

Rémi Paucher
Ecole Centrale Paris
Ecole Normale Supérieure de Cachan
remi.paucher@student.ecp.fr

Abstract

The Captchacker Project exploits the potential of Support Vector Machines to break visual captchas. We define "simulation-based" and "captcha-based" methods to build our models.

Our method performs extremely well for breaking easily-segmentable captchas, with a robust recognition. We also give a generalization to non easily segmentable – thus harder to break – captchas, using Dynamic Programming. Besides, we explain how to get a large training set without going through the long and boring task of manual labelling.

Finally, our source code is available (for free!) at [19].

1. Introduction

A CAPTCHA is a program that can generate and grade tests that humans can pass but current computer programs cannot [2].

The term CAPTCHA (for Completely Automated Turing Test To Tell Computers and Humans Apart) was coined in 2000 by Luis von Ahn, Manuel Blum, Nicholas Hopper and John Langford of Carnegie Mellon University. At the time, they developed the first CAPTCHA to be used by Yahoo.

CAPTCHAs are widely used: spam comment prevention, website registration protection, online polls, dictionary attacks prevention, search engine bots, worms and spam... As LeMonde newspaper pointed out [1], 200 billion spams are sent every day... so there's definitively a mass market in here!

So if you wanna be able to viagra spam, hack your sista's or gf's MSN account, check this paper out!

Section 2 gives a very quick overview of the common methods in word recognition. In section 3, we explain how to break a simple type of captchas: easily segmentable captchas. In section 4, we move to a harder problem: breaking non easily segmentable captchas. Section 5 explains some of the future challenges captchas breakers will have to overcome. Finally, we conclude this project in section 6.

2. State of the art

To break captchas, a first class of algorithms is based on geometric detections [13, 14, 15, 16, 17]. We did not study these methods, because they are said to be not robust, which sounds pretty intuitive. (We did not check this, as this project is a school project, we had limited time).

Another class of algorithms is based on neural networks [4, 5, 6, 7, 8, 9, 10, 11]. These methods are extremely popular mostly thanks to gurus Y LeCun and G. E. Hinton. We encourage the reader to have a look at "convolutional neural networks" and "deep-belief networks". In this project, we decided not to implement these methods despite their excellent results, because of their relative opacity, and the difficulty to tune parameters to get good models. We nonetheless invite the reader to have a look at Mike O'Neil's information about convolutional neural network implementation [5], which is a good help to understand neural networks at the beginning.

Finally, more common classification methods such as Support Vector Machines are used [18], using subparts of the captchas as input data. This is the method we have chosen to study, on several type of captchas, easily and non-easily segmentable (sections 3 and 4).

3. Easily segmentable Captchas

3.1. Catpchas studied

As a study case, we focused on captchas from Egoshare's website (http://www.egoshare.com/). These captchas can be segmented by thresholding the intensity and separating connected components.

Here are some Egoshare captchas:

They are small 80x25 images which always contain three digits. Characters do not overlap, therefore segmentation is quite easy.

3.2. Preprocessing

Segmentation was implemented in C++ with OpenCV and consists in three steps:

1. First, we convert the image into a gray-scale one, and we threshold it to remove the background;

2. Then, we calculate the three largest connected components with both 4-connectivity and 8-connectivity;

3. Finally, if there are less than three 8-connected components, we have to consider the 4-connected components. Besides if the third smaller 4-connected component is too small, this means that a digit has been split in several different parts, so we have to consider the 8-connected components.

This segmentation performs very well, with a success rate being almost 100%. In our experiment, we tested it on about 3000 captchas, and the segmentation technique failed only once, it was on that captcha:

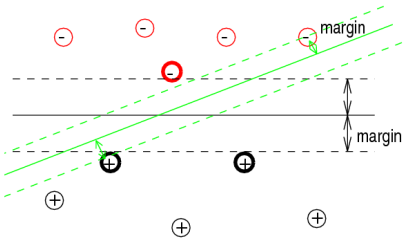
Using color information would easily solve this problem.

