

The Subversion of CAPTCHAs through Machine Learning Algorithms

Word Count: 3990

Abstract

This essay examines the potential of machine learning as a means to subvert text-distortion CAPTCHAs. The research question is: “Which machine learning algorithm can more effectively solve text-distortion CAPTCHAs: ‘k-nearest neighbours’, or ‘classification and regression trees’.

The effectiveness of the machine learning algorithms was considered both theoretically, and with a practical experiment. To practically demonstrate the effectiveness of each algorithm, both were given 500 CAPTCHAs to solve. The CAPTCHAs were generated by an open-source PHP script Securimage, though they were first processed with a custom written de-cluttering script tailored to Securimage CAPTCHAs. Effectiveness was evaluated based on the percentage of characters each algorithm was able to correctly identify within the CAPTCHA.

It was found that the k-nearest neighbours algorithm was able to identify characters in CAPTCHAs with a higher accuracy than the classification and regression trees algorithm. These results are explained by the theory of each algorithms function. The data ultimately suggests that the k-nearest neighbours algorithm is more effective than the classification and regression trees algorithm for solving text-distortion CAPTCHAs.

Word Count: 179

Table of Contents

1 - Introduction	4
2 - Machine Learning	4
2.1 - K-nearest neighbours	6
2.2 - Decision Tree Induction	8
3 - CAPTCHAs	10
4 - Tests	11
4.1 - Simplifying Classification	12
4.2 - Creating Training Data	14
4.3 - Feature Selection	15
4.4 - Analysis	16
4.5 - Limitations	18
5 - Conclusions	20
Appendix A - Training Data	22
Appendix B - Machine Learning Script	22
Works Cited	33

1 - Introduction

As computer technology has advanced throughout the years, computers have been able to perform more complex tasks. Machine learning is one advancement in computer science that gives computers greater problem solving ability. The field allows computers to recognize patterns in data, much like humans do. This has a wide variety of applications in many different fields; for example, machine learning has been used to both beat the Go world champion, and also as a tool for early stage cancer detection (DeepMind; Kourou, et al. 8). Two common machine learning algorithms were investigated: “k-nearest neighbours”, and “classification and regression trees”. The investigation was centred around solving CAPTCHAs with these algorithms. A CAPTCHA is intended to be solved by only humans. This ensures that bots cannot abuse web services with spam, for example. If machine learning algorithms can solve CAPTCHAs with a degree of reliability, it identifies a fundamental flaw within the CAPTCHA system.

2 - Machine Learning

Machine learning is a field of computer science study that focuses on giving computers “the ability to learn without being explicitly programmed” (Simon, 89). A machine learning algorithm can be supplied different inputs, and then make decisions based on this data. This broad definition has resulted in many different algorithms and techniques being included under the name of machine learning. One category of algorithms is the “supervised learning” set (Brownlee). A supervised learning algorithm is first supplied example training data. This is then utilized when the algorithm is provided an input. The algorithm references the training data in

order to produce the most relevant possible output. Supervised learning falls into two general categories: classification and regression (Brownlee). They both achieve similar goals: both attempt to predict a desired output from the input, based on previous examples. Regression, however, exists within the scope of rational numbers, whereas classification has a predefined list of options to which the input should be matched. Regression therefore allows for an output never seen before in the training data; in comparison, classification must always match to a list of options, even if the input doesn't greatly resemble any of the examples in the training data.

In supervised learning, the efficacy of the machine learning algorithms relies largely on the quality of the training data. An entry in the training dataset would contain a number of features (consistent with all other entries in the dataset), accompanied with one designated label (Gordon, *Hello World - Machine Learning Recipes #1*). A feature is simply a variable (expressed numerically) that describes some quality of the entry. The label defines what the entry should be classified as. When classifying a new piece of data, the machine learning algorithm considers all the features given in the input, then based on the trends observed in the training dataset, assign it the most likely label (Gordon, *Hello World - Machine Learning Recipes #1*). This combination of machine learning algorithm and supplied training dataset is referred to as a classifier (Gordon, *Hello World - Machine Learning Recipes #1*). All features included should be simple and relevant; that way the classifier is more likely to understand how each individual affects the outcome of the label (Gordon, *What Makes a Good Feature? - Machine Learning Recipes #3*). If an irrelevant feature is included, there is a possibility that the algorithm will perceive a feature-label relationship where one doesn't actually exist. When later testing the classifier on unseen data, it will attempt to apply this conjectured relationship, rather than basing its prediction on

information that actually matters. This is likely to lower correct classification rates, reducing the efficacy of the classifier.

K-nearest neighbours and decision tree induction are both machine learning algorithms which can be applied in a supervised learning context, for classification purposes. Although both rely on mathematics to perform classifications, the calculations performed to achieve classification are completely different. These different approaches can result in different outputs on occasion, even when given an identical input and training dataset, despite having the same objective.

2.1 - K-nearest neighbours

The K-nearest neighbours algorithm works by keeping the training dataset stored in memory for comparisons. Each entry in the dataset can be considered as a point, its location represented by the various features it possesses. For example, if a dataset had three features, it could be plotted on the x, y, and z axis as a visual representation. For any more than three features, the concept remains the same; it just becomes harder for humans to visualize as the number of axes outnumbers regular three-dimensional space. When a new input is given for classification, it is similarly plotted among the training dataset. The algorithm then calculates the k (where k is defined by the user) nearest neighbours of the input point. The input point is then classified based on the labels of the nearby points.

