Department of Information Technology Limerick

Institute of Technology

An Examination on the Robustness of Alphanumeric CAPTCHA as a Challenge Response Test using Machine Learning

K00205031

April 2019

This work is submitted in part fulfilment of the requirements for the degree of

Bachelor of Science (Software Development)

Abstract

Alphanumeric Completely Automated Public Turning Tests to Tell Computers and Humans Apart (CAPTCHAs) are widely used across the internet as a security mechanism to prevent automated processes. Studies continue to show that CAPTCHAs can be automatically resolved, due to enhancements in computer vision and machine learning technologies. This project aims to examine the robustness of three popular CAPTCHAs: Really Simple CAPTCHA, Pastebin CAPTCHA and Python CAPTCHA 0.3, by developing a modular machine learning system. The machine learning system was able to achieve average success rates of 97% for Really Simple CAPTCHA, 94% for Pastebin CAPTCHA and 33% for CAPTCHA 0.3. With such success rates, it can be argued that alphanumeric CAPTCHAs are becoming insecure for modern-day technologies.

Acknowledgements

I owe my deep gratitude to Mary Nolan, who continues to encourage and support me. I would not be where I am today without her.

I would like to express my deep gratitude to both my supervisors Alan Ryan and Brendan Watson, for their patient guidance, enthusiastic encouragement and useful critiques. I would also like to thank Niall Corcoran, for their advice and assistance throughout. My special thanks are extended to the Software Development faculty members.

I am also grateful to Ciarán Hehir who supported me throughout this venture.

Finally, I wish to thank Ciarán Quillinan, David Wallace, Declan O' Halloran and Lee O'Neill for their support and encouragement throughout my study.

Table of Contents

List of Figures

List of Tables

1.0 Introduction

CAPTCHA "**C**ompletely **A**utomated **P**ublic **T**uring Test to Tell **C**omputers and **H**umans **A**part" or HIP "Human Interactive Proofs" are challenge-response tests created by a program that generates and grade these tests that most humans can easily pass and current computer programs cannot (Von Ahn, Blum, and Langford, 2002). The aim of this project is to automatically solve CAPTCHAs that are in use today, by creating a machine learning program and examining the robustness of these types of challenge-response tests.

During the last few years, computer vision has improved due to new technologies. One of these technologies is a Convolutional Neural Network (CNN) (Hong, et al., 2015). This paper will attempt to show that a CNN, which is a class of deep, feedforward artificial neural network, most commonly applied to analysing visual imagery is able to achieve a high success rate at solving different types of CAPTCHAs (Stark, et al., 2015). Challenge response tests were created by Alan Turing as a way to differentiate between humans and Artificial Intelligence (AI) (Turing, 2009). CAPTCHAs were created with the intention of exploiting a security problem to advance the field of Artificial Intelligence as stated in Von Ahn, et al., 2003, pg 294. The ability to break CAPTCHAs using automated programs has led to improvements in the challenge response test (Moradi, and Keyvanpour, 2015), (Kolupaev, and Ogijenko, 2008). CAPTCHA breaking benefits other fields such as image labelling and verifies the secureness of existing CAPTCHAs.

Many variations of CAPTCHAs exist ranging from text, image, audio and alternative solutions such as Google's reCAPTCHA which is image checkbox based. An alphanumeric CAPTCHA requires a user to correctly transcribe multiple distorted characters within a generated image. These types of CAPTCHAs will be discussed in depth, in chapter two.

Many older CAPTCHAs have become vulnerable to attacks as discussed by Chen, et al., but there are still many websites that continue to use alphanumeric-based CAPTCHAs such as Wordpress and Microsoft. Breaking CAPTCHAs using image segmentation and machine learning are not new concepts and have been proven to work on CAPTCHAs used by Microsoft (Hong, et al, 2015.). The author's intention for this project is to add additional research to this field.

CAPTCHA is a popular security mechanism to prevent automated interactions with a site such as registration, commenting and automated scraping. It would be uncommon for CAPTCHA to appear when a user requests a simple interaction with a site such as searching but multiple search requests within a timeframe could possibly flag a security feature such as CAPTCHA to validate those requests. Google has a similar feature to detect bot search requests; once flagged the bot will be required to resolve a checkbox image CAPTCHA. Until that challenge response test is resolved the bot will be unable to continue with its automation.

This project will focus on examining the robustness of alphanumeric CAPTCHAs by developing a modular machine learning system. Section 2 presents a review on CAPTCHA generation breaking and different CAPTCHA types that currently exist, including difficult overlapping types. Section 3 provides an explanation of convolution neural networks. Section 4 describes the steps and techniques involved to automatically resolve a CAPTCHA. Section 5 documents the systems architecture and requirements. Section 6 explains in depth how the system is implemented. Finally, section 7 is a reflection on the project, providing additional research around CAPTCHA breaking and their robustness.

2.0 CAPTCHA

CAPTCHA was created by Andrei Broder and his colleagues in 1997 (Banday, and Shah, 2011, pp 67). They discuss different aspects of CAPTCHA such as generation methods, types, usability purposes and the advantages and disadvantages of different CAPTCHAs. This CAPTCHA was quickly proven to be inadequate due to the improvement of Optical Character Recognition (OCR) success rates (Kopp, et al., 2017). OCR is a technology where characters are automatically recognised through an optical mechanism (Mithe, et al., 2013). Therefore, anti-automated recognition techniques for text based CAPTCHAs were developed. Two of those techniques are anti-segmentation and anti-recognition. Anti-segmentation and anti-recognition are processes which makes it difficult for an automated process to identify characters within a CAPTCHA. Methods used to prevent segmentation are character overlapping, random dot sizes and counts, straight or wavy lines with different widths, etc. An example of this is where neighbouring letters are slightly overlapping, creating a conjoined handwriting appearance and adding random lines or dots to the image. Anti-recognition methods that are commonly used are different text sizes, fonts and character rotation to reduce the accuracy of a classifier (Bursztein, et al., 2014). Figure 2.1 shows both of these techniques implemented.

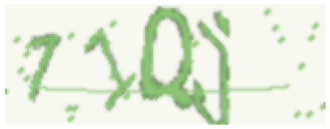


Figure 2.1: CAPTCHA 0.3, with noise and overlapping characters.

In a 2011 study carried out by Fidas, et al., it was found that CAPTCHAs are an annoyance to a genuine user as they can be difficult for humans to resolve and that 48.5% of users were able to solve a CAPTCHA challenge on their first try, while the remaining 51.5% required two or more tries on average. Additionally, 61.4% of users identified character distortion as the main obstacle when attempting to resolve a CAPTCHA.

2.1 Text CAPTCHA

Text-based CAPTCHAs have either digits, letters or both of these combined with distortion so that OCR cannot resolve the text and requests the user to identify the text string to type in a text box provided near the CAPTCHA. If a CAPTCHA can be recognised by OCR it is considered ineffective (Chellapilla, and Simard, 2005). A CAPTCHA should resist existing resolving software but must be solved with a high success rate by a human. One of the popular CAPTCHA on the internet is Really Simple Captcha, a Wordpress plugin with over one million installations (WordPress.org., 2018). Figure 2.2 below is an example of Really Simple CAPTCHA.

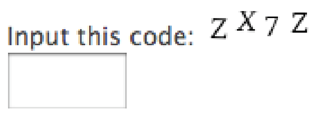


Figure 2.2: Really Simple CAPTCHA, a Wordpress plugin.

2.2 Audio CAPTCHA

CAPTCHAs have always been problematic, users can become frustrated when being challenged with a test especially when it may require multiple attempts. Users that were most affected by this were those with visual impairments or reading disorders such as dyslexia. An alternative to image-based CAPTCHAs was introduced: audio CAPTCHAs. Audio CAPTCHAs play an audio recording that involves reading out letters and numbers. Background noise and sound distortion are applied to prevent text-to-speech automation. Most internet services provide an audio CAPTCHA along with a visual CAPTCHA. (Darnstädt, et al., 2014) In 2013, Sano, Otsuka, and Okuno were able to crack Google's reCAPTCHA 2013 with 58.75% accuracy. Such accuracy is enough to consider those audio CAPTCHAs ineffective. Google then made improvements to their CAPTCHA system and in 2017, Hughey, and Levin were able to resolve their audio CAPTCHA with 85.15% accuracy in 5.42 seconds on average. They called their resolution software unCaptcha which utilised free online speech-to-text engines. It allowed for a large-scale attack on the reCAPTCHA system using minimal resources. Figure 2.3 is an example of Google's audio CAPTCHA.

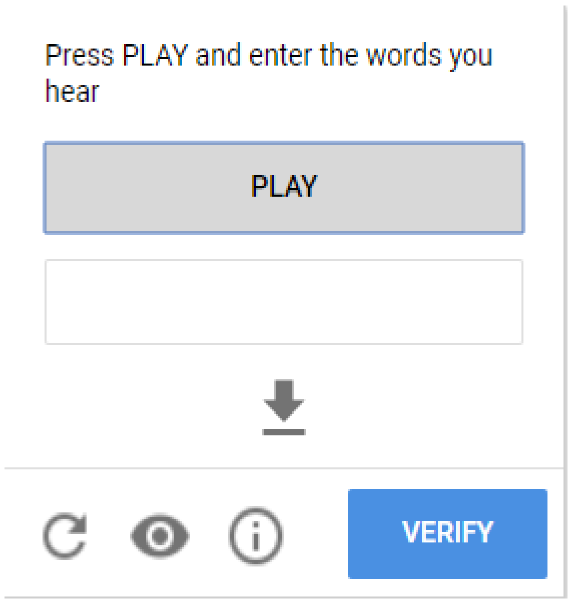


Figure 2.3: Google Audio reCAPTCHA.

2.3 CheckBox CAPTCHA

CheckBox CAPTCHA displays a checkbox of images with a given keyword and the task is to select all the images that are associated with the keyword. A user is allowed to have one incorrect answer to successfully solve the CAPTCHA. In 2016, Sivakorn, Polakis, and Keromytis developed their own reCAPTCHA automation technology which has a 44.3% resolution accuracy. Google has updated its reCAPTCHA service to version 3, which uses a scoring system between 0.0 and 1.0 based on interactions with a website. 1.0 indicates a good interaction with a website, whereas a score of 0.0 is very likely a bot. Based on that score, the webmaster can specify to display a CAPTCHA. (reCAPTCHA: Easy on Humans, Hard on Bots. 2018). Figure 2.4 is an example of Google's reCAPTCHA V2.

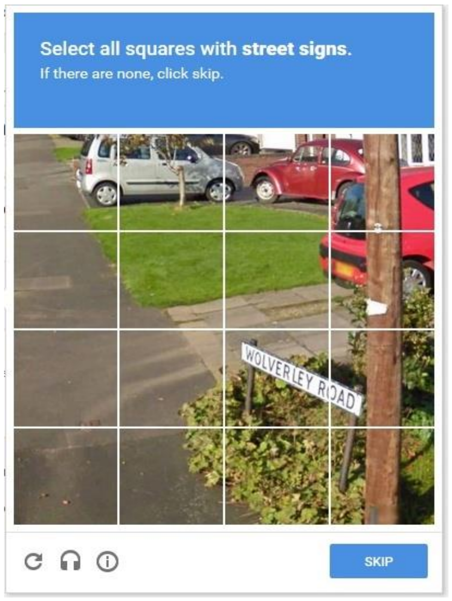

Select all squares with **street signs.**
If there are none, click skip.
WOLVERLEY ROAD
SKIP

Figure 2.4: Google reCAPTCHA V2.

2.4 CAPTCHA Generation

The main steps involved in the generation of an alphanumeric CAPTCHA image are setting the width, height and background for the CAPTCHA. The background can be a pattern, image or colour. *n* random characters are generated from a predefined set, normally the set is a mix of alphanumeric characters excluding ‘L/l’, ‘I/i’, 1, ‘J/j’, ‘O/o’ and ‘0’ since they are commonly mixed up. A font style, size and colour is then chosen, these can also be picked at random from a predefined set or just one. The characters may be tilted left or right slightly to help prevent OCR. A CAPTCHA test is usually case insensitive. Finally, noise such as lines, circles or other objects is generated on the CAPTCHA image which further helps prevent OCR. CheckBox CAPTCHA uses a random image with one to many of the same objects within it such as street signs or cars. For example, Google reCAPTCHA V2 uses this approach. Its generation logic is not public so it is assumed the image is selected from their Google Maps service using object detection technology such as a CNN to select the appropriate image. Similarly, object detection technology was also used to defeat reCAPTCHA V2 as proven by Shen, Li and Jiao at GeekPwn 2018, International Cybersecurity and AI Contest. They could successfully resolve a CAPTCHA if it was a street sign selection as the CNN was only trained on street signs (Li Shen, et al., 2019).

A request is made when a client's computer accesses a web server, if the requested web resource is protected e.g. by CAPTCHA then access would only be granted if the CAPTCHA test was passed. The web server will likely use a CAPTCHA web service or their own CAPTCHA generation algorithm to generate a CAPTCHA. Different CAPTCHA services/algorithms have different techniques for the generation of the challenge-response test, but algorithms normally use pseudorandom letters and numbers to generate a CAPTCHA because of its speed in comparison to a real random number (Pope, and Kaur, 2005). The CAPTCHA solution and the GUID "Global Unique Identifier" i.e. the client is stored on the web server. The GUID ensures that only the client that requested that specific CAPTCHA can produce a valid solution. Alternatively, the CAPTCHA solution and GUID may be stored in an encrypted cookie on the client's machine. The server verifies the CAPTCHA test by comparing the GUID and CAPTCHA solution stored within the cookie with the CAPTCHA submission (Banday, and Shah, 2011). If the CAPTCHA test is passed access is granted otherwise a new CAPTCHA image is generated for another attempt. Some CAPTCHA implementations may block users from attempting the challenge again for some time if they repeatedly fail the test.

2.5 CAPTCHA Breaking

Security researchers continue to investigate methods into resolving CAPTCHA. It is suggested that if a CAPTCHA challenge is resolved successfully more than 0.01% by a computer program then the scheme is considered broken (Chellapilla, and Larson, 2005). However, the use of CAPTCHA and Information Technology (IT) security is constantly changing. Research into this topic found that a more conservative accuracy of 1% is becoming more widely used in studies. A common approach in existing CAPTCHA-breaking solutions are pre-processing, character segmentation, post-processing and finally character recognition. Pre-processing is image transformations such as greyscaling, a kind of black-and-white or grey monochrome. Pre-processing helps make the following CAPTCHA-breaking stages easier. Segmentation is the isolation of each character within a CAPTCHA image. Post-processing helps remove any noise that may be in the isolated character. Recognition is the classifying of an isolated character. Each of these stages is discussed more in depth in the methodology section. A CNN is commonly used for character recognition as well as object classification. However, CNN's require a large training set for classification. Unfortunately for CAPTCHA recognition, there are little training sets available. Alternatively, online CAPTCHA-breaking services exist. These services, such as 2captcha.com, pay humans per CAPTCHA they resolve successfully and return the resolved CAPTCHA to the user via an Application Program Interface (API) for a small fee. 2captcha currently charging a rate of $0.84 per 1000 normal CAPTCHAs and $2.99 per 1000 reCAPTCHAs. This approach has a high success rate but can be costly. CAPTCHA breaking creates a win-win situation as hackers are creating new algorithms to solve CAPTCHAs, helping to advance the field of AI and security (Von Ahn, et al., 2002.).

2.6 Conclusion

CAPTCHA has been in use since 1997 and continues to be the preferred method of automation prevention. But simply put users do not like CAPTCHAs and implementing a cost-efficient CAPTCHA, with usability features, as a security mechanism is a difficult task. There are multiple types of CAPTCHA, that are constantly changing due to technology becoming more sophisticated, to choose from. The next section will discuss CNNs, one of the main technologies used to automatically resolve CAPTCHA images.

3.0 Convolutional Neural Network

A Convolution Neural Network is a machine learning model that is one of many different Artificial Neural Networks (ANNs) that are most commonly applied to analyzing visual imagery. ANNs are a computational model that represent the structure of biological neural networks by connecting artificial neurons, forming a network. ANNs are an attempt to mimic how the human brain works by inputting data. The ANN will then observe, learn and hopefully detect patterns that are far too complex for a human programmer to extract and teach the machine to recognize.