3.3. Learning features

3.3.1 Support Vector Machines (SVM)

As explained in section 2, we decided to use SVM to solve this classification problem. We wrote Python scripts using the libSVM Python wrapper for this part.

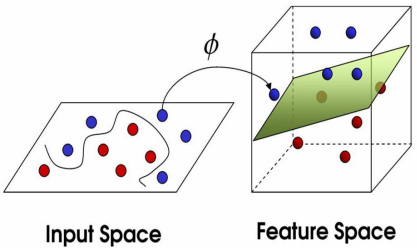
The idea of SVM is to separate classes via a hyperplane. The choice of this hyperplane is based on the maximization of the margin (the min distance between the hyperplane and the two classes).



In this picture, we have two classes, the black hyperplane is the one maximizing the margin, thus chosen to be the decision boundary. The circles with strong edges are the “support vectors”.

When the data are not linearly separable, we can either use a higher dimensional space and/or authorize outliers.

Putting the data into a higher dimensional space can make the classes linearly separable:



Thanks to this trick the optimization algorithm in this new space is similar to the linear case in the original space. However, the Φ application is generally not so easy to find.

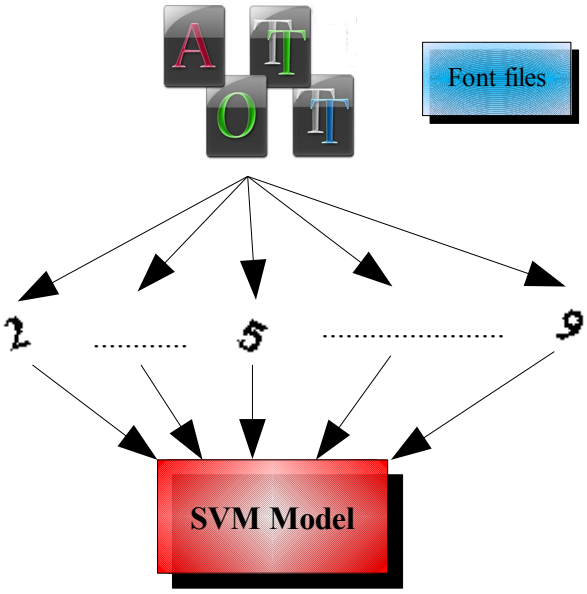
Another idea is to allow outliers, but this implies to choose an error cost C. Choosing the cost C requires to strike a happy medium: a too low cost generates many outliers, and a too high cost generates overfitting.

Here are some illustrations in 2D with two classes, with respectively a low and high error cost:

3.3.2 *"Simulation-based" method*

We call the following method *"simulation-based"* because we use simulated training data to build our models.

In our experiments, we generated the simulation-based database with three different fonts (Californian FB, Comic and Vera) with each time two different thicknesses, a rotation parameter ranging from -30° to 30°, and a scale ranging from 17 to 22 pixels. We ended up with a database containing 1116 images per character.