Calculating Nearest Neighbours

The standard Euclidean distance function is used to calculate the nearest neighbours of the given input point (Larose, 99). In standard two dimensional space the formula is:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Given entry A, and entry B, with respective features $A_1, A_2 \dots A_n$ and $B_1, B_2 \dots B_n$, the formula is adapted to be as follows:

$$\sqrt{(B_1 - A_1)^2 + (B_2 - A_2)^2 + \dots + (B_n - A_n)^2}$$

The main issue with this formula is equal representation of every feature. All features must be represented numerically to work in mathematical computations. If one feature is a boolean for example, it would typically be given a value of 0 for false, and 1 for true. Consider three example entries:

	Gender (0=male, 1=female)	Height (cm)
Entry 1	1	170
Entry 2	0	180
Entry 3	1	180

After using the Euclidean distance formula the results are:

Distances	
Entry 1 vs Entry 2	10.05
Entry 1 vs Entry 3	10.00
Entry 2 vs Entry 3	1.00

Entries with a smaller distance are considered more similar. Therefore the 180 cm female is considered 10 times more similar to the 180 cm male, than the 170 cm female. Whether this is an accurate evaluation of similarity depends on the situation, but most often it is ideal for all features to be represented equally. This should not only be considered for binary values, but the scale of all features. If another feature was added such as “average sleep per night in hours”, it

would likely be represented more than the binary 0/1 gender value (given that the difference could be around 3-4), but less than the height value (difference anywhere from 10-50).

2.2 - Decision Tree Induction

Decision tree induction is the process of forming a decision tree based on information provided in the training data. Unlike k-nearest neighbours, decision tree induction is a set of algorithms rather than an algorithm in singular. The decision tree formed is essentially a cascading list of if-then-else-statements, with each statement based on a singular feature that the data contains (Tan, 150). The statements should narrow down the possibilities of what the label could be, until a sole answer is reached (Tan, 150). Therefore, every entry in the training dataset will reach its proper label if it is evaluated by the decision tree, due to the tree being based on the training dataset in the first place. This provides a method with which to similarly evaluate test data. Because a piece of test data will have all the same features, it can follow the same decision tree and reach a predicted label. Whether this label is correct is based on how well the decision tree generalizes patterns in the training dataset, as well as the extent to which each piece of test data follows these patterns.

One example of a decision tree induction algorithm is the CART (Classification and Regression Trees) method. All decision tree algorithms induce the tree by splitting the training dataset up according to features it possesses (Tan, 150). In each split that CART makes the criteria is based on only one feature, and separates the dataset into only two new categories (Tan, 156). To split the dataset, two pieces of information must be determined: the best possible feature to split on, and where in the range of feature values to split (Tan, 160-162). There are many ways

of determining this information; one method commonly used by CART implementations is the Gini index (Tan, 158).

The Gini index is a measure of how impure a dataset is, measured on a scale of 0 to 1 (Tan, 159). If a dataset has a Gini index of 0, it is completely pure; a value of 0 signifies that every entry in the dataset belongs to the same classification (Tan, 159). If the Gini index is instead 1, the dataset is completely impure; every single classification possible is equally represented inside the dataset (Tan, 159). The goal when splitting the parent dataset is to reduce the impurity as much as possible in each subsequent child dataset. That way the Gini index reaches 0 as quickly as possible, therefore also isolating each label in the dataset with as few statements as possible.

Given a database with classes $B_1, B_2, \dots B_n$, the Gini index is calculated as follows (Tan, 159):

$$1 - ((P(B_1))^2 + (P(B_2))^2 + \dots + (P(B_n))^2)$$

Where $P(B_n)$ is the probability of a classification being B_n if it was chosen at random. For example, given a dataset with 2 classes: B_1 and B_2 and entries consisting of $[B_1, B_1, B_2, B_2, B_2]$, the Gini index would be:

$$1 - ((2/5)^2 + (3/5)^2) = .48$$

The Gini index can also be used to determine which feature is the best to split upon. To do this, each possible split condition is considered (Tan, 162). The Gini index is then calculated for both of the potential subclasses given the specific split condition. From there each index is multiplied by the percentage of entries that went into that class, then added together (Tan, 160). This value determines the goodness of the split (Tan, 160). The goodness of the split should be as

low to zero as possible. The lower the value, the better the reduction of the Gini index. When all possible split conditions have been considered, the split with the lowest goodness is chosen and added to the decision tree. This process continues until every Gini index reaches 0, and therefore all entries have been classified.

3 - CAPTCHAs

A “CAPTCHA” is a “Completely Automated Public Turing Test To Tell Computers and Humans Apart” (Chellapilla, et al. 1). The Turing Test is named after its inventor, Alan Turing. A conventional Turing Test is run by humans; after a series of interactions a human attempts to determine whether they interacted with another person, or with a computer. The computer attempts to mimic human like behaviour such that the two entities are indistinguishable. CAPTCHAs facilitate the same process of distinguishing between computer and human; the main difference is that the trial needs to be “completely automated”, and therefore a computer must conduct the test and determine the outcome, which is counter to the conventional roles. The very first CAPTCHA was developed in 1997 by scientists at the company AltaVista (Chellapilla, et al. 1). AltaVista operated a search engine, which allowed administrators to add their own URLs to their database. However, this service was found susceptible to bots automating the process of repeatedly adding the same URL, which would then give the perpetrator an unfair advantage over competitors (Chellapilla, et al. 1). To resolve this conundrum, the AltaVista scientists pioneered the first CAPTCHA based on the human ability to recognize text (Chellapilla, et al. 1).