CNNs are a feed-forward neural network, meaning that the output of each layer is forwarded to the next layer, forming a connection. The connection between layers never forms a loop. CNNs, like neural networks, are made up of neurons with learnable weights and biases. (Krizhevsky, et al., 2012) CNN image classifications take an input image, process it and classify it under certain categories. For example, giving a trained CNN an image of a dog as input will result in the highest value in the output layer being a dog if trained correctly. Computers see an input image as an array of pixels.

CNNs use supervised learning for classification. In 2013, Sathya, and Abraham, stated that supervised learning assumes the availability of a teacher who classifies the training examples into classes. Supervised learning is when we want to map input data to output data. For example, inputting an image of a dog to a CNN will hopefully result in the output prediction of a dog. Sathya, and Abraham, also describe unsupervised learning as a learning model that learns through trial and error. It is primarily used when you wish to learn the inherent structure of the data without using provided labels.

3.1 Convolution Layer

The first layer is the Convolution layer which extracts features from an input image. Unlike neural networks, where the input is a vector, the input for a CNN is a multi-channelled image i.e. an RGB image. Using small squares of input data the layer preserves relationships between pixels. The Convolution layer can perform filter operations such as edge detection and blur by applying filters.

Stride, also called a kernel, is the number of pixels that are shifted over an input. For example when the stride is 2 then move the filters 2 pixels at a time. When the stride is 3 then move the filters 3 pixels and so on. Figure 3.1 is an example of Stride. With a filter of 3 by 3 and an input of 7 by 7, Stride moves the filter by 2.

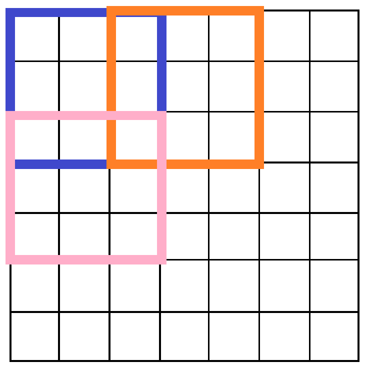


Figure 3.1: 7x7 Input with a filter of 3x3 and a Stride of 2.

When the filter does not fit the input image correctly, padding with zeros can be applied around the border, this is known as zero-padding. For example, figure 3.2 demonstrates when a 3 by 3 filter is applied to an input of 4 by 4, the output volume would be a 4 by 4. The input dimensions decreased due to the filter size but normally you want to preserve as much information about the original input as possible so we can continue to extract features. To preserve features apply a zero padding of size 1 and a stride of 1, resulting in an input volume of 6 by 6 and an output of 4 by 4.

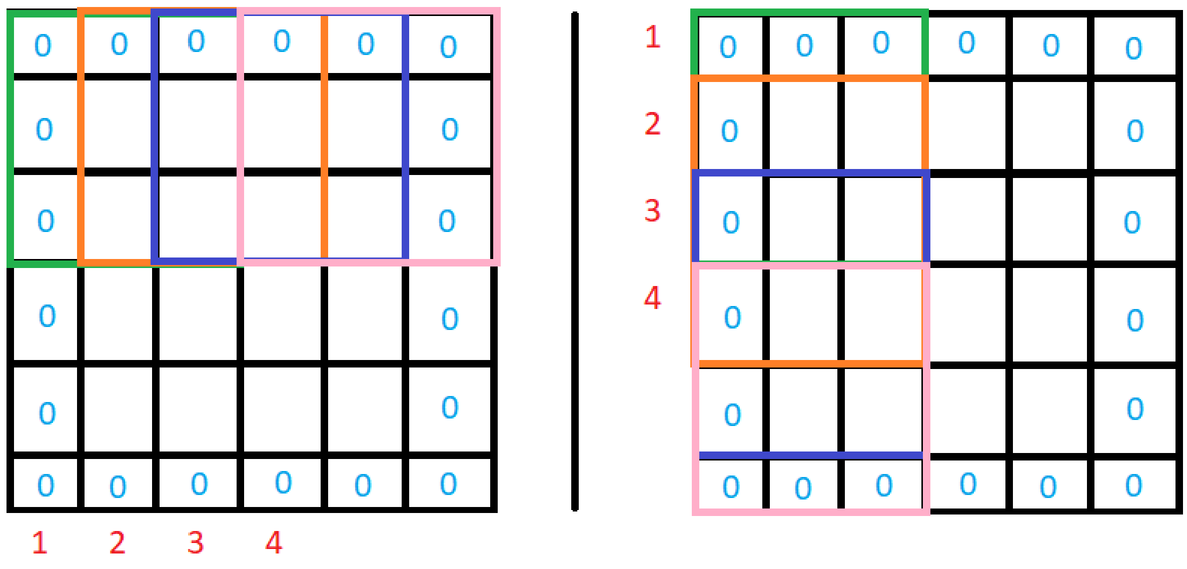


Figure 3.2: Zero padding on a 4 by 4 input, with a 3 by 3 filter, resulting in a 4 by 4 output.

3.2 Activation Layer

After each convolutional layer, a nonlinear layer, also known as an activation layer, is applied. The purpose of this layer is to introduce nonlinearity into the CNN network. Without an activation layer, the output would be a simple linear function. Linear equations are limited in their complexity preventing it from learning complex functional mappings of data. Rectified linear unit (ReLU) is the most common nonlinear function because of its speed and accuracy in comparison to other activation functions. Figure 3.3 demonstrates how ReLU layers essentially changes all negative activations to 0. This is necessary so a CNN can emulate real-world data.

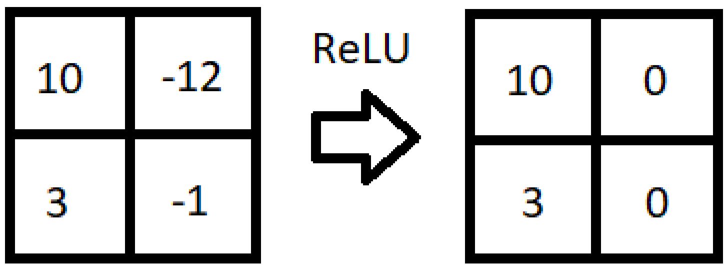


Figure 3.3: ReLU Activation.

3.3 Pooling Layer

In the Pooling layer, images that are too large are reduced by removing the number of parameters and computation in the network. Downsampling also called Spatial pooling or subsampling reduces the dimensionality of each map while retaining the important information. Max, Average and Sum Pooling are all different types of downsampling. Max Pooling, which is the most common approach, takes the largest element from a feature map. Average pooling takes the average element and Sum pooling is the sum of all the elements in the map.

3.4 Fully Connected Layer

In the Fully Connected (FC) layer, the matrix is flattened into a vector and fed into a fully connected layer like a neural network. Neurons in the fully connected layer have full connections to all activations in the previous layer. With fully connected layers, features are combined together to create a model. Finally, an activation function is used such as Softmax to classify the outputs.

The Softmax function takes a vector of random real-valued scores and transforms it into a vector of values between zero and one, that sum to one.

3.5 Keras

Keras is a high-level neural networks API, written in Python. Python is an object-oriented, high-level programming language with dynamic semantics (Python.org. 2019). Keras is capable of running on top of TensorFlow, CNTK, or Theano. Keras allows for the rapid development of a project due to its user-friendliness, modularity and extensibility (Keras.io. 2019). Keras developed an API with humans in mind. It reduces the numbers of user actions required for common use cases in comparison to using just tensorflow. Keras allows for modules to be plugged together with little restrictions, for example, neural layers and activation functions. Keras works with Python, Python offers a range of libraries especially image processing libraries such as OpenCV. OpenCV will allow for CAPTCHA image pre-processing and post-processing. Past research papers such as "An Approach for Chinese Character Captcha Recognition Using CNN" by Yang, et al., 2018 have used Keras to build their CNN for CAPTCHA character recognition and achieved high success rates.

3.6 Conclusion

Convolutional Neural Network architecture is complex, but it is a popular deep learning technique for automated CAPTCHA resolution. Like all deep learning techniques, it is dependent on the quality and amount of data provided. Once supplied with a high-quality dataset, CNN's are capable of rapid visual recognition. CNN’s can be applied to recognising characters, thus resolve CAPTCHAs. Section 4 is going to discuss how to gather and process the required dataset as well as the structure of the CNN.

4.0 Methodology

4.1 CAPTCHA Gathering

The three CAPTCHAs that will be examined are Really Simple CAPTCHA, CAPTCHA 0.3 and Pastebin CAPTCHA, due to their online popularity. Really Simple CAPTCHA has currently over 900,000 installations and is developed with Personal Home Page (PHP) (WordPress.org. 2018). CAPTCHA 0.3 is developed in Python and is used frequently in Python backend applications with over 180,000 downloads (Packaging.python.org., 2019). Pastebin is an online content hosting service that allows users to store snippets of text. It is mainly used to store code, instant messaging chat logs or for malicious intent such as Malware (Recorded Future., 2016). From October 2018 to March 2019, Pastebin has had over 33,000,000 million pages (Similarweb.com., 2019). Figure 4.1 is an example of each CAPTCHA.

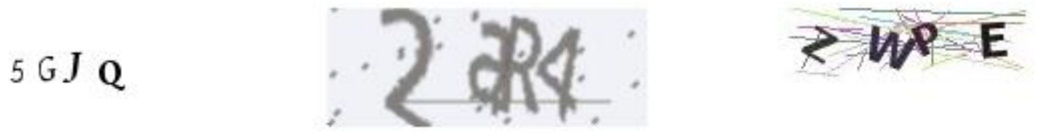


Figure 4.1: Really Simple CAPTCHA, CAPTCHA 0.3 and Pastebin CAPTCHA.

The first step is gathering training and testing sets for each CAPTCHA for a CNN model. Once trained this will allow a CNN to recognise a CAPTCHA character. A guideline within the data science community is to split the data between 80% for training and 20% for testing. With less training, data estimates have greater variance. While with less testing data, performance statistics will have more variance. There is no direct answer for the amount of data required to develop a CNN. The approach used for estimating the required amount of CAPTCHAs was to look at past research on CAPTCHA character recognition and estimate based on the respective data set sizes. Chellapilla and Simard hand labelled 2500 CAPTCHAs, 1600 for training, 200 for validation, 200 for testing and 500 for testing segmentation. Sivakorn, et al. manually labelled 3,000 images for training and tested their CAPTCHA breaking system on 2,235 CAPTCHAs.

Table 4.1 shows the data set for each CAPTCHA. For Really Simple CAPTCHA, 1,500 CAPTCHA images were generated for training and 500 for testing. From an analysis of CAPTCHA 0.3, a larger dataset would be required due to its difficulty, therefore 3,000 CAPTCHA images were generated for the training set and 500 for the testing set. Finally, Pastebin CAPTCHA's were scraped using a Python script and manually labelled, 500 of those images were used for training and 100 for testing. Pastebin has no issues with scraping its content as per its scraping policy.

CAPTCHA	**Training**	**Testing**
Really Simple CAPTCHA	1,500	500
CAPTCHA 0.3	3,000	500
Pastebin	500	100

*Table 4.1**:** Training and Testing set for CAPTCHAs.*

Determining how much training data is required, appears to be a trial and error approach from previous research papers. CNN training produces accuracy reports, which can be utilised to determine if more data is necessary or not. If the training loss is significantly lower than the testing error, the model accuracy will improve with more data. Otherwise, if both error values are about the same value, additional data will not help the model. The addition of more neurons should increase testing error and more data will then help at that point. With CAPTCHA it is difficult to produce 100% perfect training data since the CAPTCHA image goes through multiple processes to hopefully produce a clean output. Therefore, it is important that the inputted data is validated before being processed by the model.

4.2 CAPTCHA Processing

Before development, an analysis was conducted to attempt to recognise patterns within each CAPTCHA. Through analysis, it was discovered that each CAPTCHA had the same length. This is a security flaw due to automated software being able to validate its output. The length and font size should be randomised (Mahato, et al., 2015). The characters ‘1’, ‘L/l’, ‘J/j’, ‘I/i’, ‘0’ and ‘O’ were excluded when the CAPTCHA images were generated as discussed in section 2.4. Really Simple CAPTCHAs generation script was not modified, therefore the letter ‘J’ was included in the generation of its CAPTCHAs, replicating real world CAPTCHA examples. Every CAPTCHA image is converted to greyscale and then image processing techniques are applied. The reasoning for this is because, in a coloured image, each pixel has three colours, with each pixel colour value ranging from 0-255, representing RBG. By converting to greyscale it results in less computational intensity and increases performance. The goal behind CAPTCHA processing is to remove noise within the CAPTCHA image without distorting the characters which produce more accurate segmentation results and therefore increases character classification results.

4.2.1 Really Simple CAPTCHA

Arguably the "simplest" out of the three CAPTCHAs examined. The only anti-recognition technique used is the bolding of random characters. Originally a threshold of 120 was used but this resulted in character splitting. Thresholding is an image processing technique where each pixel within an image is replaced with black if the pixel intensity is less than some fixed constant *T* or white if the pixel intensity is greater than constant *T*. Split characters make it more difficult for CAPTCHA segmentation and will be discussed in depth in section 4.3. Therefore, the threshold amount was increased to 230 which resulted in keeping the characters intact. The threshold was then inverted so that black pixels became white and white pixels became black for segmentation processing. Figure 4.2 demonstrates these steps.

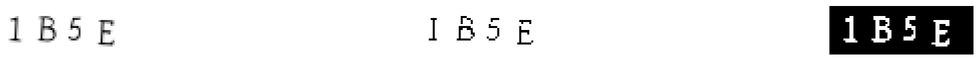
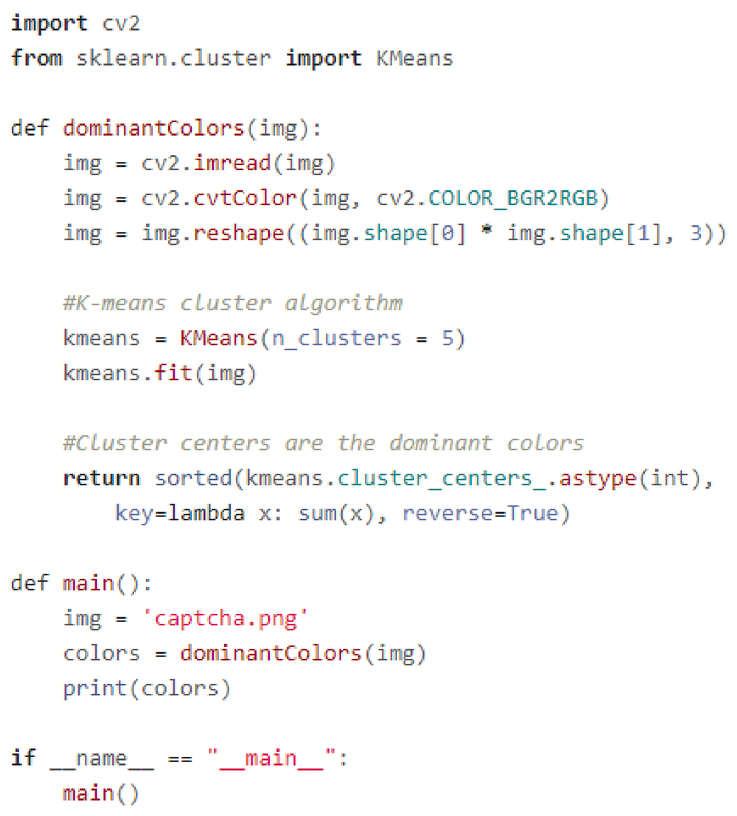


Figure 4.2: Really Simple CAPTCHA after each stage of thresholding.