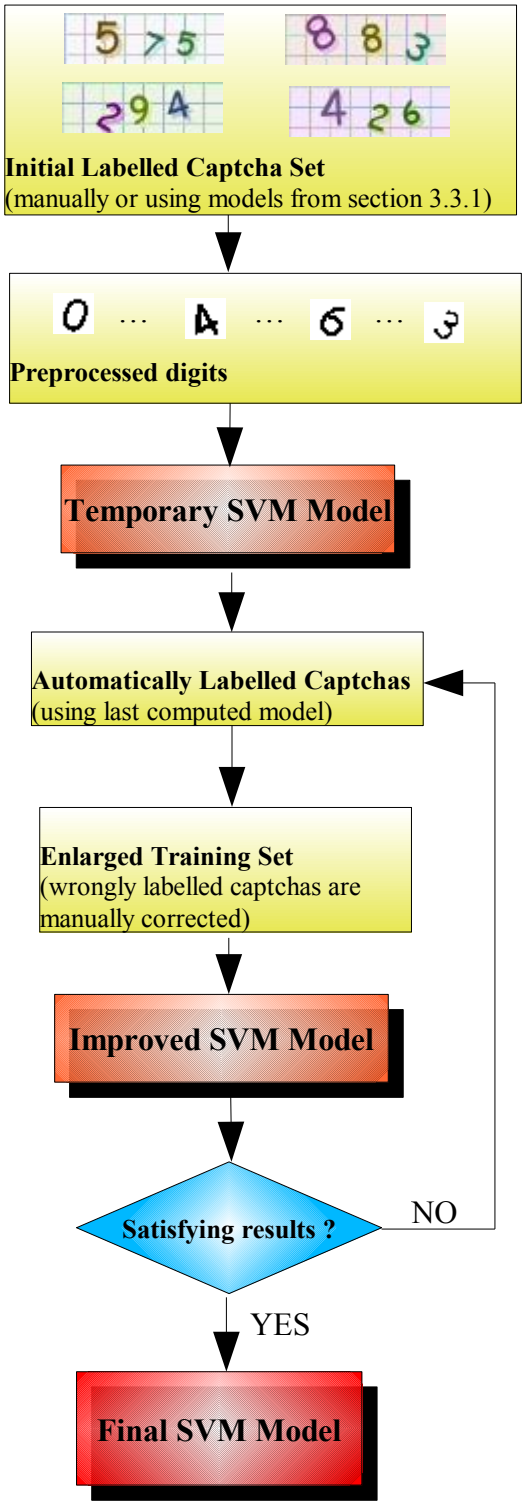


This diagram shows the basic idea of the font-based database generation. Starting from font files, we create rescaled and rotated digits. These digits are centered via a simple C++ program we wrote, and their intensities are normalized. Finally, we used these pictures to build the SVM model.

3.3.3 *"Captcha-based" method*

On the other hand, we worked on a *"captcha-based" method*, meaning our training data was based on captchas from the website.

As we did not want to label millions of captchas manually, we used the following method. We build a first model based on a few labelled captchas (typically a hundred), we use it to label other captchas, we correct manually the wrongly labelled ones, we reinject them in the training set to build a better model, and continue this iterative process until we have satisfying results. This trick is called *"active learning"*.



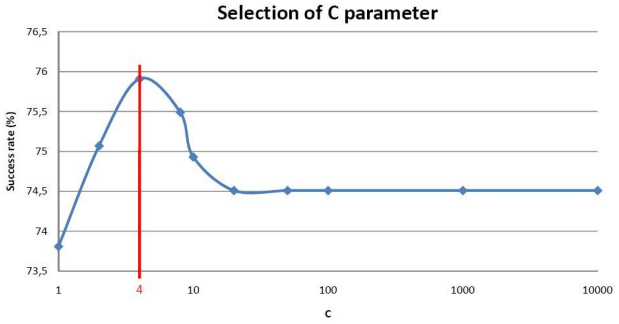
This method, less general than the simulation-based, obviously leads to better results (see next section for performance analysis).

3.4. Performance

Once our two databases were generated, we had to choose the parameters used by the C-Support Vector Classification algorithm. Thanks to cross-validation, we chose the model minimizing the empirical risk on a 714 captcha test set. Our test set was obviously randomly selected, without any common data with the training set of the captcha-based database - otherwise all the success rates would have been biased.

3.4.1 Simulation-based database

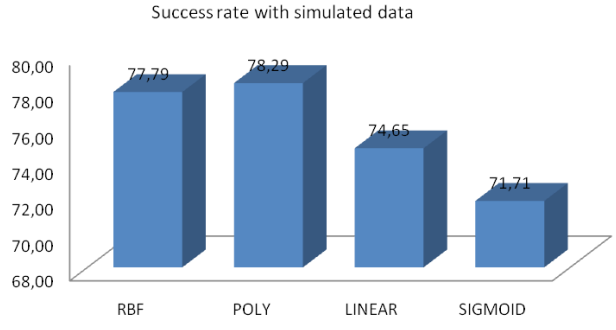
Based on the training set described in section 3.3.1, we trained a multiclass SVM. The error cost was found via a k-fold validation.