In a text-recognition CAPTCHA, the CAPTCHA generated will display warped text in an image format; this way the result is in an obscure format for computers, yet still recognizable for humans. This type of challenge is what the CAPTCHA relies on; any given CAPTCHA test must be easily solvable for humans, yet in an area that computers are not yet skilled enough to surmount (Chellapilla, et al. 2). Visual recognition CAPTCHAs therefore rely upon human pattern recognition ability.

A group of Microsoft researchers at the Second International Workshop for Human Interactive Proofs deemed that the “human success rate [when attempting to solve a CAPTCHA] should approach 90%”, whereas “automatic scripts should not be more successful than 1 in 10,000 (0.01%)” (Chellapilla, et al. 2). If success rates drop below 90% for human CAPTCHA solving, the website administrators run the risk of frustrating their users, making it less likely for them to use the website (Chellapilla, et al. 2). Thus, web services such as reCAPTCHA have been developed; these provide CAPTCHAs from a generator that has already been tested for solvability. That way web developers can simply query the service for a CAPTCHA, instead of running their own CAPTCHA generator, which could be liable to unsuitable difficulty levels.

4 - Tests

After researching the difference between CART and k-nearest neighbours, tests were orchestrated in order to compare their efficacy with regard to solving word recognition CAPTCHAs. To generate the CAPTCHAs the PHP script “Securimage” was used. A service was needed that allowed for bulk generation of CAPTCHAs along with their solutions in plain text, and Securimage provided that feature. It was also the third result found after Googling for “Open

Source CAPTCHA generator”. Other popular services that were browsed (such as visualCaptcha and reCAPTCHA) were either not based on word recognition, or hosted as a CAPTCHA provider, so bulk access was infeasible.

4.1 - Simplifying Classification

One complication that is necessary to address when solving word recognition CAPTCHAs is the large number of possible combinations when stringing together letters. Securimage defines its character set to use on generation as:

```
public $charset='ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz23456789';
```

(Drew)

This character set uses a large portion of the alphabet, uppercase and lowercase, accompanied by 8/10 numerical digits. Although no reason is listed in the source code for these missing characters, one can only assume that it is due to their visual similarity with one another. For example, a warped “Q” could appear similar to either a “0” or an “O”.

By default, Securimage automatically generates CAPTCHAs with 6 characters (Drew). Given that the variable \$charset contains 50 characters, each individual character in the CAPTCHA has 50 possible options to choose from. With 6 characters, the number of possible CAPTCHA combinations given by Securimage equals 50^6 combinations, or 15.625 billion. For the machine learning algorithm to recognize all 15.625 billion possible combinations, it would require a training dataset with every single one of these combinations included at least once. Ideally each combination would be expressed more than once, that way variations in the appearance of the image due to Securimage’s distortion was accounted for. This is obviously not

a feasible expectation for the training dataset; a dataset of this size would take up massive amounts of hard drive space and also result in extremely slow classification.

One way to reduce the number of combinations, is to separate the CAPTCHA image into individual characters. That way, the classifier needs to pick only from 50 possible options, rather than 15.625 billion, thereby reducing the scope of operations. This transforms recognizing the CAPTCHA from a challenging, time consuming classification problem that's vast in scope, into 6 classification problems that are much more reasonable.

Although it may be possible to use machine learning to separate characters it wouldn't be through using classification techniques. It was therefore decided to write the character separation portion of the script manually through trial and error, that way all machine learning would be focused solely on letter classification. An example CAPTCHA from the test data, code "sywbwa", looks so when first generated:



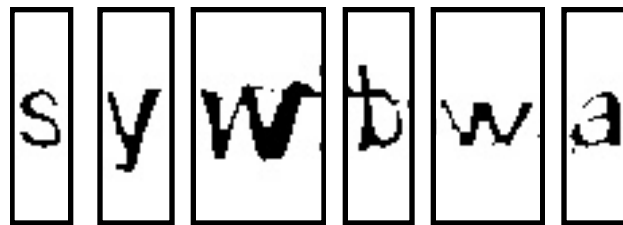
The image processing removes some of the darker lines obscuring the image (they are a consistent colour each time), and converts each pixel to either completely black (0, 0, 0), or white (255, 255, 255):



This image is then taken and processed so that any black pixel with a white pixel bordering it (in both cardinal and sub-cardinal directions) is removed. This has the added effect of shrinking the letters in the CAPTCHA.



The image is then separated into individual characters determined through finding columns with no pixels. If there are less than 5 of these spaces, columns with the lowest number of black pixels are chosen.



4.2 - Creating Training Data

To establish a training dataset Securimage was instructed to create 500 new CAPTCHAs, along with their solutions. Each of these images were processed and separated into different characters. It was manually ensured that each CAPTCHA was properly separated; any that failed to separate were removed. Of the 500 generated, 368 were separated correctly, and thus used as training data. This created a total of 2,208 different characters. Another script cropped and centred each letter to 63 by 57 pixels (this value was based on the largest character when cropped perfectly). Each image was then added to the training data folder; each type of character was

automatically sorted into subfolders to keep track of classification. This is why being able to generate CAPTCHA solutions was an important factor in the CAPTCHA generator, otherwise characters would have had to be manually sorted. Characters that varied largely between capital and lowercase (a/A, b/B) were sorted into different classifications whereas characters with similar shapes (c/C) remained in the same group. Securimage is not case-sensitive when inputting CAPTCHA solutions, so grouping the characters by similarity of shape was an effort to simplify matters for the classifier (Securimage).