4.2.2 Pastebin

Pastebin CAPTCHA contains random narrow coloured lines in the background. Using colour in a CAPTCHA image can be exploited and due to Pastebin CAPTCHA utilising colour, this was tested for exploitation. The exploitation method used was most dominant colours. This method worked by using a clustering algorithm known as K-means. K-means is a type of unsupervised learning. This algorithm works by finding groups in the data, with the number of groups represented by a variable. The algorithm will then cluster the pixel intensities of a coloured CAPTCHA image. Figure 4.3 provides the K-means implementation code.



```python
import cv2
from sklearn.cluster import KMeans

def dominantColors(img):
    img = cv2.imread(img)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img = img.reshape((img.shape[0] * img.shape[1], 3))

    #K-means cluster algorithm
    kmeans = KMeans(n_clusters = 5)
    kmeans.fit(img)

    #Cluster centers are the dominant colors
    return sorted(kmeans.cluster_centers_.astype(int),
        key=lambda x: sum(x), reverse=True)

def main():
    img = 'captcha.png'
    colors = dominantColors(img)
    print(colors)

if __name__ == "__main__":
    main()
```

Figure 4.3: K-means cluster concept code.

Unfortunately, this resulted in inaccurate results due to the random coloured background lines, tampering pixel intensities and small character sizes. This method was attempted on another real-world CAPTCHA, Superbuy CAPTCHA, to prove its concept as there was a lack of, or possibly no, previous research papers using this method. Using the method resulted in returning each individual characters red, green, blue (RGB). Once each RGB was returned a mask can be created to extract those specific colours from the CAPTCHA image, resulting in each individual character with little to no damage to the character itself, producing a perfectly cleaned CAPTCHA. Figure 4.4 is an example of Superbuy CAPTCHA and CAPTCHA 0.3 is a great example of a CAPTCHA that prevents most dominant colour exploitation by using the same colour throughout.

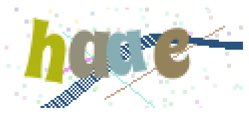
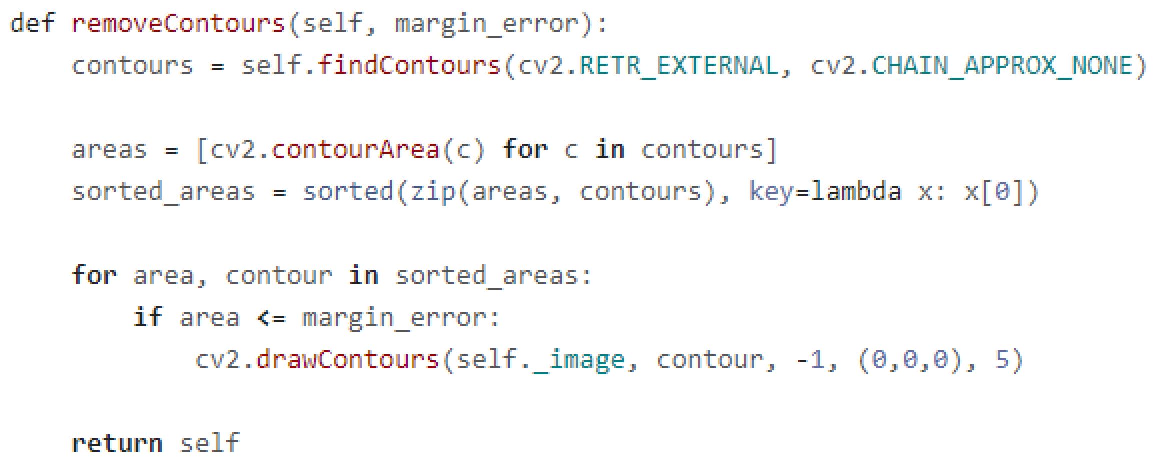


Figure 4.4: Superbuy CAPTCHA.

After further examination of Pastebin CAPTCHA, it was discovered that a threshold of 100 removed the majority of background lines due to their intensity but sometimes tiny pixels would remain. FindContours is a function provided by Opencv which detects all contours in a supplied image and returns their coordinates. A contour can be described as a curve that joins all the continuous points and has the same colour or intensity. FindContours was used to locate all the individual tiny pixels in the processed CAPTCHA image. Alternatively, ConnectedComponents, another OpenCV function, could have been used but FindContours was prefered due to the ease of calling DrawContours after detecting a contour to fill it. For each of those discovered contours, their area is calculated to order them by size. Finally, if the contour area size is less than a given value it is filled with the colour black, essentially blending it into the background. Figure 4.5 provides the code implementation for dots/contours removal.



```
def removeContours(self, margin_error):
    contours = self.findContours(cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)

    areas = [cv2.contourArea(c) for c in contours]
    sorted_areas = sorted(zip(areas, contours), key=lambda x: x[0])

    for area, contour in sorted_areas:
        if area <= margin_error:
            cv2.drawContours(self._image, contour, -1, (0,0,0), 5)

    return self
```

Figure 4.5: Dot removal code.

Once the tiny dots were removed, the image was dilated to increase the character size. Dilation works by increases the white region in an image (Docs.opencv.org., 2019). If the characters were close together, dilation would not be used, since it has the possibility of joining characters together. Figure 4.6 demonstrates the tiny dot removal and dilation process for Pastebin CAPTCHA.

Figure 4.6: Pastebin CAPTCHA before and after dot removal with dilation.

4.2.3 CAPTCHA 0.3

Finally, CAPTCHA 0.3, the most difficult CAPTCHA to resolve in comparison to the other CAPTCHAs. Depending on the randomness of the CAPTCHA generation, it can even be difficult for a human to resolve, which contradicts the definition of CAPTCHA. Figure 4.7 shows an example of its difficulty.

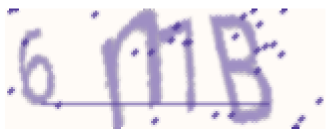
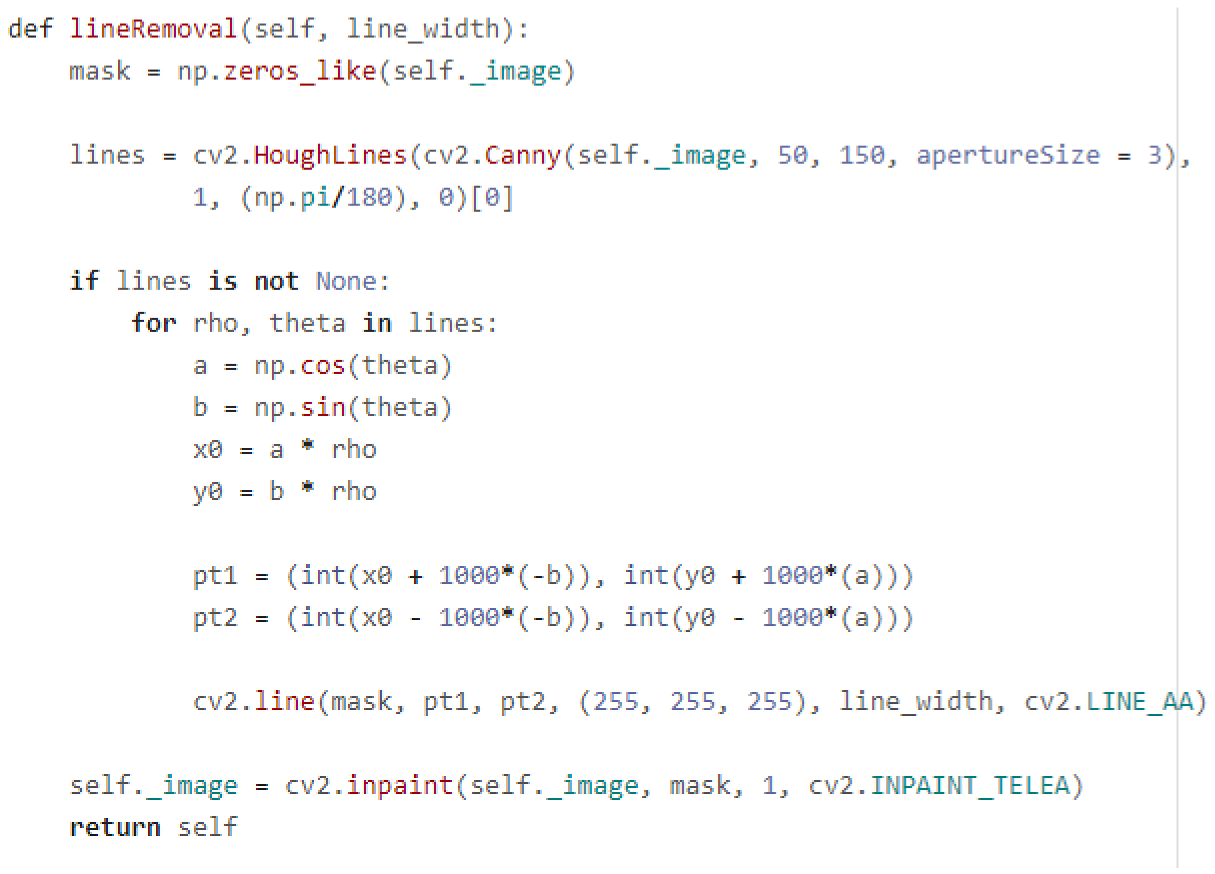


Figure 4.7: Difficult CAPTCHA 0.3, answer 6N7B.

The first step was to remove the single random line goes through at least one if not all characters in the CAPTCHA. Erosion, an image processing technique that works by decreasing white regions/pixels in an image, was attempted as a solution to remove both the line and dots, but unfortunately, the amount of erosion necessary would erode make characters unrecognisable (Docs.opencv.org., 2019). This CAPTCHA needed a somewhat different approach for noise removal. After additional research, Hough Transform was discovered. Hough Transform is used to extract features, only if the feature can be represented mathematically, even if it is broken or distorted. To make use of Hough Transform, edges within an image need to be detected. Edge detection helps reduce the number of pixels to process and maintains the structure of the image. Opencv has a built-in function to detect edges easily called “Canny”. These edges are then used by Hough Lines, an altered Hough Transform function built by Opencv to detect straight lines in an image. To understand how Hough Lines works is, mathematically a line can be represented as $y = m(x) + b$ or in parametric form as $p = x \cos \theta + y \sin \theta$. The perpendicular distance from the origin to the line is p and the angle formed by the perpendicular line and horizontal axis is θ. θ is defined by a function parameter, normally *math.pi / 180.0*. This will compute p a total of 180 times. For example, imagine a horizontal line in the middle of an image, take the first point (x, y) in the line, detected by Canny and calculate p. For every (p, θ) pair, increases its accumulator value by one, making the accumulator at, for example (100, 90) equal to 1. Continue this process for each point on the line and once completed, the pair (100, 90) should have the highest accumulation. Meaning there is a line at distance 100, 90 degrees from the origin. Once the lines are detected within the image they are filled using Inpainting, an Opencv function. Inpaint works essentially by filling specified pixels with neighbouring pixels. Figure 4.8 shows the implementation of the line removal code and figure 4.9 demonstrates the line removal before and after.



```
def lineRemoval(self, line_width):
    mask = np.zeros_like(self._image)

    lines = cv2.HoughLines(cv2.Canny(self._image, 50, 150, apertureSize = 3),
            1, (np.pi/180), 0)[0]

    if lines is not None:
        for rho, theta in lines:
            a = np.cos(theta)
            b = np.sin(theta)
            x0 = a * rho
            y0 = b * rho

            pt1 = (int(x0 + 1000*(-b)), int(y0 + 1000*(a)))
            pt2 = (int(x0 - 1000*(-b)), int(y0 - 1000*(a)))

            cv2.line(mask, pt1, pt2, (255, 255, 255), line_width, cv2.LINE_AA)

    self._image = cv2.inpaint(self._image, mask, 1, cv2.INPAINT_TELEA)
    return self
```

Figure 4.8: Line Detection and Removal code.

Figure 4.9: CAPTCHA 0.3 before and after line removal.

Secondly, the CAPTCHA image was converted to greyscale and a threshold of 160 was applied but this led to problems. It was discovered that random opacity was utilised. Opacity can be an additional security feature when generating a CAPTCHA image to prevent automated recognition since it prevents a fixed threshold amount. This caused some CAPTCHAS to return just the noise, which in theory could be used to extract the characters from the CAPTCHA image but it would not be feasible as it would only work for a small majority of the CAPTCHA images. Figure 4.10 is an example of two different CAPTCHA 0.3s with 160 thresholding.

Figure 4.10: Example of two CAPTCHA 0.3s with 160 thresholds.

Another approach attempted was Histogram Equalization, an image processing technique that equalizes the histogram of an image. An image histogram quantifies the number of pixels in an image for each intensity value encountered and then produces a graphic representation of the intensity distribution. Histogram Equalization stretches out the intensity range from the histogram intensity distribution by modifying the contract in the image. Figure 4.11 shows Histogram Equalization applied to CAPTCHA 0.3

Figure 4.11: Example of CAPTCHA 0.3 before and after Histogram Equalization.

Histogram Equalization resulted in removing the opacity security feature but made the characters within the CAPTCHA image bulkier. This was not an ideal solution as it conjoined characters, increased the difficulty in removing the dots and sometimes the dots modified the shape of the characters making it difficult for the CNN to classify the character.

Otsu and Adaptive thresholding were another two thresholding types discovered. With default thresholding, an arbitrary value is used but with Otsu thresholding, it automatically calculates a threshold value. The calculation uses a bimodal image. A bimodal image in simple terms is an image whose histogram has two peaks. The calculation then takes the middle value of those peaks and applies them as a threshold value. Figure 4.12 demonstrates Otsu thresholding. It bypassed the opacity anti recognition technique 99.99% of the time with 0.01% returning a blank CAPTCHA.

Figure 4.12: CAPTCHA 0.3 before and after Otsu thresholding.

Adaptive thresholding calculates a different threshold value for different regions within the image, unlike default thresholding where a global threshold value is applied. Adaptive thresholding also bypassed the opacity security technique but at times eroded characters Therefore Otsu thresholding was the chosen method for this CAPTCHA.

The last step was to remove the random dots. The dot removal code used for Pastebin CAPTCHA was utilised here also and was successful. Figure 4.13 displays CAPTCHA 0.3 after all of the applied functions.

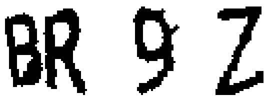


Figure 4.13: CAPTCHA 0.3 after image processing.

4.3 CAPTCHA Segmentation

Once the CAPTCHA images have been pre-processed accordingly they need to be separated into individual letters because training a CNN on just CAPTCHA images would increase the software's difficulty significantly because it would cost more computational power and a larger training/testing dataset and from reviewing previous research papers it is the most common approach (Wang, et al., 2017). The segmentation process works by detecting individual chunks of pixels within a CAPTCHA image using the FindContours function. It is similar to the removal of dots method mentioned in section 4.2.2 but it will only return the specified amount of largest contours. The specified amount is a variable equal to the maximum CAPTCHA length and the largest means that the contours are sorted by their amount of white space. A perfectly preprocessed image will have only the maximum CAPTCHA length of contours but sometimes not all noise is removed or characters are conjoined. Therefore, the largest contours are explicitly retrieved because CAPTCHA characters will more than likely be the largest chunks of pixels within the image. Once the contours are returned their aspect ratio is calculated. Aspect ratio is the ratio of the width to the height. The ratio helps to determine if the contour is one character or more. If the ratio is greater than x it is a high indication that the contour has more than one character within it. Originally x was a static value, but this required constant code changes via a trial and error approach to determine a good separation ratio and update for a new CAPTCHA. Therefore, the software has implemented an automated calculation of x, where x is the smallest outlier aspect ratio that is calculated after gathering all the contour aspect ratios for each training CAPTCHA image. Using the smallest outlier aspect ratio instead of the mean resulted in more accurately segmented images by approximately 18%. Outliers are discovered using a z-score test with a score of greater than 3. A z-score is used primarily in statistics to determine the number of standard deviations a number is from the mean (Hayes, 2019). Thus, the contour is split in half and stored. Figure 4.14 demonstrates this process. Once contour identification is complete, the contour list is validated to ensure that the list has between the minimum and maximum CAPTCHA length to prevent invalid training data. If the software is run to process CAPTCHA images and not predict a CAPTCHA, then the contours are saved to their corresponding letter folder.