As far as the kernel type of the SVM is concerned, we have four different choices in the LibSVM library:

- Radius-Based Function (RBF),
- polynomial kernel,
- linear kernel,
- sigmoid kernel.

Here is a chart illustrating the performance of these different kernels:

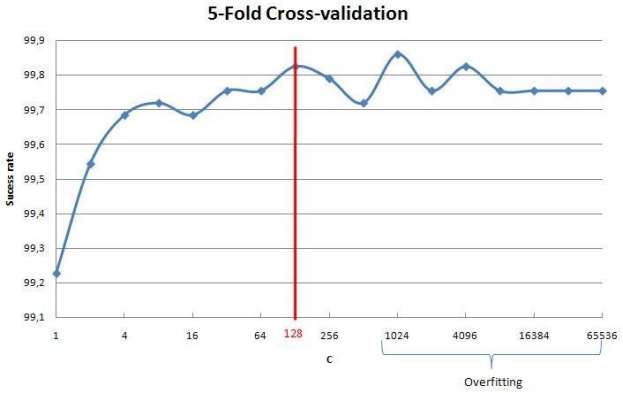


The polynomial kernel (with a degree higher than 3) performs slightly better than the RBF one. The last two have worse performance.

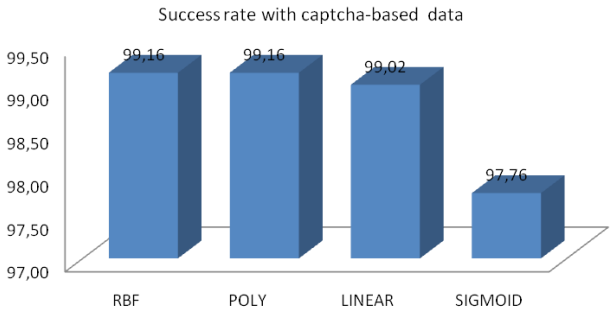
These rates correspond to success rates of well-decrypted captchas, which means that the polynomial kernel produces 92.2% of success per character.

3.4.2 Captcha-based database

The Captcha-based database was generated from 1896 labeled captchas, which makes about 570 images per character. Thanks to cross validation, we find the optimal cost C=128.



The kernel performance chart has similar shape:



As expected, we obtain much better results than with the simulated-based training set, because we used training data much closer to the testing data. However, this approach is less general than the simulation-based one.

3.5. Other easily segmentable captchas

We tried our model with other easily segementable captchas, and it seems to work well – we do not have accurate success rate, as we have not manually labelled captchas from this website.

Here are some breakable examples from other websites:

This shows that our method is quite robust, and can give good results even with a quite specific training set.

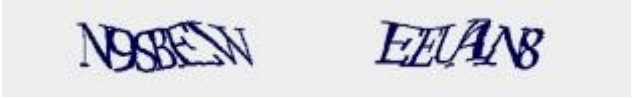
4. Non-easily segmentable Captchas

As once segmentation is done, captchas are easily breakable, companies have begun to design more complex captchas using overlap.

Examples from yahoo.com:

Examples from gmail.com:

In this section we studied the new Hotmail catpchas, whose kind are poorly breakable. Here are examples:



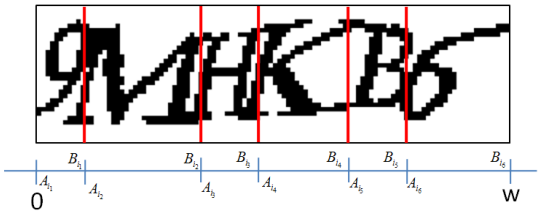
4.1. Dynamic programming

4.1.1 Presentation and formalization of the problem

To break Hotmail captchas, we trained a SVM model on our simulated database. Then, thanks to this model, we are able to associate to each subwindow a prediction (the likeliest class) as well as a score telling how sure we are that this subwindow belongs to this class.

$$\left\{ \left([A_i, B_i], s_i \right) / \left(0 \le s_i \le 1, [A_i, B_i] \subset [0, w] \right) \right\}$$

Given a Hotmail captcha, we would like to segment the image in six rectangular subwindows making a partition of the captcha, that is to say finding six non overlapping segments [Ai, Bi] that cover entirely the original image and maximizing the sum of the likelihoods s_i over the six subwindows.