4.3 - Feature Selection

The features for the training data were based off of the pixel density. Each character fit inside the same 63 by 57 pixel dimensions, so that way each feature was able to represent the value of a certain pixel in a specific location consistent among all examples. Originally each feature was just a 0 or a 1 indicating pixel presence, but that gave poor classification results. The main problem with binary encoding is that every pixel is valued equally, but really, a few stray pixels off to the side should not be as significant as one that lays inside the core shape of the character. The solution was to give each pixel a value from 0 to 9, based on how many pixels were filled inside the 3x3 grid around it. That way, the machine learning algorithms could differentiate between a pixel being there at random, with a low pixel density around it, versus more important pixels with consistently high values among many examples. See Figure 1 below, on each pixel its density is also displayed. Note the extra pixels hanging from the top arch of the two shape, with a value of “2” and “4” respectively. Because these don’t have high values, they are not interpreted as high importance to the shape of the image.

4.4 - Analysis

The scikit-learn Python library was used in order to perform the analysis. Scikit-learn is a reputable machine learning library used by companies such as Spotify and Change.org (Scikit-Learn). The library provided implementations of both the k-nearest neighbours and CART algorithms to ensure the machine learning was being performed correctly. For the tests 500 new CAPTCHAs were generated, each one was processed with the letter separation script, and then the pixel data of those characters was fed into the machine learning algorithms as features. Both algorithms received the same 500 test CAPTCHAs to guarantee fairness. The results of each CAPTCHAs processing can be seen below in Figure 2 and 3:

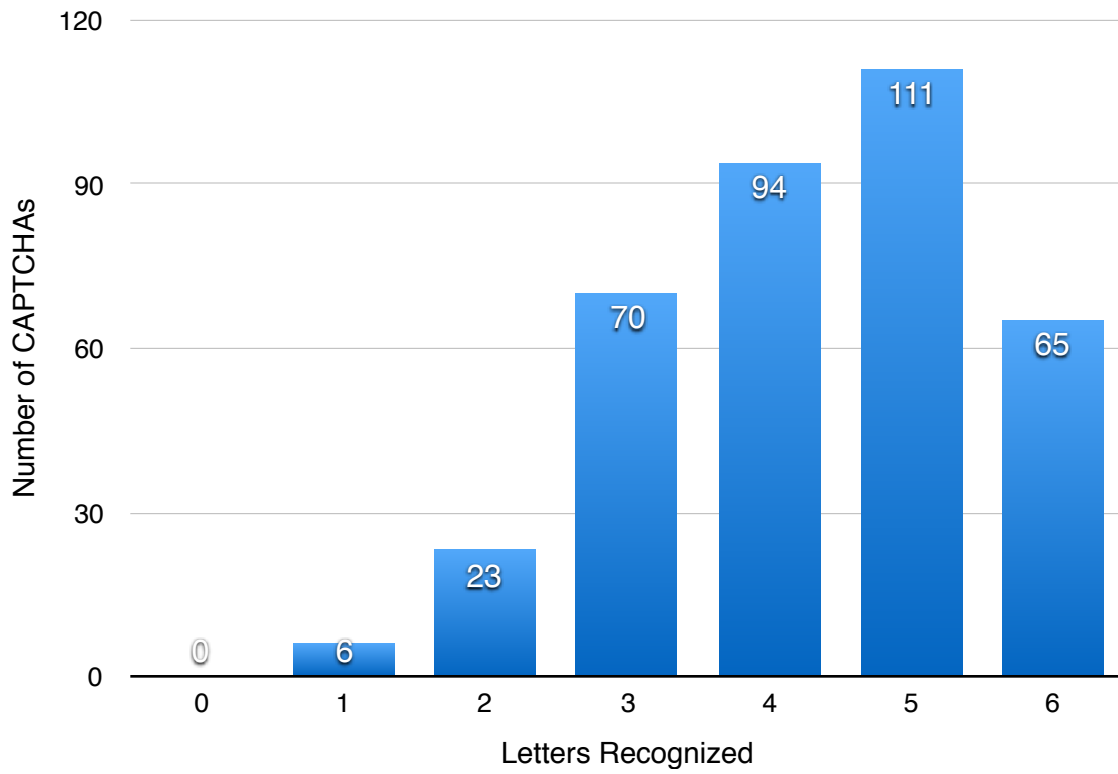


Figure 2

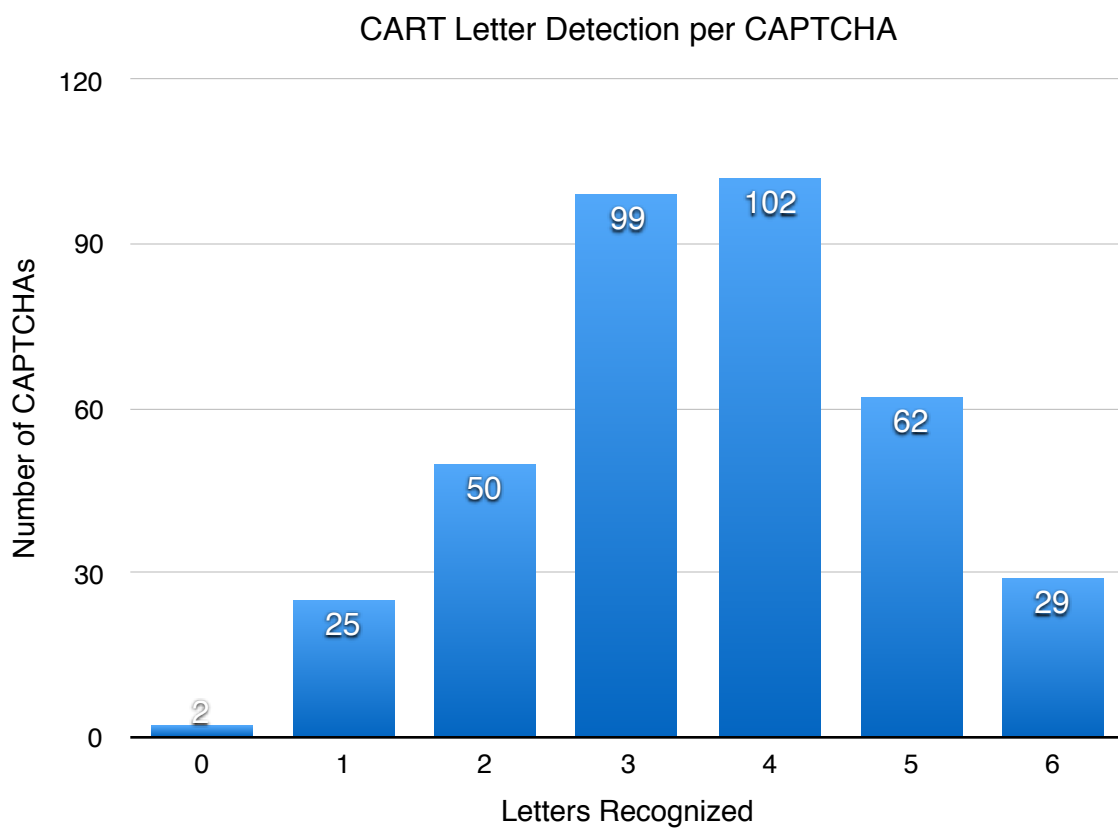


Figure 3

In total, 369 CAPTCHAs were successfully separated into characters. To solve a CAPTCHA, all 6 character must be identified correctly. KNN was therefore able to solve 65/369, or $\approx 17.6\%$, and CART was able to solve 29/369, or $\approx 7.86\%$. Although both were able to easily subvert the expected 1/10,000 solve rate, the KNN algorithm proved to be more accurate than CART in CAPTCHA solving and CAPTCHA letter recognition. CART was only able to recognize 1314 letters, or 59.35%, whereas KNN was able to recognize 1583 letters, or 71.50%.

It was ultimately the design of the CART algorithm that made it less effective in comparison to KNN. The principal difference between the two is the extent to which each values features. While KNN is able to consider all pixels at once when classifying, CART classifies based on the values of a few specific pixels. The specificity of CART's decision making is an obvious issue. It would be an unreasonable expectation for a human to classify letters based on only a few specific pixels. It is even more unreasonable to expect a computer, an ignorant entity relative to a human, to do the same. This is why KNN is more effective for this specific task; the algorithm more accurately mimics how human letter recognition works. Although possibly CART's deficiencies could be mitigated through better feature selection, it would unlikely bridge the ability gap between the two algorithms. CART is simply not as suited to classifying characters as KNN is.

4.5 - Limitations

One large limitation of solving CAPTCHAs with machine learning is the specificity to which the classifier is trained. The Securimage character separation tool relies heavily on the number of characters and colour values in the image being consistent. The classifier relies on the same font being used so characters contain the same general shape. If any one of these properties

were altered, characters would not resemble those in the training dataset, and therefore successful classification rates would drop. The classifier and character separation tool are also both tailored specifically towards Securimage CAPTCHAs; classification of a CAPTCHA represented in a different format would likely not receive successful classification rates near what was achieved for Securimage. To properly classify a text-recognition CAPTCHA produced by different software, a new letter separation method would need to be designed, and the classifier would need to be trained on data retrieved from this software. A classifier that could solve multiple types of CAPTCHAs is hypothetically possible using machine learning, yet would require training data from any type of CAPTCHAs intended to solve. It would additionally warrant the use of more complex features than just pixel values. Ideally this solve-all classifier would be able to recognize and classify based on general shapes of the letters, due to raw pixel values being liable to variation over different CAPTCHA types. The limitation in this case is not due to faults with machine learning, but instead based on the amount of time, resources, and knowledge of the individual undertaking the creation of the classifier.

Another possible hindrance in the way of using machine learning to solve CAPTCHAs is their increasing complexity. reCAPTCHA, a CAPTCHA service created by Google, claims to be “the most widely used CAPTCHA provider in the world” (ReCAPTCHA). Although reCAPTCHA initially relied on distorted text for human verification, they recently created a new API, the “No CAPTCHA reCAPTCHA” (ReCAPTCHA). This uses “advanced risk analysis techniques”, utilizing “a broad range of clues that distinguish humans from bots” (reCAPTCHA). The important factor here is that the clues mentioned, aren’t described in detail. Upon investigation, one group found that all Javascript provided by Google was obfuscated, “to

prevent analysis by third parties” (Sivakorn, 1). This makes machine learning classification defunct in solving this CAPTCHA type. Text recognition CAPTCHAs have an easily understandable goal, with simple information provided: use the image given to correctly identify the text, then the CAPTCHA is solved. Although in the new system the intent is still there to solve the CAPTCHA, now there is no way of knowing what conditions need to be fulfilled to pass the test. This now takes the shape of a reinforcement learning problem, in which a program must discover the ideal circumstances to replicate in order to solve the CAPTCHA. Although machine learning techniques could still be used to help subvert CAPTCHAs; classification techniques are only going to become less effective. The crux is that CAPTCHA providers have deemed classification problems too simple for computers.