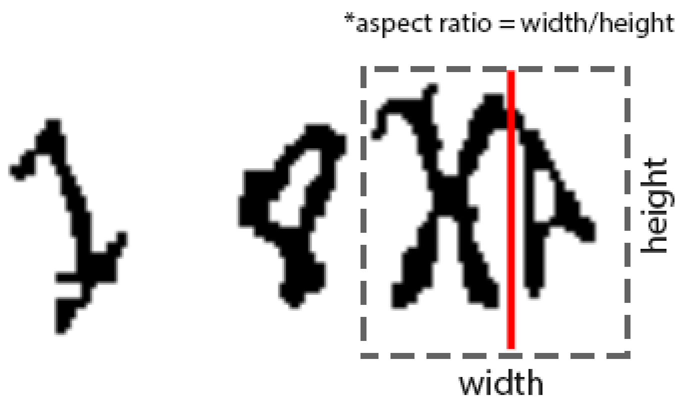


Figure 4.14: CAPTCHA segmentation based on aspect ratio.

4.4 CNN Development

Image processing is essential for accurate classifications. Without clean data, a CNN would have difficulty identify characters within a CAPTCHA. The CNN is trained on thousands of processed character images relevant to its CAPTCHA. The first step to developing the CNN was to specify the testing ratio. The testing ratio for each CNN is 20% as discussed in section 4.1. All character images have a fixed size of 28 x 28. This CNN does not require large image imports, resulting in faster training and less computational power. A sequential model which is a pre-built model by Keras is used to build the CNN. It allows for the linear addition of layers to the CNN. First, the convolutional layers are developed, a 2D convolutional layer is added with 32 outputs, a kernel size of 3 x 3, an input shape i.e. the character image dimensions, the activation function relu and with padding type same. Same padding ensures that the input image gets fully covered by the filter/stride specified. The next layer is a pooling layer that uses max pooling, with the pool size and stride specified. These two layers are repeated except the 2D convolutional layer has twice the amount outputs in comparison to the first. Convolutional layers output produce multidimensional outputs and need to be converted into a one-dimensional output to support the fully connected layers. This process is known as flattening and is applied to the convolutional layers. The following two dense layers are the fully connected layers. A dense layer is a layer of neurons, where each neuron receives input from the previous layer, hence a fully connected layer. A neuron works by taking several inputs, applies an activation function and returns the output (Nielsen, 2015). Figure 4.15 is an example of how a neuron works.

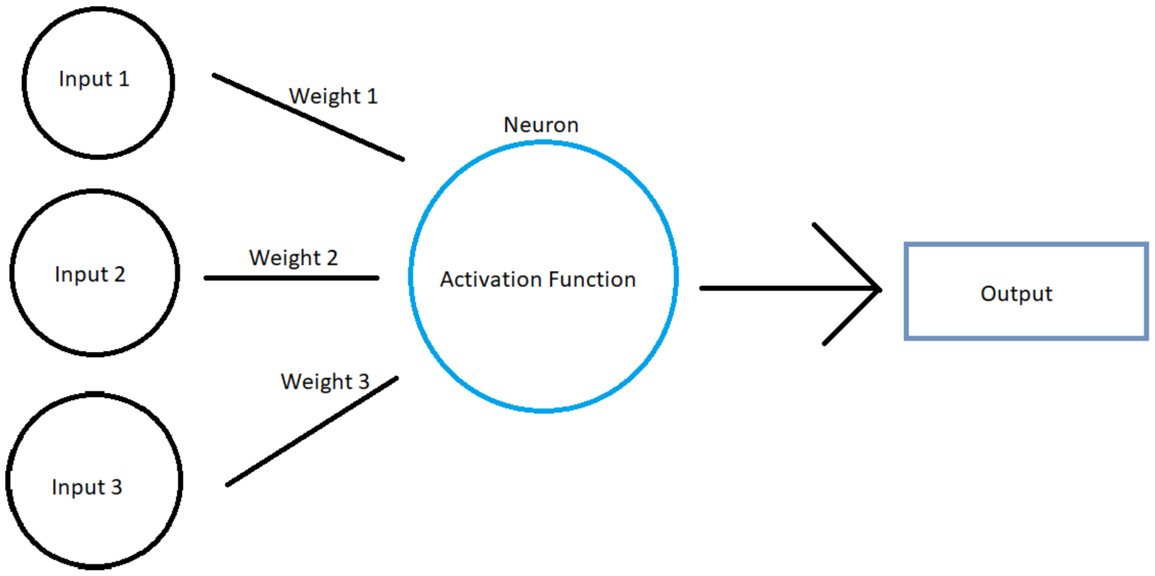


Figure 4.15: Simple CNN neuron example.

The first dense layer has *n* hidden layers, where n is supplied and uses the activation function relu. The second dense layers number of layers is equal to the counts of labels in the model and uses the softmax activation function. The softmax function turns numbers into a probability value between 0 and 1. Next, the model needs to be compiled. The Keras compile function needs three parameters: loss, optimizer and metrics. Categorical cross entropy is the loss type used. The loss function is used to measure inconsistency between a predictive value and the actual label. Entropy tends towards zero as neurons get better at achieving the desired output. Put simply the loss function is a way of measuring how wrong predictions are. The optimizer type used is Adam. Adam stands for Adaptive Moment Estimation. The Adam optimizer updates weights in the CNN by responding to the output from the loss function. The loss function guides the optimizer to produce the most accurate result possible. The last of the compile functions is metrics. The metrics type used is accuracy. Metrics help to evaluate a CNN and is an essential part of any project. Accuracy is the ratio of correct predictions to the total number of input samples. Finally, the model is trained to use the Fit function provided by Keras. It accepts the following parameters: training data, target data, validation data, the number of epochs and batch size. The number of epochs and batches are supplied. Each epoch will make the model iterate over the data, improving the model until a certain point. The batch size defines the number of samples that will be propagated through the CNN network, for example, it will take 6 character images at a time when training the model. Figure 4.16 displays the structure for CNN, with the fully connected layer prediction an inputted CAPTCHA character.

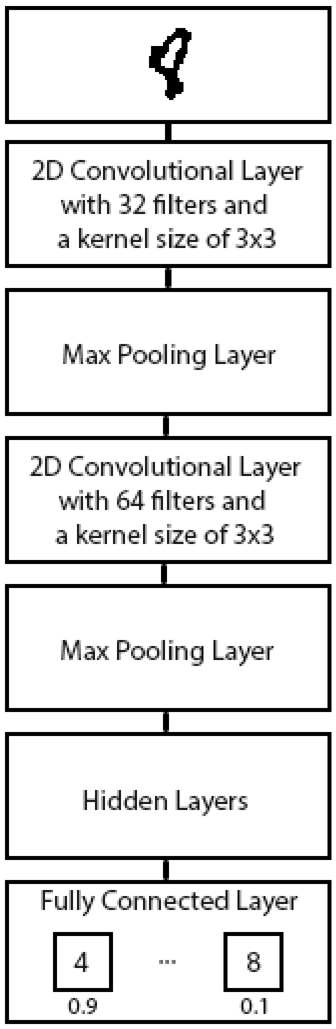

2D Convolutional Layer with 32 filters and a kernel size of 3x3
Max Pooling Layer
2D Convolutional Layer with 64 filters and a kernel size of 3x3
Max Pooling Layer
Hidden Layers
Fully Connected Layer
4
...
8
0.9
0.1

Figure 4.16: Structure of the Convolutional Neural Network.

4.5 Conclusion

Years of research has been conducted in CAPTCHA generation and breaking and from this chapter, it should be evident that CAPTCHA breaking is a complicated and tedious process. The lack of datasets publicly available for CAPTCHAs makes it difficult to develop machine learning models. Removing noise from a CAPTCHA image requires a trial and error approach where the user understands how to apply image processing techniques. Each stage of the CAPTCHA breaking process is dependant on one another, for example, segmentation is dependant on a successful noise removal and a CNN model is dependant on a successful segmentation. The order of layers in which a CNN can be developed is never ending but past research papers provide a good guideline to go by. The next chapter will discuss the overall architecture, requirements and testing for the CAPTCHA breaking system.

5.0 Software Report

5.1 Agile Development

Agile Methodology is a project management approach, primarily used for software development in large organisations. Agile methods provide a structured development process using incremental, iterative work sequences that are known as sprints, where the solutions and requirements evolve through collaboration between self-organising cross-functional in teams. There are multiple types of agile methods in existence such as Scrum and Extreme Programming (XP), where both of those methods focus on situations where development is done in pairs of small teams. As this project is an individual project, the chosen agile method was Personal Extreme Programming (PXP). PXP modifies XP practices i.e. communication, feedback, simplicity and courage, so they can work in a single programmer situation (Agarwal, and Umphress, 2008). Throughout the project, feedback was provided through weekly supervisor meetings. During these meetings, new functionality and progress was demonstrated and feedback was given. Feedback was then applied to the project through an iterative and incremental approach.

5.2 Analysis and Requirements

Past research papers provided insight into multiple ways to classify CAPTCHA characters and guidelines on how to successfully develop a CAPTCHA character classification system. For the functional and non-functional requirements the technique FURPS+ is used to validate and prioritise requirements. The FURPS+ acronym stands for Functionality, Usability, Reliability, Performance, Supportability and the “+” specifies constraints such as design, interface, physical and implementation (Huda, 2018).

5.2.1 Functional Requirements

Functional requirements describe the behaviour of the system when certain conditions are met. Simply put, it specifies something the system should do. Users of the system are provided with the following functionality:

- Create multiple CAPTCHA setups.
- Create/Update the CNN model specifications.
- Process CAPTCHA images.
- Develop multiple CNN model.
- Predict a CAPTCHA image output.

5.2.2 Non-functional Requirements

Non-functional requirements describe how the system should behave and the limits on its functionality. In simple terms, a requirement that specifies what the system should do.

5.2.2.1 Usability

Usability is requirements based on the user interface of the system. The user interface should be developer friendly. Help functions need to be provided via the command line.

5.2.2.2 Performance

Performance is the systems, start-up, setup and response time. Initially, the system is going to require setup for a new CAPTCHA but once setup is completed, response times should be rapid.

5.2.2.3 Reliability

The system should prevent the application from crashing. Therefore, error handling should be in place to prevent crashing and explain the cause.

5.2.2.4 Supportability

The system must implement testing frameworks to ensure the system is defect free. Pythons unit test framework will be used for testing.

5.2.2.5 Implementation

Implementation involves constraints that limit the development of the system. The system will be written entirely in Python and configuration files, excluding CNN model outputs, will be in JSON format.

5.2.3 Use Case Documentation

This section will also provide the documentation for the three main features the user has available to them. These features are: processing CAPTCHA images, building CNN models, and predicting a CAPTCHA image. Figure 5.1 illustrates these features using a use case diagram.

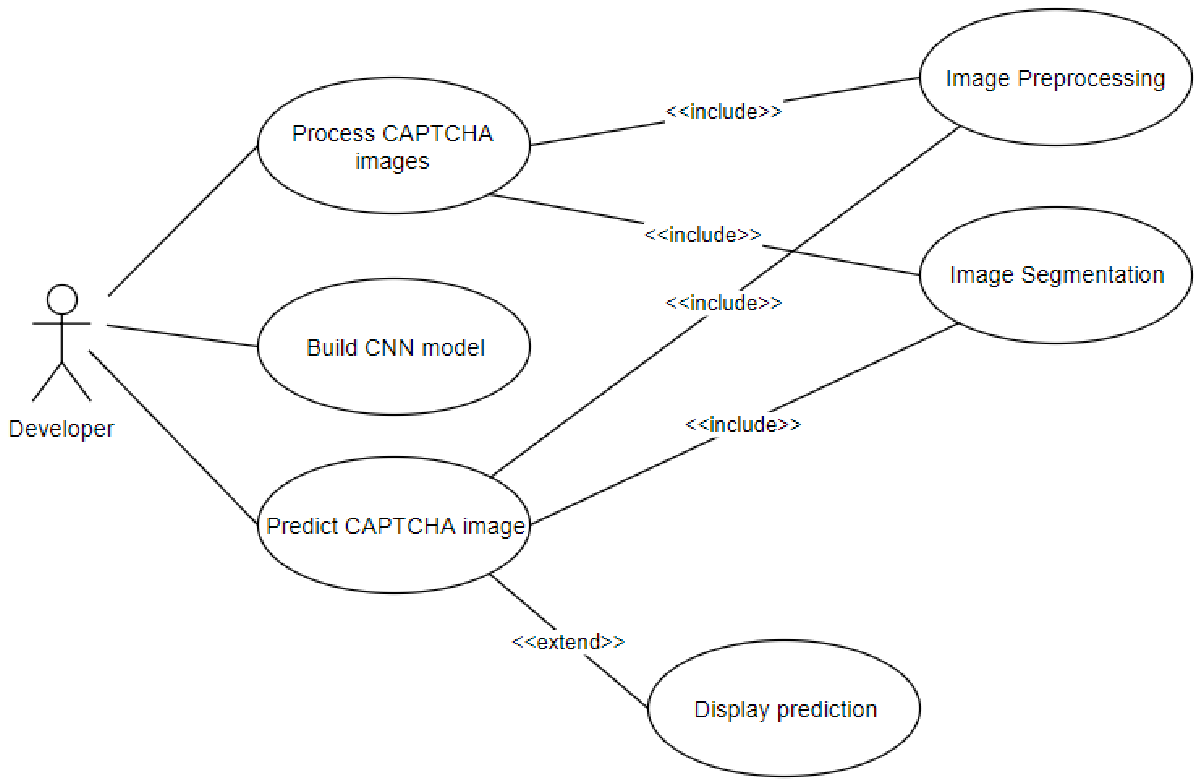


Figure 5.1: Use case diagram of the system.

Use Case Name: Process CAPTCHA images.

Participating Actor: Developer.

Entry Conditions:

- A developer wants to process images of a CAPTCHA.
- The user navigates to the command line, enters the argument “--config” along with the CAPTCHA configuration filename and the argument “--imageprocessing”.
- The “Process CAPTCHA images” function of the system has been invoked.

Flow of Events:

- The system responds by displaying a progress screen until all CAPTCHA images specific to the configuration are processed.
- The developer confirms.

Exit Conditions:

- All CAPTCHA images specific to the configuration are processed.

Use Case Name: Build a CNN model.

Participating Actor: Developer.

Entry Conditions:

- A developer wants to build a CNN specific to a CAPTCHA.
- The user navigates to the command line, enters the argument “--config” along with the CAPTCHA configuration filename and the argument “--build”.
- The “Build a CNN model” function of the system has been invoked.

Flow of Events:

- The system responds by developing a CAPTCHA specific CNN model and displaying its progress screen.
- The developer confirms.

Exit Conditions:

- A CNN model is built for a specific CAPTCHA.

Use Case Name: Predict CAPTCHA image.

Participating Actor: Developer

Entry Conditions:

- A developer wants to predict a CAPTCHA.
- The user navigates to the command line, enters the argument "--config" along with the CAPTCHA configuration filename and the argument "--predict" along with the image in base64 format.
- The "Predict CAPTCHA image" function of the system has been invoked.

Flow of Events:

- The system responds by displaying its CAPTCHA prediction.
    - If the prediction was unsuccessful it outputs "FAILED".
- The developer confirms.

Exit Conditions:

- A prediction is made to resolve a CAPTCHA.

5.3 Architecture and Design Patterns

This section provides a high-level overview of the system architecture and design. The system follows three key software development principles. The first principle is the mnemonic acronym SOLID, which stands for: Single responsibility principle, Open/closed principle, Liskov substitution principle, Interface segregation principle, and Dependency inversion principle. The SOLID principles allow for simple, robust and updatable code. The second principle followed is Don’t Repeat Yourself (DRY). DRYs main goal is to avoid repetitive code and in return will again allow for more robust and clean code. Code that needs to be updated will likely only need to be updated at a single point. The final principle is “You ain't gonna need it” (YAGNI). YAGNI states a developer should not add functionality until it is necessary. Therefore this system provides what the system needs and nothing more.

Another key aspect of the system is the need for modularity due to the fact each CAPTCHA has a unique generation method. This adds additional complexity to the system but it is achieved through modular code, configuration files in JavaScript Object Notation (JSON) format and multiple trained CAPTCHA models.