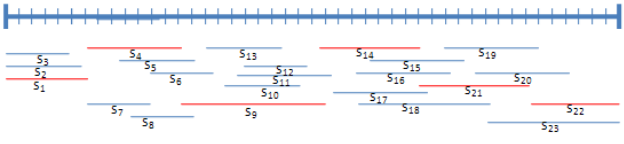
This is basically the following discrete optimization problem:

$$\max_{(i_1,i_2,i_3,i_4,i_5,i_6)} s_{i_1} + s_{i_2} + s_{i_3} + s_{i_4} + s_{i_5} + s_{i_6}$$

given the constraints:

$$\begin{cases} A_{i_1} = 0 \\ B_{i_6} = w \\ \forall k \in [1,5] \; B_{i_k} = A_{i_{k+1}} \end{cases}$$

More visually, given a set of labeled segments, the problem consists in choosing the set of six distinct segments that cover entirely the whole segment.



In this example, the valid solution is the set of red segments.

In our case, the set of segments will have all segments of lengths from 9 (small characters like "3") to 30 (large letters like "M" or "W").

4.1.2 Resolution of the problem

To solve this type of problem, scanning all possible solutions one by one is not an option, since there is a very large number of solutions, so the computation time would be extremely high.

By *"path to a point K"*, let us refer to a sequence of non-overlapping segments making a partition of [0, K].

As we have the two constraints (the number of six segments and no overlapping), classic methods of dynamic programming consisting in defining a function at each point telling which path is the best to get to this point, cannot be applied directly and need to be improved.

To solve the problem, we associated to each point a *dictionary* telling the optimal path to get to this point for a path length from 1 to 6, each key L representing the optimal path from 0 to this point of length L. The solution of the problem is the sixth key of the dictionary at the point of abscissa w (the best 6-long path to get to the last point).

The problem is now to determine this function at each horizontal point of the image. The idea is that, given a new segment i, we will consider the concatenation of the previous optimal paths to A_i with $[A_i, B_i]$ and we will see if this path has a better score than the one in the dictionary of point Bi. So here is the algorithm:

- Sort the list of labeled segments with respect to B_i
- Initialize the target function at point 0 (One key in the dictionary representing the 0-long path)
- For each segment I in the list of segments:
  - For each path P in the dictionary of the point A_i:
    - Let L_j be the length of the path;
    - If $L_j > 5$, continue (the concatenation of the path and the segment will have a length higher than 6)
    - Else, if the path of length L_{j+1} in the dictionary of Bi has a lower score than the path P+I, replace it with the path P+I

So for each horizontal position in the image, we end up with the optimal paths of all lengths from 1 to 6.

This method is very efficient in terms of computation time (O(n log (n)). The slow part was to determine all scores on every subwindow. Indeed, we used here the Python Image Library, so cropping images before computing the score and then converting the image to a list of coefficients was very slow – all these operations are not handled natively. The best way would have been to do that in C++.

4.1.3 Results

The model we choose has to be very good to give the best performance. Indeed the score has to be very high if and only if the right character has been predicted.

We noticed that the higher the number of classes is, the harder it is becomes to obtain good results. Indeed some classes are not well detected (in our test the 'D' character was always detected as a 'U'). When dealing with many classes, it becomes slower and slower to build the model. It took us one hour to build a 12 class model. Our model was not so good, so we did not obtain very good results.

As we have just been lent a PC from our school's Math Lab, we have launched a 36 class model computation – with a much larger training set than we can afford on our laptops. The results do not make any sense, the letter Q is over-represented in the results. Given the long computing time to build one full 36 class model, we have not tried other models with different parameters yet.