5 - Conclusions

Overall, it was shown that the k-nearest neighbours algorithm is more effective at solving text-distortion CAPTCHAs than the CART algorithm. The tests also proved that visual CAPTCHAs are no longer effective for distinguishing between humans and computers consistently. While a 1/10000 solve rate was deemed an ideal strength, KNN was able to solve approximately 1/6 CAPTCHAs. This has large implications for any website still attempting to recognize bots with text-distortion CAPTCHAs. A malicious user could potentially collect data based on the text-distortion CAPTCHAs they are using, then train bots with machine learning to bypass the verification step. This would then allow bots to mass enact on the user’s behalf, spreading spam or buying out tickets for example. At this point, text-distortion CAPTCHAs are

no longer secure. The implementation of a text-distortion CAPTCHA should be seen as a security flaw, as it can no longer sufficiently distinguish between humans and computers.

Appendix A - Training Data

Including all the training data into the appendix of this EE would be impractical, as the file is 23 mb of text and would take up literally hundreds of pages. As such, the information has been hosted online, and can be viewed at one's leisure. The link to access all the information can be found here:

https://www.dropbox.com/sh/5rsh34n73wz183t/AAB4sOLQW5Z8_AOXXDK6OBAsa?dl=0

This folder contains all the training data, the script used to solve new CAPTCHAs, and various other scripts written during the process of assembling and moving around training data. Also included is a modified version of Securimage. No aspects of the CAPTCHA generation was changed, just a few extra lines of code were added in order to save CAPTCHAs to a folder rather than display them on the webpage.

All the Python scripts were written in version 2.7.10. Some scripts require the Sci-kit Learn Library and/or the Pillow graphics processing library.

Appendix B - Machine Learning Script

This is the script that was used to recognize the solve the CAPTCHAs. Although it is probably easier to view it online from the Dropbox link, IB requirements state that the script must be included in its entirety in the appendix. It was also written in Python, requiring both the Sci-kit learn and the Pillow graphics processing libraries.

```
from PIL import Image
import os.path
import math
#Density info contains the training data from all the images
```

```

from densityinfo import *
from sklearn.neighbors import KNeighborsClassifier
from sklearn import tree
#Classification must be based on numbers instead of strings.
This array relates the number that each character is classified
as to its string value
reverseArray = ["2", "3", "4", "5", "6", "7", "8", "9", "a",
"aCapital", "b", "bCapital", "c", "d", "dCapital", "e",
"eCapital", "f", "fCapital", "g", "gCapital", "h", "hCapital",
"k", "l", "lCapital", "m", "mCapital", "n", "nCapital", "p",
"r", "rCapital", "s", "t", "tCapital", "u", "v", "w", "y",
"yCapital", "z"]

#classifier either gets defined as KNN classifier or as decision
tree classifier
#classifier = tree.DecisionTreeClassifier()
#classifierType = "CART"
classifier = KNeighborsClassifier(15)
classifierType = "KNN"

#features and labels are both contained in the densityinfo
training data file
classifier.fit(features, labels)

#This function is used later in splitting characters -
#If there's multiple bridges of the same column, the ones in the
center are the most likely to bridge the two letters together
def compare(array1, array2):
    midpoint = (array1[3]+array1[2])/2
    array1distance = abs(midpoint-array1[1])
    array2distance = abs(midpoint-array2[1])
    if (array1distance < array2distance):
        return -1
    elif (array1distance == array2distance):
        return 0
    else:
        return 1

imagecounter = 0
#this is the path to the file location of all the newest
CAPTCHAS
while os.path.exists("generatenewcaptchas/newcaptchas/" +
str(imagecounter) + ".png"):
    if (not os.path.exists("results/" + classifierType + "/" +
str(imagecounter) + "/")):

```

[illegible]


```

minimumxrange = max(0, minimumxrange-1)
maximumxrange = min(214, maximumxrange+1)
minimumyrange = max(0, minimumyrange-1)
maximumyrange = min(79, maximumyrange+1)

if (len(locations) == 0):
    newimage.putpixel((x, y), (0, 0, 0))
else:
    #find if the surrounding pixels have more white spaces
or character colored spaces, decide whether to include it in new
image or not

    whitespaces = 0
    goodspaces = 0

    for location in locations:
        minimumxrange = max(0, location[0]-1)
        maximumxrange = min(214, location[0]+1)+1
        minimumyrange = max(0, location[1]-1)
        maximumyrange = min(79, location[1]+1)+1
        for testx in range(minimumxrange, maximumxrange):
            for testy in range(minimumyrange, maximumyrange):
                testr, testg, testb = image.getpixel((testx,
testy))

                if (testr == 255 and testg == 255 and testb ==
255):
                    whitespaces += 1
                elif (testr == 140 and testg == 140 and testb ==
140):
                    goodspaces += 1

    if (whitespaces > goodspaces):
        newimage.putpixel((x, y), (255, 255, 255))
    else:
        newimage.putpixel((x, y), (0, 0, 0))

image = newimage
#this is the newest version of the CAPTCHA we're going to
refer to from now on

newimage = Image.new("RGB", (215, 80), (255, 255, 255))