5.3.1 System Description

The system has five main components to it. Figure 5.2 provides a high-level overview of the system. The Parser component is responsible for importing and outputting JSON configuration and model output files. The Image Processing component processes CAPTCHA images via available functions. Once an image is processed, it is outputted into a subdirectory named “/cleaned/”. The Segmentation component uses the processed CAPTCHA images, segments them in individual characters and outputs them into a new subdirectory called “/characters/”. The Model component is responsible for developing CNN models using the character data. Finally, the Predict component is responsible for predicting CAPTCHA images using the trained CNN model.

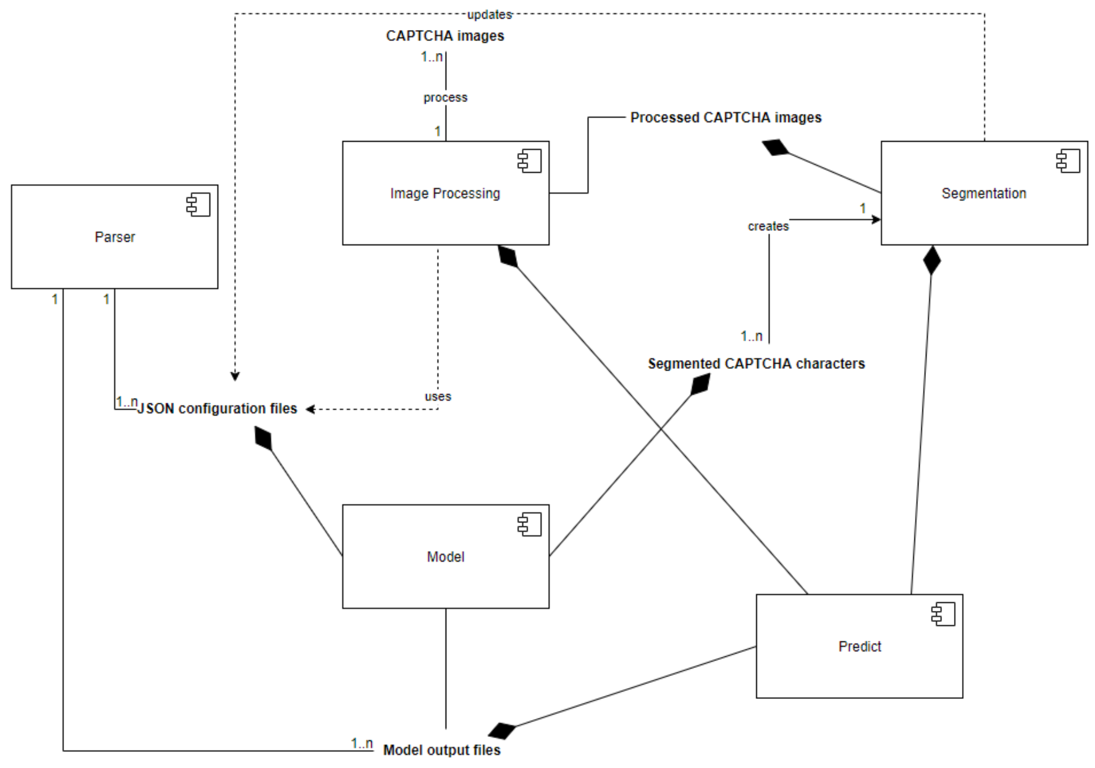


Figure 5.2: System component diagram.

Classes were designed so objects of that class would be built in stages, to reduce complexity. For example, the Model object has its initialisation functions along with three additional functions to build a model. The initialise function setups the models configuration such its import image size and stride. The train function imports and processes character images for the model to train on. Finally, the build function develops a CNN model based on all the provided data. Figure 5.3 shows the classes of the system and their associations.

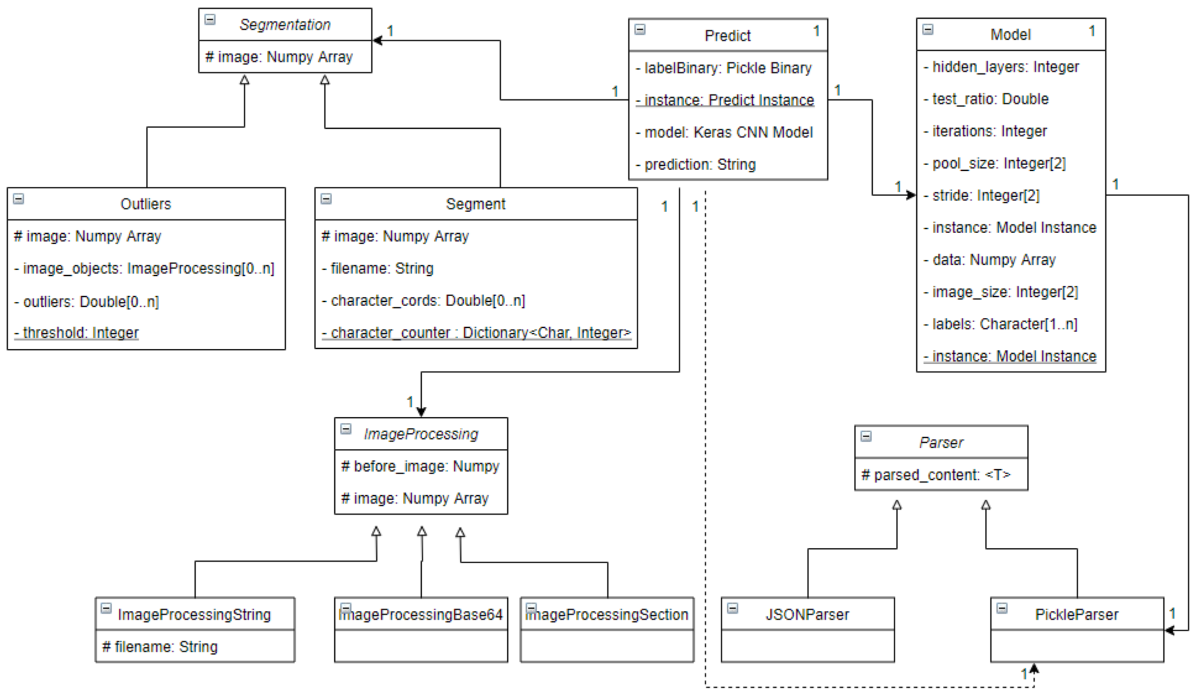


Figure 5.3: Class diagram for the system.

Each class contains specific business logic regarding the system. The purpose of each class is as follows:

- **Segmentation**: Is an abstract class allowing the subclasses, Outliers and Segment to inherit or override its functionality.
    - **Outliers**: Discovers outliers within the CAPTCHA data.
    - **Segment**: Segments a CAPTCHA into individual characters.
- **ImageProcessing**: Is an abstract class allowing the subclasses, ImageProcessingString, ImageProcessingBase64 and ImageProcessingSection to inherit or override its functionality. It processes images using image processing techniques.
    - **ImageProcessingString**: Provides a filename as additional functionality. Image objects are created via a string input i.e a path.

    - **ImageProcessingBase64**: Image objects are created via a base64 string encoded image.
    - **ImageProcessingSection**: Image objects are created via a section of an image.
- **Predict**: Attempts to resolve a CAPTCHA image.
- **Model**: Builds CNN models for CAPTCHAs.
- **Parser**: Is an abstract class allowing the subclasses, JSONParser and PickleParser to inherit or override its functionality.
    - **JSONParser**: Handles imports, outputs and updates for JSON configuration files.
    - **PickleParser**: Handles imports and outputs for Model files.

All of these components work in conjunction with one another to classify CAPTCHA characters. The current system is quite complex. Therefore, design patterns were implemented to hide the complexity and increase system performance.

5.3.2 System File Structure

The system file structure is important because, it is how it determines where to looks for configurations, model and image files. The “data” folder is where these files are stored. Figure 5.4 demonstrates the folder structure. The subdirectory “captchas” is where images relating to a specific CAPTCHA are stored. The “configs” subdirectory has two subdirectories for CAPTCHA JSON configurations and a model configuration. The final subdirectory “models”, is where the Model class outputs CNN trained model files.

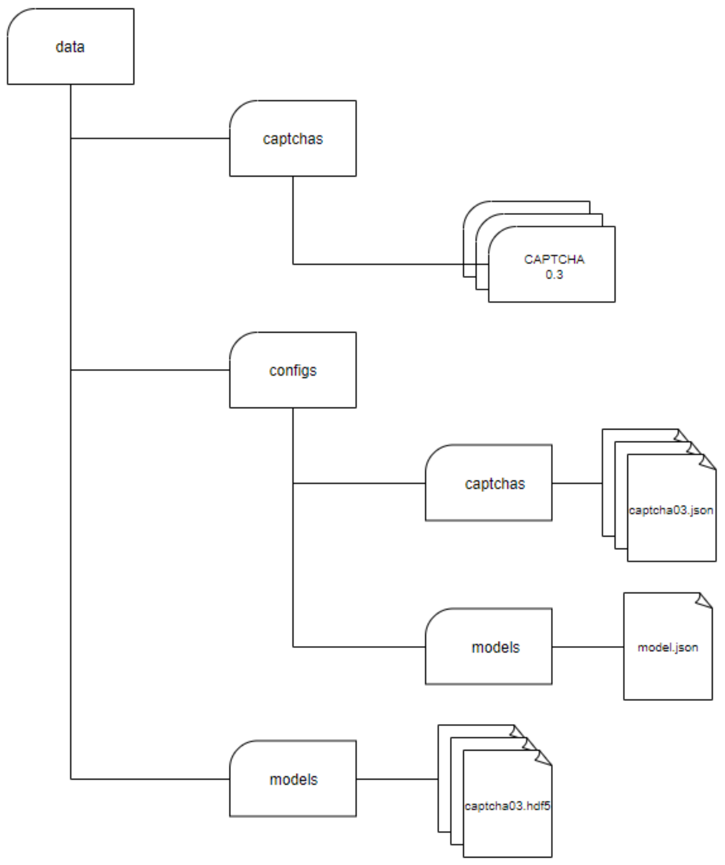


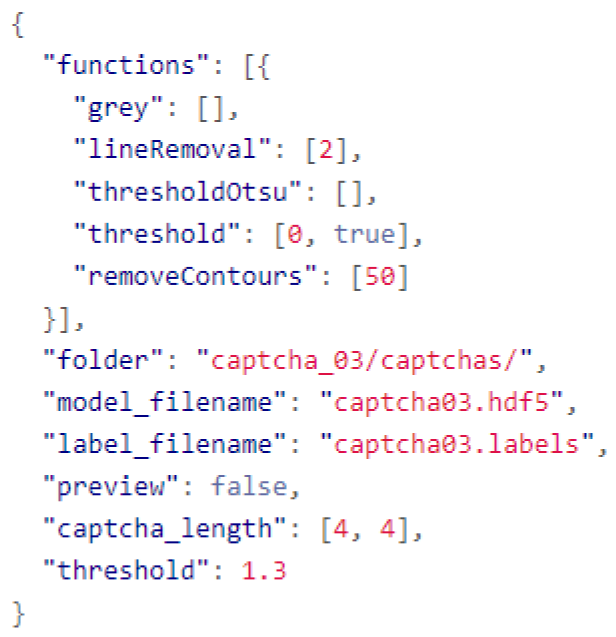
Figure 5.4: The file structure for the data folder.

5.3.3 CAPTCHA Setup

Adding a new CAPTCHA to the system requires a CAPTCHA JSON configuration file, CAPTCHA analysis skills and CAPTCHA images. Figure 5.5 is an example of a CAPTCHA JSON configuration file. The reasoning for each key, value pair, where an underlined key is automatically generated, is the following:

- **functions**: Are functions provided by the ImageProcessing class used to process the CAPTCHA image. The format goes: *"Function Name" : [Parameters]*.

- **folder**: The path where the CAPTCHA images are located. Subdirectories will also be created in this folder.
- **model_filename**: The filename for a trained model.
- **label_filename**: The filename for labels relating to a model.
- **preview**: A boolean to display a CAPTCHA image prediction output.
- **captcha_length**: The minimum and maximum length of a CAPTCHA.
- **threshold**: The aspect ratio threshold generated via the Outliers class to determine if the letter is conjoined.



```
{
  "functions": [{
    "grey": [],
    "lineRemoval": [2],
    "thresholdOtsu": [],
    "threshold": [0, true],
    "removeContours": [50]
  }],
  "folder": "captcha_03/captchas/",
  "model_filename": "captcha03.hdf5",
  "label_filename": "captcha03.labels",
  "preview": false,
  "captcha_length": [4, 4],
  "threshold": 1.3
}
```

Figure 5.5: CAPTCHA JSON configuration file.

Determine what image processing functions are required and in what order to remove noise from a CAPTCHA image requires background knowledge, primarily in image processing techniques. Removing noise from a CAPTCHA image is achieved through a trial and error approach. This process can require a significant amount of time depending on the CAPTCHA. The last necessity is CAPTCHA images. Gathering CAPTCHA images can be achieved through a manual or automatic approach, but their filename needs to be the CAPTCHA resolution for model training and testing purposes.

5.3.4 Factory

A Factory is a creational pattern, that allows the creation of objects without exposing the constructor logic to the client. It defines an interface for the creation of an object but allows derived classes to decide which class to instantiate. It also follows the Single Responsibility Principle because the entire responsibility of object creation is moved to a factory.

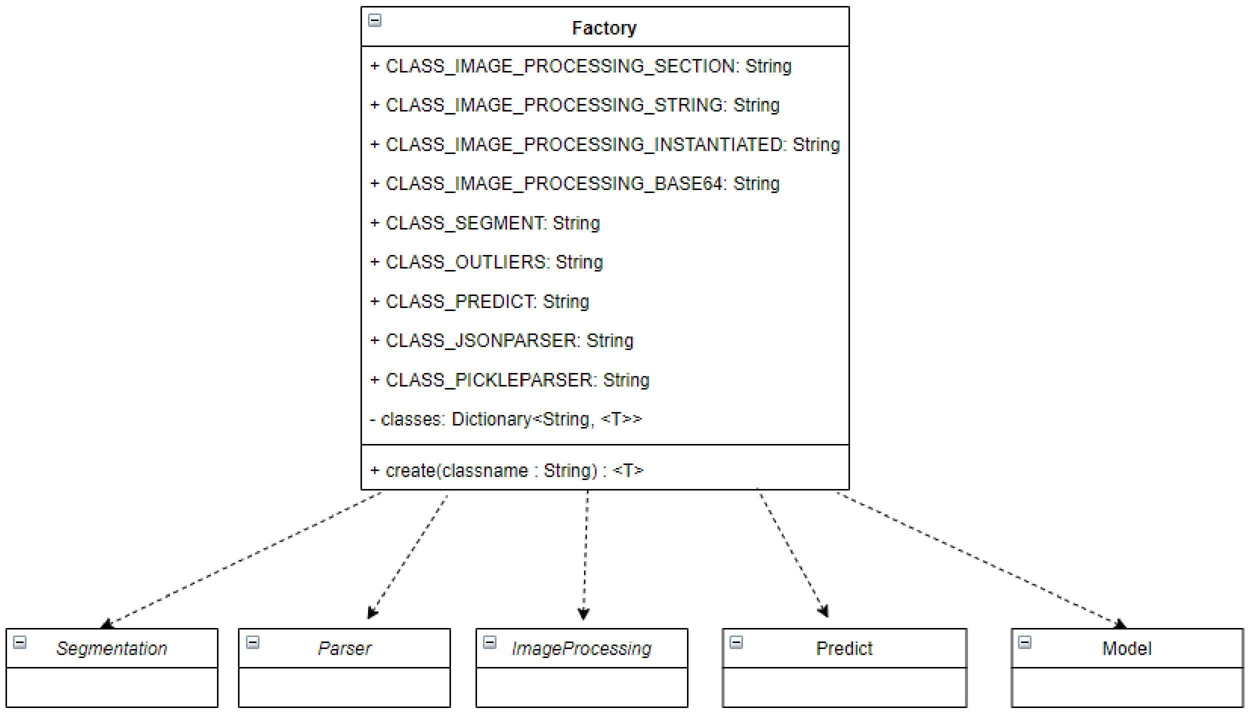

Factory
+ CLASS_IMAGE_PROCESSING_SECTION: String
+ CLASS_IMAGE_PROCESSING_STRING: String
+ CLASS_IMAGE_PROCESSING_INSTANTIATED: String
+ CLASS_IMAGE_PROCESSING_BASE64: String
+ CLASS_SEGMENT: String
+ CLASS_OUTLIERS: String
+ CLASS_PREDICT: String
+ CLASS_JSONPARSER: String
+ CLASS_PICKLEPARSER: String
- classes: Dictionary<String, <T>>
+ create(classname : String) : <T>
Segmentation
Parser
ImageProcessing
Predict
Model

Figure 5.6: Factory design pattern.