4.1.4 Improvement ways

To perform better regarding the model, our training set needs to be improved. For example, we need to model the distortion the same way that it is modeled in Hotmail captchas. In our training set, we only considered cosines distortions whereas distortion in Hotmail captchas if far more complicated. This is why some characters are not well detected in our model.

We also noticed that, despite the very poor recognition results with the sliding window method, the captchas were rather well segmented. In our tests our segmentation success rate was about 40%. Therefore, we could use the same captcha-based method to generate our training set based on segmented characters in original captchas, as we did with Egoshare captchas.

Once we would have improved the 36 class model (see 4.1.3), we will try to use it to segment automatically captchas, then build a captcha based model, and apply the active learning technique to get better and better models.

We will need to switch some Python parts to C++ (see 4.1.2), because so far the segmentation lasts about 40 seconds (on a 2GHz PC, with the 12 class model build – not the 36 class model).

5. Future battles in the world of Captchas.

As you know, the captcha world also has his battles. Hackers want quick and robust guesses in their programs, and designers want captchas the easiest possible to understand for humans, and the hardest for algorithms.

In this project, we focused on visual captchas, but here are some other ideas, more or less used: audio captchas, animated captchas, …

6. Conclusion

To cut a long story short, this project well combined theoretical studies and implementation, in such an exciting subject: captcha breaking.

We would like to thank our instructor Iasonas Kokkinos for all the time he spent and for his good advice.

References

[1] 200 milliards de spams sont envoyés chaque jour dans le monde. http://www.lemonde.fr/technologies/article/2009/02/09/200-milliards-de-spams-sont-envoyes-chaque-jour-dans-le-monde_1152784_651865.html#ens_id=1152867

[2] http://www.lafdc.com/captcha/

[3] K. Chellapilla, P. Simard. Using Machine Learning to Break Visual Human Interaction Proofs (HIPs)

[4] Y. Lecun, L. Bottou, Y, Benshio, P. Haffber. Gradient-Based Learning Applied to Document Recognition. Proc. of the IEEE, November 1999.

[5] Mike O'Neill. Neural network for Recognition of Handwritten Digits.

[6] L. Bottou, Y, Benshio, Y. Lecun. Global Training of Document Processing Systems using Graph Transformer Networks.

[7] Y. Lecun, L. Bottou, Y, Benshio. Reading checks with multilayer graph transformer network.

[8] P. Simard, D. Steinkraus, J. Platt. Best Practices for Convolutional Nerural Networks Applied to Visual Document Analysis.

[9] C Burges, O. Matan, Y. LeCun, J. Denker, L. Jackel, C. Stenard, C. Nohl, J. Ben. Shortest Path Segmentation: a method for training a Neural Network to Recognize Character Strings.

[10] D. You, G. Kim. An approach for locating segmentation points of handwritten digit strings using a neural network.

[11] Geoffrey E Hinton's homepage. http://www.cs.toronto.edu/~hinton/

[12] G. Mori, J. Malik. Recognizing Objects in Adversial Clutter: Breaking a Visual Captcha.

[13] S. Huang,, Y. Lee, G. Bell, Z. Ou. A projection-based Segmentation Algorithm for Breaking MSN and Yahoo capchas.

[14] J. Yan, A.S. El Ahmad. A low cost attack on a Microsoft Catpcha.

[15] J. Sadri,C. Y. Suen, Tien D. Bui. New Approach for segmentation and recognition of handwritten numeral strings.

[16] E. Vellasques, L.S. Oliveira, A.S. Britto Jr, A.L. Koerich, R. Sabourin. Filtering Segmentation cuts for digit string recognition.

[17] L. de Oliveira, E. Lethelier, F. Bortolozzi, R. Sabourin. Handwritten Digits Segmentation based on Structural Approach.

[18] L. S. Oliveira, R. Sabourin. Support Vector Machines for Handwritten Numerical String Recognition.

[19] The Captchacker Project Home Page http://code.google.com/p/captchacker