```

#create another new image in order to make more modifications to

#go through each pixel, only include it if it's completely surrounded by other black pixels

```

for x in range(0, 215):
    for y in range(0, 80):
        r, g, b = image.getpixel((x, y))

        if (r == 0 and g == 0 and b == 0):
            minimumxrange = max(0, x-1)
            maximumxrange = min(214, x+1)+1
            minimumyrange = max(0, y-1)
            maximumyrange = min(79, y+1)+1
            failedTest = False

            for testx in range(minimumxrange, maximumxrange):
                for testy in range(minimumyrange, maximumyrange):
                    testr, testg, testb = image.getpixel((testx, testy))
                    if (testr == 255 and testg == 255 and testb == 255):
                        newimage.putpixel((x, y), (255, 255, 255))
                        failedTest = True
                        break

                if (failedTest):
                    break

            if (not failedTest):
                newimage.putpixel((x, y), (0, 0, 0))

```

image = newimage

#save modifications again

#this is to determine where the splits should be made in order to separate the image into characters

startedTracking = False

positionList = []

imageList = []

```

for x in range(0, 215):
    columnpixels = 0;
    for y in range(0, 80):
        r, g, b = image.getpixel((x, y))
        if (r == 0 and g == 0 and b == 0):
            columnpixels += 1

```

```

if (columnpixels > 1 and not startedTracking):
    #new start of letter, add to position list
    startedTracking = True
    positionList.append(x)
elif (columnpixels < 1 and startedTracking):
    #end of same letter, add to position list
    startedTracking = False
    positionList.append(x)

#to get 6 letters the image needs to be split in 12 places.
This indicates there isn't enough splits (characters bridged by
distortion)
while (len(positionList) < 12):
    largestwidth = 0
    largestindex = 0
    #the current character we have with the largest width is
probably the one that has two letters and needs to be split
    for loop in range(0, len(positionList)/2):

        startX = positionList[loop*2]
        endX = positionList[loop*2+1]
        if (endX-startX > largestwidth):
            largestwidth = endX-startX
            largestindex = loop

    startX = positionList[(largestindex*2)]
    endX = positionList[(largestindex*2)+1]
    splitLocation = 0
    bridgeCheck = 1
    #goes through the "character", finds the point where there
are the least amount of pixels in a column, thats most likely to
be where the bridge is
    while (splitLocation == 0):
        columns = []
        for x in range(startX, endX):
            columnpixels = 0
            for y in range(0, 80):
                r, g, b = image.getpixel((x, y))
                if (r == 0 and g == 0 and b == 0):
                    columnpixels += 1

            if (columnpixels == bridgeCheck):
                columns.append([columnpixels, x, startX, endX])

```

```

    if (len(columns) > 0):
        sortedcolumns = sorted(columns, cmp=compare)
        splitLocation = sortedcolumns[0][1]

    bridgeCheck += 1

    #recreate the position list, accounting for the new splits
    that need to be made
    newPositionList = []
    for loop in range(0, len(positionList)/2):
        if (largestindex != loop):
            startX = positionList[loop*2]
            endX = positionList[loop*2+1]
            newPositionList.append(startX)
            newPositionList.append(endX)
        else:
            startX = positionList[loop*2]
            endX = positionList[loop*2+1]
            newPositionList.append(startX)
            newPositionList.append(splitLocation)
            newPositionList.append(splitLocation)
            newPositionList.append(endX)

    positionList = newPositionList

    #nothing is done for letters that have been split in two by
    columns, because the issue did not occur frequently, and it
    would have been more challenging to solve than

    successfulSplitting = False
    if (len(positionList) == 12):
        successfulSplitting = True
        #go through each character, encode as its own image
        for loop in range (0, 6):

            startX = positionList[loop*2]-3
            endX = positionList[loop*2+1]+4

            newimage = Image.new("RGB", (endX-startX, 80), (255, 255,
255))

            for x in range(startX, endX-1):

```

```

    for y in range(0, 80):

        r, g, b = image.getpixel((x, y))
        newimage.putpixel((x-startX, y), (r, g, b))

    imageList.append(newimage)

#get what each letter is actually meant to be

    characterfile = open("results/" + classifierType + "/" +
str(imagecounter) + "/0.txt")
    characterstring = characterfile.read()

    predictionList = []
    imageBinaryList = []
    for loop in range(0, 6):
        #for each image, crop to the bounds it occupies
        image = imageList[loop]
        width, height = image.size
        firstxPixel = 0
        firstyPixel = 0
        lastxPixel = 0
        lastyPixel = 0
        breakstatement = False
        for x in range(0, width):
            for y in range(0, height):
                r, g, b = image.getpixel((x, y))
                if (r == 0 and g == 0 and b == 0):
                    firstxPixel = x
                    breakstatement = True
                    break

            if (breakstatement):
                break

        breakstatement = False
        for y in range(0, height):
            for x in range(0, width):
                r, g, b = image.getpixel((x, y))
                if (r == 0 and g == 0 and b == 0):
                    firstyPixel = y
                    breakstatement = True
                    break

            if (breakstatement):
                break

```

```

breakstatement = False

for x in range(width-1, -1, -1):
    for y in range(0, height):
        r, g, b = image.getpixel((x, y))
        if (r == 0 and g == 0 and b == 0):
            lastxPixel = x
            breakstatement = True
            break

    if (breakstatement):
        break

breakstatement = False

for y in range(height-1, -1, -1):
    for x in range(0, width):
        r, g, b = image.getpixel((x, y))
        if (r == 0 and g == 0 and b == 0):
            lastyPixel = y
            breakstatement = True
            break
    if (breakstatement):
        break

#create new cropped image
newImage = Image.new("RGB", (lastxPixel-firstxPixel,
lastyPixel-firstyPixel), "white")
for x in range(firstxPixel, lastxPixel):
    for y in range(firstyPixel, lastyPixel):

        r, g, b = image.getpixel((x, y))
        newImage.putpixel((x-firstxPixel, y-firstyPixel), (r,
g, b))

image = newImage
width, height = image.size

newImage = Image.new("RGB", (63, 57), "white")
blankSpaceX = 63-width
blankSpaceY = 57-height

#center image in 63*57 box (bounds found from largest
character found in test data)