5.3.5 Facade

The Facade structural pattern hides the complexities of the system by adding an interface to the system. It involves a single class that delegates calls to system classes while providing simplified methods required by clients. Figure 5.7 demonstrates MarkCAPTCHA, the systems Facade design.

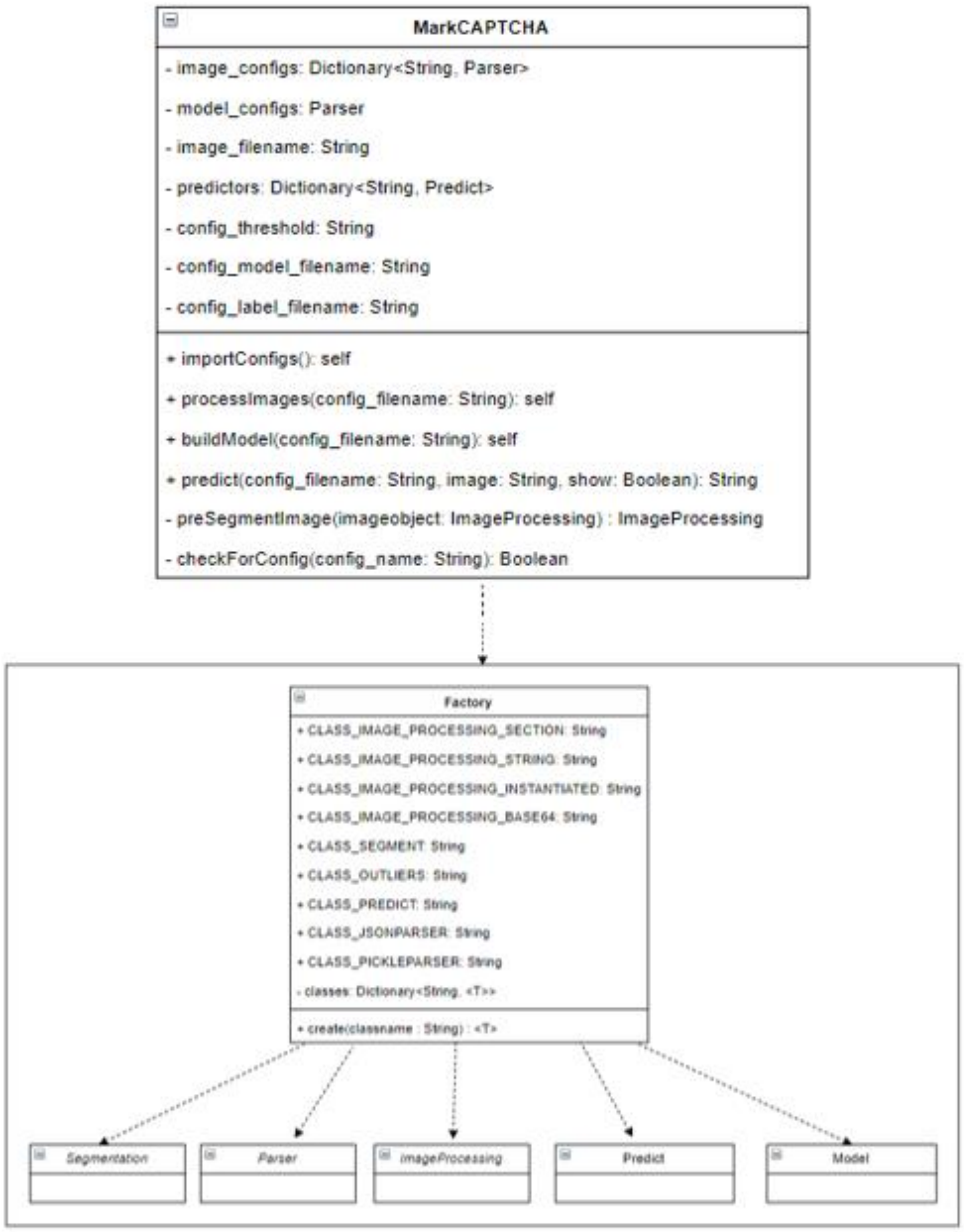

MarkCAPTCHA
- image_configs: Dictionary<String, Parser>
- model_configs: Parser
- image_filename: String
- predictors: Dictionary<String, Predict>
- config_threshold: String
- config_model_filename: String
- config_label_filename: String
+ importConfigs(): self
+ processImages(config_filename: String): self
+ buildModel(config_filename: String): self
+ predict(config_filename: String, image: String, show: Boolean): String
- preSegmentImage(imageobject: ImageProcessing) : ImageProcessing
- checkForConfig(config_name: String): Boolean
Factory
+ CLASS_IMAGE_PROCESSING_SECTION: String
+ CLASS_IMAGE_PROCESSING_STRING: String
+ CLASS_IMAGE_PROCESSING_INSTANTIATED: String
+ CLASS_IMAGE_PROCESSING_BASE64: String
+ CLASS_SEGMENT: String
+ CLASS_OUTLIERS: String
+ CLASS_PREDICT: String
+ CLASS_JSONPARSER: String
+ CLASS_PICKLEPARSER: String
- classes: Dictionary<String, <T>>
+ create(classname : String) : <T>
Segmentation
Parser
ImageProcessing
Predict
Model

Figure 5.7: Facade Design Pattern.

5.3.6 Singleton

The Singleton creational pattern involves a class which is responsible for the instantiation of an object while making sure only one instance of that object exists, as demonstrated in figure 5.8. The class also provides a way to access a single instance without creating a new instance. The instantiation method chosen was Lazy Instantiation since it is the most popular variant and only allocates resources when the instance is needed unlike Eager Instantiation (Stencel, and Węgrzynowicz, 2008).

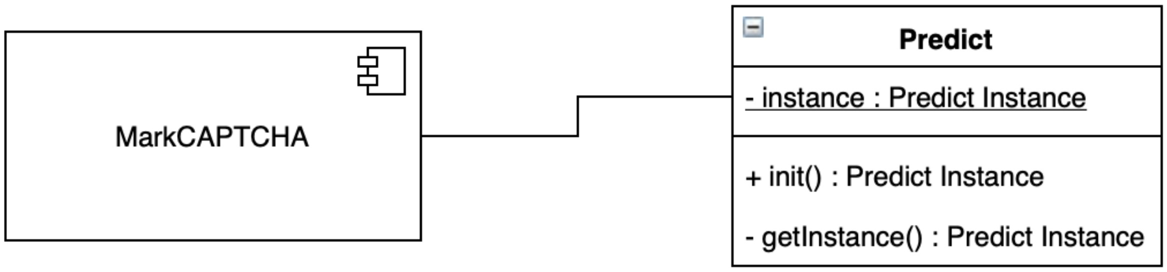

MarkCAPTCHA
Predict
- instance : Predict Instance
+ init() : Predict Instance
- getInstance() : Predict Instance

Figure 5.8: Predict Singleton implementation.

This design pattern was implemented for the Model and Predict class. Both classes require significant amounts of computation power in comparison to the other classes, therefore multiple instances of these classes would result in unnecessary overhead. For example, the Predict class requires significant loading time to load a model. Implementing a Singleton prevented multiple loadings of a model, resulting in accuracy prediction times decreasing by approximately 98%.

5.4 Testing

Testing methods utilized in this project are Black Box Testing, Unit Testing and User Acceptance Testing (UAT). These testing techniques help minimize the number of defects in the system, which in turn produces high quality software.

5.4.1 Black Box Testing

Black Box Testing is a testing technique where the functionality of the system is tested without knowledge of the details of the internal systems. It provides an external viewpoint mimicking a user’s viewpoint. It helps to detect inconsistencies and defects at an early stage of testing. Volunteer testers, Black Box tested the system.

5.4.2 Unit Testing

Unit Testing is where individual components of the system are tested. It concentrates on the system's code. Unit Testing validates that each component of the system works as designed and ensures that changes to the system do not break existing functionality. Unit Testing is done by the developer. Unit tests were achieved with the unittest Python package. Figure 5.9 is an example of Unit tests for the ImageProcessingSection class.

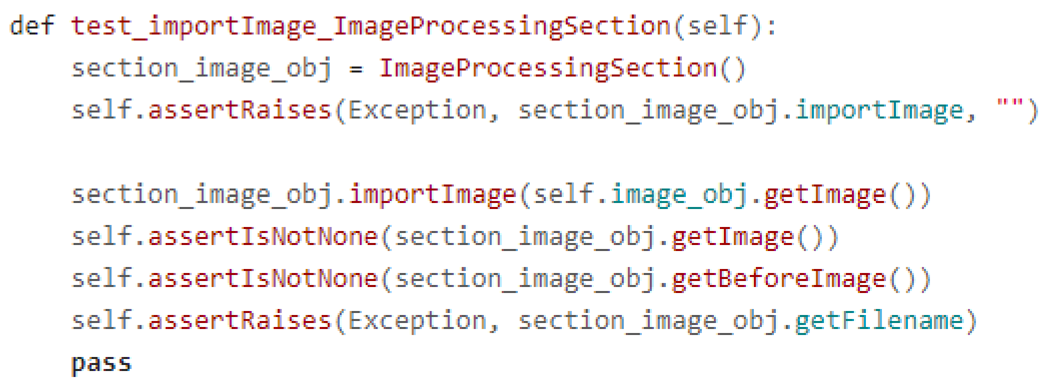

```python
def test_importImage_ImageProcessingSection(self):
    section_image_obj = ImageProcessingSection()
    self.assertRaises(Exception, section_image_obj.importImage, "")

    section_image_obj.importImage(self.image_obj.getImage())
    self.assertIsNotNone(section_image_obj.getImage())
    self.assertIsNotNone(section_image_obj.getBeforeImage())
    self.assertRaises(Exception, section_image_obj.getFilename)
    pass
```

Figure 5.9: Example of Unit Test for ImageProcessingSection.

5.4.3 User Acceptance Testing

UAT is the final phase of the testing process. UAT ensures that requirements and specifications are met. Testing is done by the end user to confirm that the system works in real-life situations. They determine if the system is accepted or not.

5.5 Conclusion

Personal Extreme Programming allowed for agile methodologies to be adapted, allowing for frequent feedback and updates to be applied to the system, thus meeting requirements. The overall system was quite complex and demanding but after design patterns were implemented, it improved performance and complexity. The next chapter will discuss the technologies used and the implementation of the system.

6.0 Software Implementation

6.1 Tools and Technologies

6.1.1 Python

Python was chosen as the development language due to its extensive standard libraries via Pythons Package Index and requires fewer lines of code in comparison to other programming languages. For example, Python is often 5-10 times shorter than equivalent C++ code (Python.org., 1997).

6.1.2 Integrated Development Environment

PyCharm Unlimited was the chosen Integrated Development Environment (IDE). PyCharm is extremely useful for Python development as it identifies compiler errors, runtime errors, unused variables and more. It also automatically generates Unified Modeling Language (UML) class diagrams. For each PyCharm project setup, it recommends the creation of a virtual Python environment, therefore only required Python packages will be installed and used on a new "virtual" Python installation without touching the default Python installation.

6.1.3 Version Control

GitHub, an online hosting service for version control using Git, was the chosen version control and source code management for this project. It allowed the projects source code to be accessed and modified anywhere once a user was authorized. It was helpful for demonstrating progress each sprint and if a mistake occurred, it could be reverted to a prior version. Feature Branching Workflow was the chosen Git workflow for this project. This workflow provides encapsulation by assigning each feature a dedicated branch. This allowed work to be accomplished on a specific feature without distributing the master codebase. It also prevents the master branch from containing defects, thus allowing for better continuous integration environments (Rayana, et al, 2016).

6.1.4 Keras

As discussed in section 3.5, Keras will be used to develop a CNN model. Time was a limiting factor during development, making Keras the ideal solution to develop a CNN since it allows for rapid prototyping to eventually produce a CNN model.

6.1.5 OpenCV

OpenCV was designed to solve computer vision problems, making it the perfect Python library to remove noise and segment CAPTCHA images. OpenCV has a collection of documentation and online tutorials to understand how to implement it into your system.

6.1.6 Docker

Docker is an application that makes it easier for developers to develop, deploy and run applications by using containers. Containers work by packaging application components such as its libraries, together in an isolated environment. Docker uses images instead of virtual machines because images use fewer resources, are portable and faster to start up. By doing so, it is ensured that the container will work on another machine regardless of customized settings on that machine. Docker was utilised in this project so the application could be set up easily across multiple environments, expediting sprint demos.

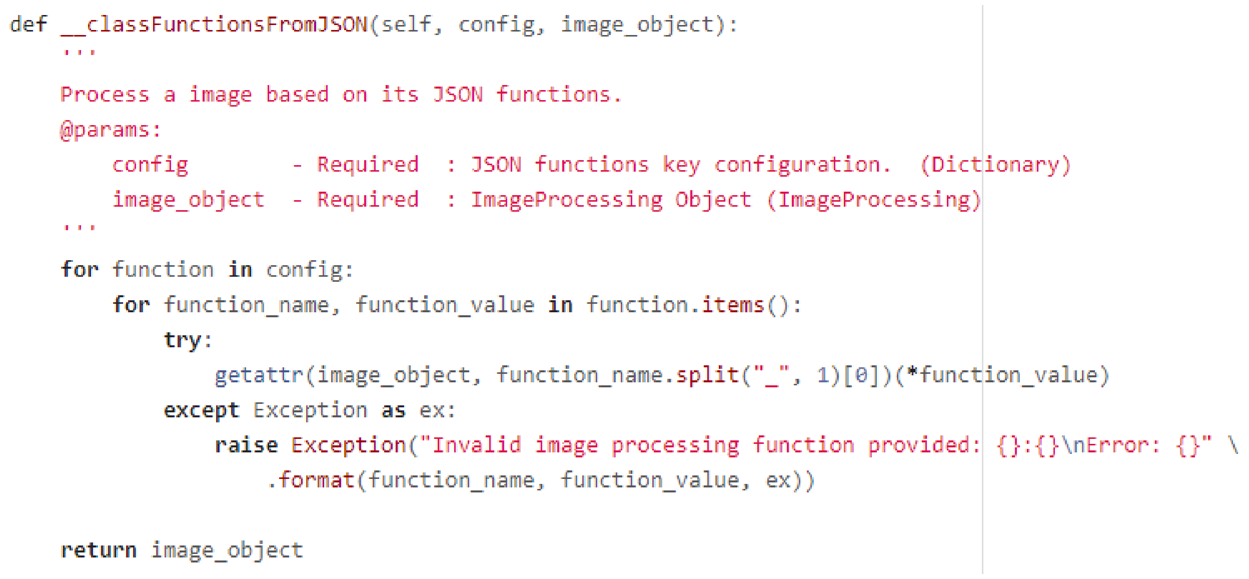
6.1.7 Pathlib

Pathlib is a Python library, that provides an interface to handle paths and files in Python without having to worry about the filesystem of the current system. The peculiarities of the different operating systems are concealed by the Path object, making the system compatible with Windows and Unix operating systems.