```

```

        for x in range(max(0, int(math.floor(blankSpaceX/2))),
            int(min(62, math.ceil(blankSpaceX/2+width)))):
            for y in range(max(0, int(math.floor(blankSpaceY/2))),
                int(min(56, math.ceil(blankSpaceY/2+height)))):

                r, g, b = image.getpixel((x-
int(math.floor(blankSpaceX/2)), y-int(math.floor(blankSpaceY/
2))))
                newImage.putpixel((x, y), (r, g, b))

            newImage.save("results/" + classifierType + "/" +
str(imagecounter) + "/letter" + str(loop) + "center.png")

#encode features based on pixel density
imageFeatures = []
for x in range(0, 63):
    for y in range(0, 57):
        minimumxrange = max(0, x-1)
        maximumxrange = min(62, x+1)+1
        minimumyrange = max(0, y-1)
        maximumyrange = min(56, y+1)+1
        pixelcounter = 0
        for testx in range(minimumxrange, maximumxrange):
            for testy in range(minimumyrange, maximumyrange):
                testr, testg, testb = newImage.getpixel((testx,
testy))

                if (testr == 0 and testg == 0 and testb == 0):
                    pixelcounter += 1
            imageFeatures.append(pixelcounter)

        imageBinaryList.append(imageFeatures)

#Now, the script has characters separated, and each one
encoded as features
#At this point, it can either be saved and used as
training data, or continue on and be classified by the machine
learning algorithm

for loop in range(0, 6):
    prediction = classifier.predict([imageBinaryList[loop]])

    newfile = open("results/" + classifierType + "/" +
str(imagecounter) + "/letter" + str(loop) + "prediction.txt",
"w")
    newfile.write(reverseArray[prediction])
    newfile.close()

```

```
predictionList.append(reverseArray[prediction])

if (not successfulSplitting):
    print("failed")

newfile = open("results/" + classifierType + "/" +
str(imagecounter) + "/finalprediction.txt", "w")
finalstring = ""
for loop in range(0, 6):
    finalstring = finalstring + predictionList[loop]

newString = finalstring.replace("Capital", "")
newfile.write(newString)
newfile.close()
imagecounter += 1
```


Works Cited

"AlphaGo | DeepMind." DeepMind. Google, n.d. Web. 06 Oct. 2016.

Brownlee, Jason. "A Tour of Machine Learning Algorithms." Machine Learning Mastery.

Machine

Learning Mastery, 25 Nov. 2013. Web. 21 Aug. 2016.

Chellapilla, Kumar, Kevin Larson, Patrice Y. Simard, and Mary Czerwinski. "Building Segmentation Based Human-Friendly Human Interaction Proofs (HIPs)." Human

Interactive Proofs Lecture Notes in Computer Science (2005): 1-26. Web.

Hello World - Machine Learning Recipes #1. By Josh Gordon. Perf. Josh Gordon. Hello World -

Machine Learning Recipes #1. YouTube, 30 Mar. 2016. Web. 17 June 2016.

Kourou, Konstantina, Themis P. Exarchos, Konstantinos P. Exarchos, Michalis V. Karamouzis,

and Dimitrios I. Fotiadis. "Machine Learning Applications in Cancer Prognosis and

Prediction." Computational and Structural Biotechnology Journal 13 (2015): 8. Web.

Larose, Daniel T. Discovering Knowledge in Data: An Introduction to Data Mining. Hoboken,

NJ:

Wiley-Interscience, 2005. 90-106. Print.

Phillips, Drew. Securimage. Computer software. Securimage. Vers. 3.6.4. Drew Phillips, 3 Mar.

2016. Web. 18 June 2016.

"ReCAPTCHA." Easy on Humans, Hard on Bots. Google, n.d. Web. 30 Aug. 2016.

Shet, Vinay. "Are You a Robot? Introducing "No CAPTCHA ReCAPTCHA"" Google Online

Security Blog. Google, 03 Dec. 2014. Web. 30 Aug. 2016.

Simon, Phil. Too Big to Ignore: The Business Case for Big Data. N.p.: n.p., n.d. 89. Print.

Sivakorn, Suphannee, Jason Polakis, and Angelos D. Keromytis. "I'm Not a Human: Breaking the Google ReCAPTCHA." I'm Not a Human: Breaking the Google ReCAPTCHA

(2016):

1. Web. 28 Aug. 2016.

Tan, Pang-Ning, Michael Steinbach, and Vipin Kumar. "Classification: Basic Concepts, Decision Trees, and Model Evaluation." Introduction to Data Mining. Boston: Pearson Addison Wesley, 2005. 145-62. Print.

What Makes a Good Feature? - Machine Learning Recipes #3. By Josh Gordon. Perf. Josh Gordon. What Makes a Good Feature? - Machine Learning Recipes #3. Google, 27 Apr. 2016. Web. 17 June 2016.

"Who Is Using Scikit-learn?" Who Is Using Scikit-learn? Scikit-Learn, 22 Aug. 2016. Web. 06 Oct. 2016.