6.2 Application Implementation

6.2.1 Modular Image Processing

Image processing is the most important part of the system because, without adequate noise removal, no other functionality will work successfully. The ImageProcessing class provides the functionality required to process CAPTCHA images but what functions and in what order to successfully remove noise requires human analysis and configuration, via a JSON configuration file. Once the human analysis is completed, they produce a configuration file for that specific CAPTCHA. Then when the ImageProcessing function of the system is invoked, the facade class, MarkCAPTCHA processes the CAPTCHA following the configuration instructions in order. Originally a bug caused the CAPTCHA image processing to not be in order i.e. it would attempt to remove contours before thresholding was applied (remove contours is dependant on the image being thresholded). This was due to the "json.load" function not caring about the order. OrderedDict is a subclass of Dictionary, it cares about the order key, value pairs are imported in, unlike a normal Dictionary (Docs.python.org., 2019). Therefore, a parameter called "object_pairs_hook" was supplied to the "json.load" function with the value "OrderedDict". Figure 6.1 shows how a JSON configuration and a ImageProcessing object work cooperatively to process an image. The "getattr" function allows for modularity because for a supplied ImageProcessing object, a function name can be called via a string along with parameters. For example: "*getattr(image_object, "threshold")(100)*", would apply a threshold of 100 to the supplied image object.

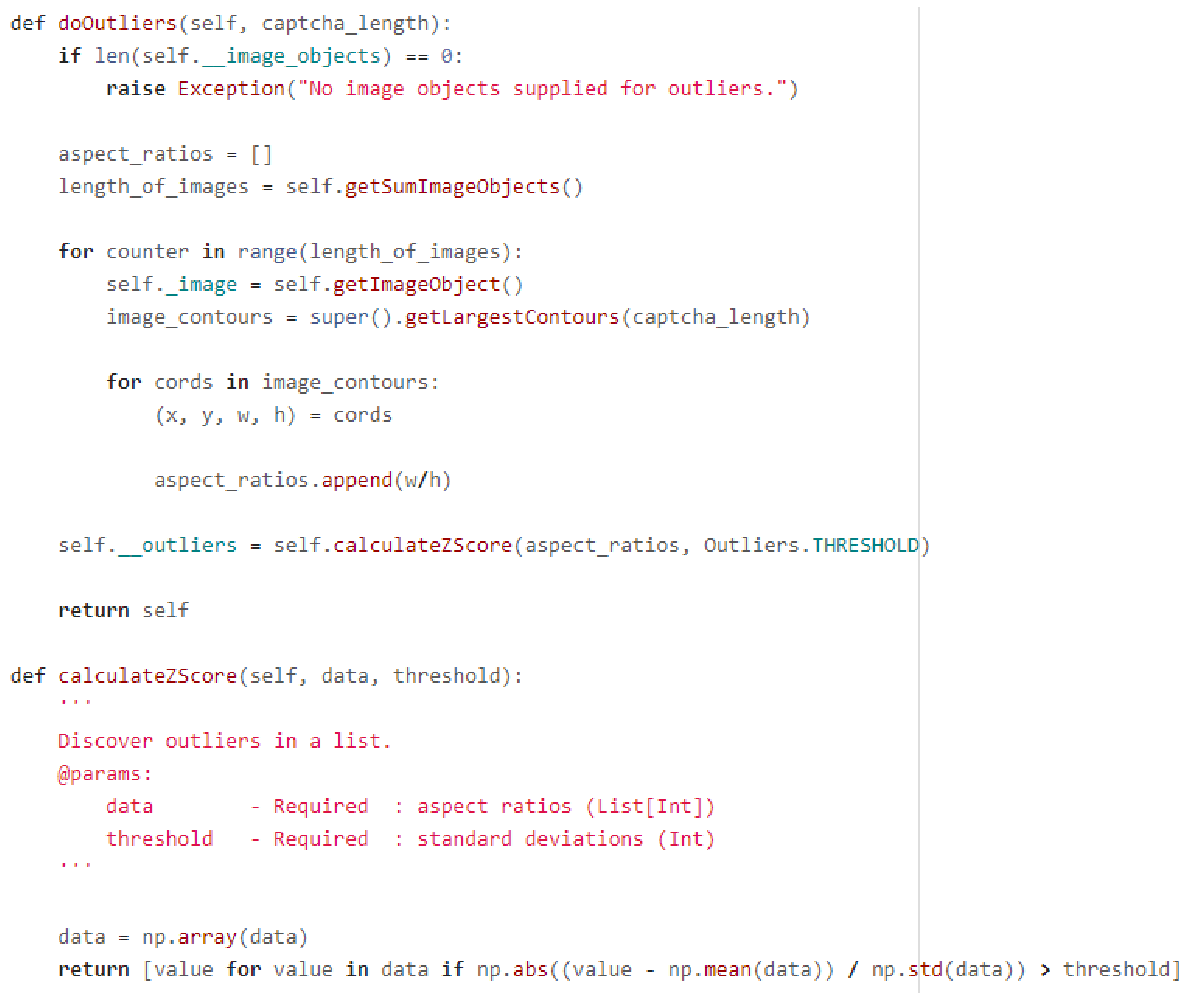


```
def __classFunctionsFromJSON(self, config, image_object):
    '''
    Process a image based on its JSON functions.
    @params:
        config         - Required  : JSON functions key configuration.  (Dictionary)
        image_object   - Required  : ImageProcessing Object (ImageProcessing)
    '''
    for function in config:
        for function_name, function_value in function.items():
            try:
                getattr(image_object, function_name.split("_", 1)[0])(*function_value)
            except Exception as ex:
                raise Exception("Invalid image processing function provided: {}:{}\nError: {}" \
                    .format(function_name, function_value, ex))

    return image_object
```

Figure 6.1: Modular code to process an image via a JSON configuration.

6.2.2 Aspect Ratio Determination

The automated aspect ratio calculation implemented prevented costing the analyst/developer additional time and removed complexity from the system. The automated calculation works in three parts. The first part is the gathering of each cleaned CAPTCHA image. This occurs in the MarkCAPTCHA class once a CAPTCHA image has been cleaned. The second step is once all the CAPTCHA images for a specific CAPTCHA are cleaned, the outliers detection function is invoked. This function calculates the aspect ratio for each contour in a cleaned CAPTCHA image. Figure 6.2 shows the implementation of this process. The final step calculates the z-score based on each contour's aspect ratio provided and outputs it into the CAPTCHA JSON configuration. Due to the large amount of memory used to store image objects for the Outlier process, the stack data structure was implemented to improve memory usage. Each time an image object is needed for segmentation, it is removed from the stack, freeing up memory.



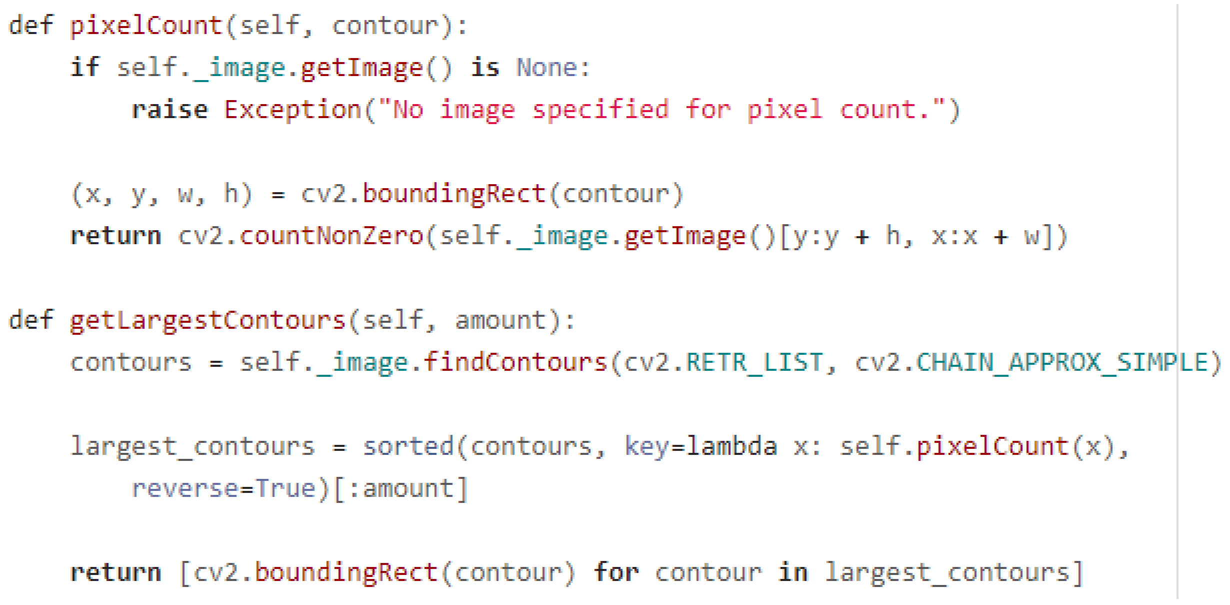
```
def doOutliers(self, captcha_length):
    if len(self.__image_objects) == 0:
        raise Exception("No image objects supplied for outliers.")

    aspect_ratios = []
    length_of_images = self.getSumImageObjects()

    for counter in range(length_of_images):
        self._image = self.getImageObject()
        image_contours = super().getLargestContours(captcha_length)

        for cords in image_contours:
            (x, y, w, h) = cords

            aspect_ratios.append(w/h)

    self.__outliers = self.calculateZScore(aspect_ratios, Outliers.THRESHOLD)

    return self

def calculateZScore(self, data, threshold):
    '''
    Discover outliers in a list.
    @params:
        data        - Required  : aspect ratios (List[Int])
        threshold   - Required  : standard deviations (Int)
    '''

    data = np.array(data)
    return [value for value in data if np.abs((value - np.mean(data)) / np.std(data)) > threshold]
```

Figure 6.2: Code for the calculation of z-score.

6.2.3 Character Segmentation and Labelling

The main process of CAPTCHA segmentation is discussed in section 4.3. This section will explain in depth the segmentation process. CAPTCHA image processing does not remove 100% of noise. The remaining noise may be detected as a character, thus decreasing character classification accuracy. The first step implemented to increase classification accuracy was locating *n* of the largest contours within the CAPTCHA image, where *n* is the maximum CAPTCHA length.

Retrieving the *n* largest contours resulted in CAPTCHA prediction accuracies increasing by approximately 28%. Figure 6.3 demonstrates this implementation.



```
def pixelCount(self, contour):
    if self._image.getImage() is None:
        raise Exception("No image specified for pixel count.")

    (x, y, w, h) = cv2.boundingRect(contour)
    return cv2.countNonZero(self._image.getImage()[y:y + h, x:x + w])

def getLargestContours(self, amount):
    contours = self._image.findContours(cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)

    largest_contours = sorted(contours, key=lambda x: self.pixelCount(x),
        reverse=True)[:amount]

    return [cv2.boundingRect(contour) for contour in largest_contours]
```

Figure 6.3: Code to retrieve n *largest contours.*

Retrieving the *n* largest contours resulted in CAPTCHA prediction accuracies increasing by approximately 28%. After the largest contours are identified, each of those contours are iterated through *i* number of times, where *i* is the maximum CAPTCHA length. Imagine a scenario where there are four contours and the maximum CAPTCHA length is four. Two of those contours are two joined characters and the remaining two are noise. The first two contours largest contours will likely be the joined characters due to their amount of white pixels and their aspect ratio will also likely be greater than the variance (minimum z-score value). Thus, the contour is split down the middle and each side of appended to a list. Figure 6.4 replicates the scenario visually.

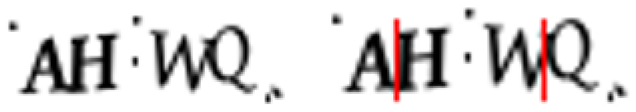
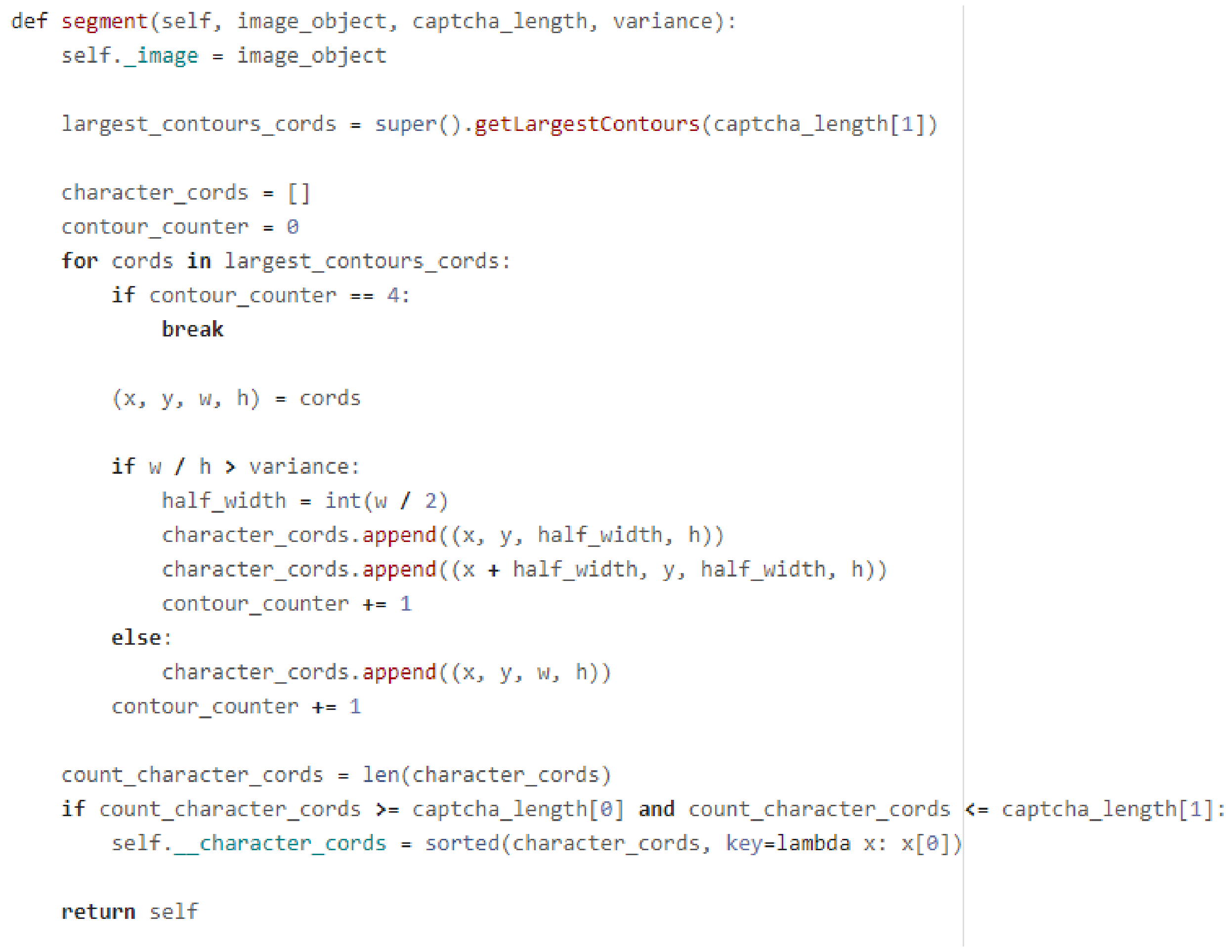


Figure 6.4: Really Simple CAPTCHA scenario demonstration.

Once both the conjoined characters are split, the list length will equal the maximum CAPTCHA length, disregard the remaining contours i.e. the noise. Finally, the list is returned in order of the contours x-axis. Figure 6.5 shows the code implementation.



```
def segment(self, image_object, captcha_length, variance):
    self._image = image_object

    largest_contours_cords = super().getLargestContours(captcha_length[1])

    character_cords = []
    contour_counter = 0
    for cords in largest_contours_cords:
        if contour_counter == 4:
            break

        (x, y, w, h) = cords

        if w / h > variance:
            half_width = int(w / 2)
            character_cords.append((x, y, half_width, h))
            character_cords.append((x + half_width, y, half_width, h))
            contour_counter += 1
        else:
            character_cords.append((x, y, w, h))
        contour_counter += 1

    count_character_cords = len(character_cords)
    if count_character_cords >= captcha_length[0] and count_character_cords <= captcha_length[1]:
        self.__character_cords = sorted(character_cords, key=lambda x: x[0])

    return self
```

Figure 6.5: Character splitting and identification.

Character labelling was a difficult task because findContours returned contours in a random order, resulting in characters images being stored incorrectly. Determining which letter folder the character image is stored in, is determined via the CAPTCHA filename. For example, a CAPTCHA labelled ‘ABCD.png’, the first contour should be ‘A’ and so forth. The resolution to this problem is the x-axis coordinates stored via each contour. The contours were rearranged in order from the smallest x-axis value to the largest. For example, the x-axis coordinate closest to 0 will be ‘A’ and the coordinate furthest away from 0 will be ‘D’. Figure 6.6 displays this process.

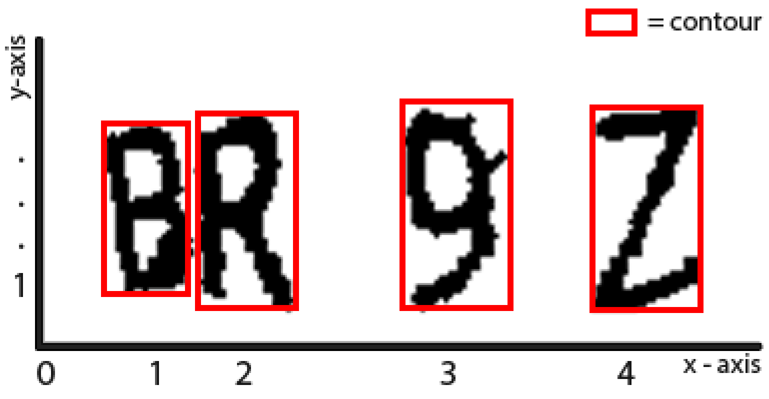
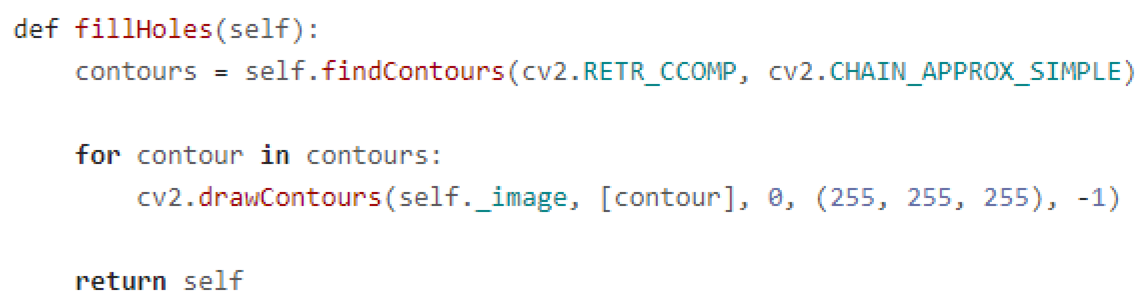


Figure 6.6: CAPTCHA letter identification.

Through further analysis, it was discovered that the segmentation process was outputting poorly segmented data for characters with an opening within them, such as the character ‘9’ or ‘D’. After debugging, it was discovered that the FindContours function was detecting these circle-like shapes as contours. To prevent this, the contours were simply filled with the colour white, helping to produce correct segments. Figure 6.7 shows the code implementation and figure 6.8 demonstrates examples of filled opening contours.



```
def fillHoles(self):
    contours = self.findContours(cv2.RETR_CCOMP, cv2.CHAIN_APPROX_SIMPLE)

    for contour in contours:
        cv2.drawContours(self._image, [contour], 0, (255, 255, 255), -1)

    return self
```

Figure 6.7: Fill contours code.

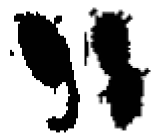
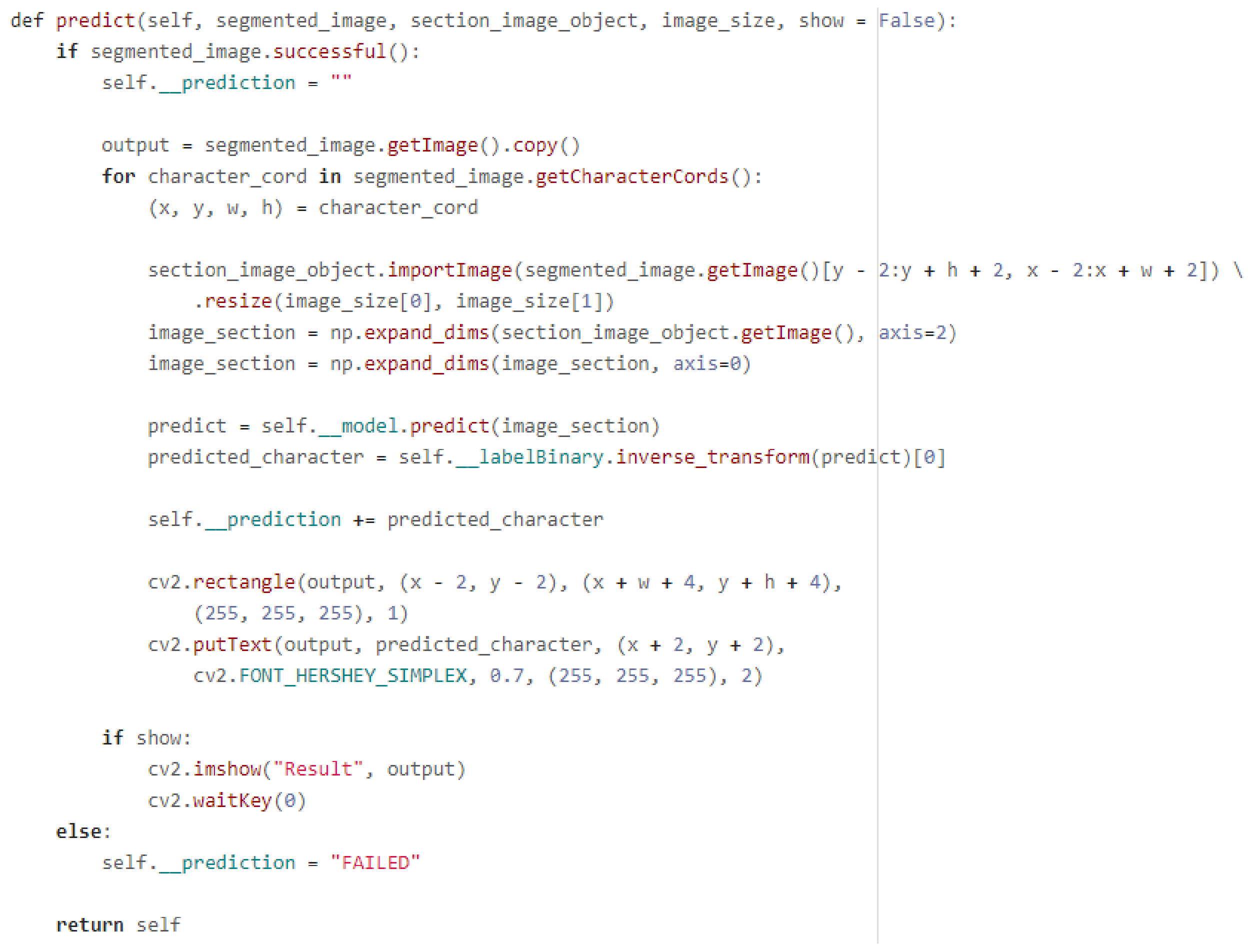


Figure 6.8: Characters with openings filled.

6.2.4 Character Prediction

Since the system follows S.O.L.I.D and DRY principles, the Predict class utilised mostly existing functionality to predict a CAPTCHA. When the predict function is invoked, it is supplied a segmented object. The segmented object has been processed and segmented as previously discussed. It contains character coordinates and a CAPTCHA image. The Predict class checks if the segmentation process was successful. If the process was successful, for each coordinate value (x-axis, y-axis, width and height), the processed CAPTCHA image is cropped and then imported into a ImageSection object. Keras provides a “load_model” function that imports a trained model and allows predictions to be made. A prediction is then made on the ImageSection image and appended to a string. This process is repeated for each contour and the appended string is finally returned. Figure 6.9 shows the implementation of the predict function.



```python
def predict(self, segmented_image, section_image_object, image_size, show = False):
    if segmented_image.successful():
        self.__prediction = ""

        output = segmented_image.getImage().copy()
        for character_cord in segmented_image.getCharacterCords():
            (x, y, w, h) = character_cord

            section_image_object.importImage(segmented_image.getImage()[y - 2:y + h + 2, x - 2:x + w + 2]) \
                .resize(image_size[0], image_size[1])
            image_section = np.expand_dims(section_image_object.getImage(), axis=2)
            image_section = np.expand_dims(image_section, axis=0)

            predict = self.__model.predict(image_section)
            predicted_character = self.__labelBinary.inverse_transform(predict)[0]

            self.__prediction += predicted_character

            cv2.rectangle(output, (x - 2, y - 2), (x + w + 4, y + h + 4),
                (255, 255, 255), 1)
            cv2.putText(output, predicted_character, (x + 2, y + 2),
                cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 255, 255), 2)

        if show:
            cv2.imshow("Result", output)
            cv2.waitKey(0)
    else:
        self.__prediction = "FAILED"

    return self
```

Figure 6.9: Predict function in the Predict class.

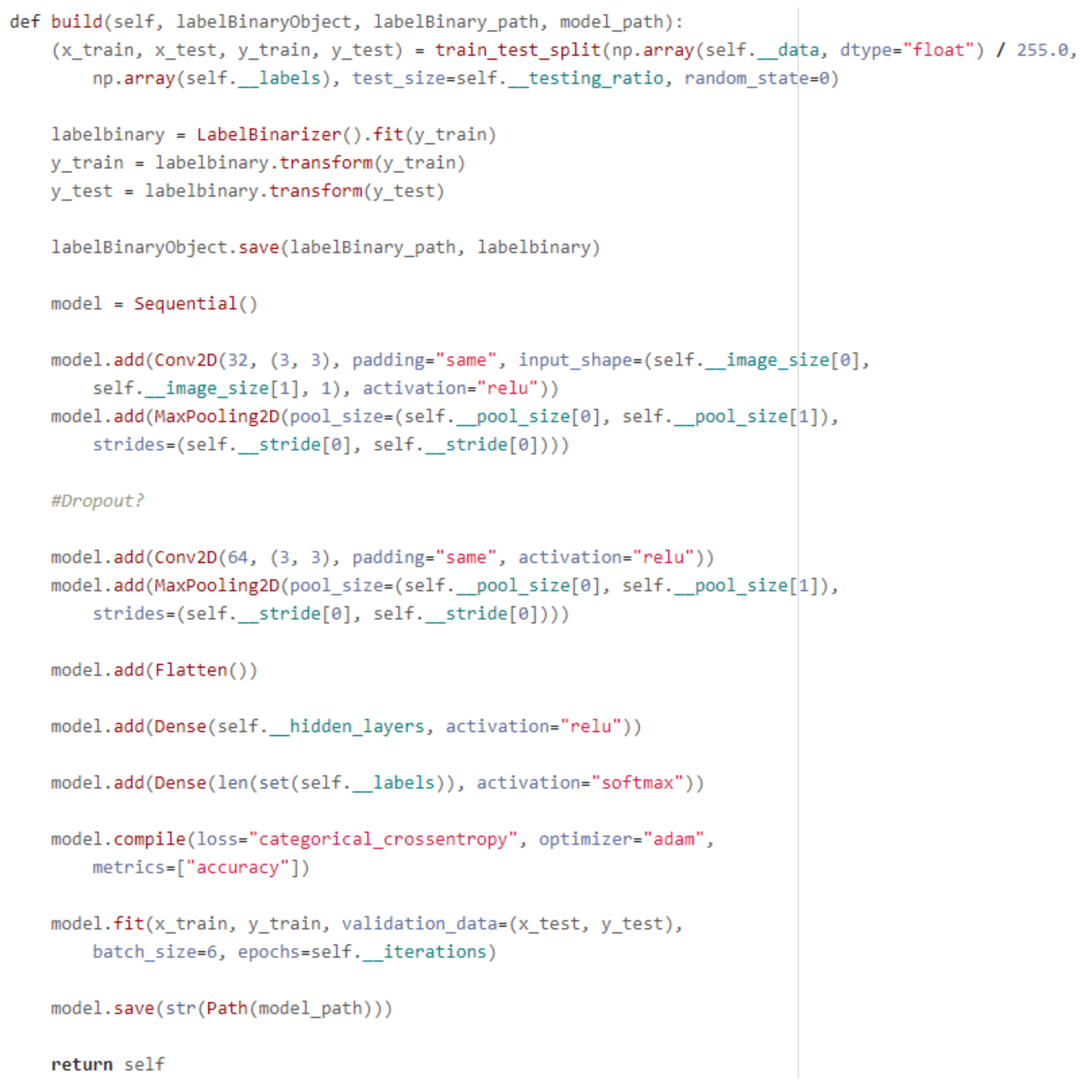
6.2.5 Modular Convolutional Neural Network

In section 4.3 the Keras CNN solution was discussed, in order to make CNN model modular, a JSON configuration file was used. A model object is explicitly instantiated via the JSON configuration. The reasoning for each key, value pair is the JSON configuration is the following:

- **image_size**: The size images will be resized and inputted into the model to be trained.
- **hidden_layers**: The number of hidden layers utilised in the model.
- **iterations**: The number of times a batch of data is passed through the model.

- **testing_ratio**: The ratio of training to testing data for example, 0.2 would mean that 20% of the total data goes to testing.
- **pool_size**: The size of the pooling window.
- **stride**: The size of the stride window used in the pooling layers.

Figure 6.10 is an example of the build method in the Model class. It shows how each key, value is utilised to build a CNN model.



```python
def build(self, labelBinaryObject, labelBinary_path, model_path):
    (x_train, x_test, y_train, y_test) = train_test_split(np.array(self.__data, dtype="float") / 255.0,
        np.array(self.__labels), test_size=self.__testing_ratio, random_state=0)

    labelbinary = LabelBinarizer().fit(y_train)
    y_train = labelbinary.transform(y_train)
    y_test = labelbinary.transform(y_test)

    labelBinaryObject.save(labelBinary_path, labelbinary)

    model = Sequential()

    model.add(Conv2D(32, (3, 3), padding="same", input_shape=(self.__image_size[0],
        self.__image_size[1], 1), activation="relu"))
    model.add(MaxPooling2D(pool_size=(self.__pool_size[0], self.__pool_size[1]),
        strides=(self.__stride[0], self.__stride[0])))

    #Dropout?

    model.add(Conv2D(64, (3, 3), padding="same", activation="relu"))
    model.add(MaxPooling2D(pool_size=(self.__pool_size[0], self.__pool_size[1]),
        strides=(self.__stride[0], self.__stride[0])))

    model.add(Flatten())

    model.add(Dense(self.__hidden_layers, activation="relu"))

    model.add(Dense(len(set(self.__labels)), activation="softmax"))

    model.compile(loss="categorical_crossentropy", optimizer="adam",
        metrics=["accuracy"])

    model.fit(x_train, y_train, validation_data=(x_test, y_test),
        batch_size=6, epochs=self.__iterations)

    model.save(str(Path(model_path)))

    return self
```

Figure 6.10: Code for a modular Convolutional Neural Network.

When training a model its metrics are calculated and displayed i.e accuracy and loss. Accuracy is a percentage of how many successful predictions were made by the model with a sample of x and label y from the testing dataset. Loss is a scalar value that describes how close predictions are to their true labels. Figure 6.11 demonstrates CAPTCHA 0.3 training metrics. These metrics produced constant results of 90%+ but when prediction accuracy was tested it did not match these metrics. Prediction accuracy implemented real world testing of MarkCAPTCHA. Table 6.1 demonstrates how initial prediction accuracy results were inferior to model accuracy metrics. Accuracies of almost 99% are often related to overfitting. Overfitting occurs when a model fits a training set too well. It needs to be avoided since it prevents a model from generalising to new examples. After inspection, the reasoning for high metrics was because character images were extremely similar due to the fact they are just two colours, black and white, where the most dominant colour was black for the background. Therefore, prediction accuracy is used to determine how accurate a CAPTCHA model is.

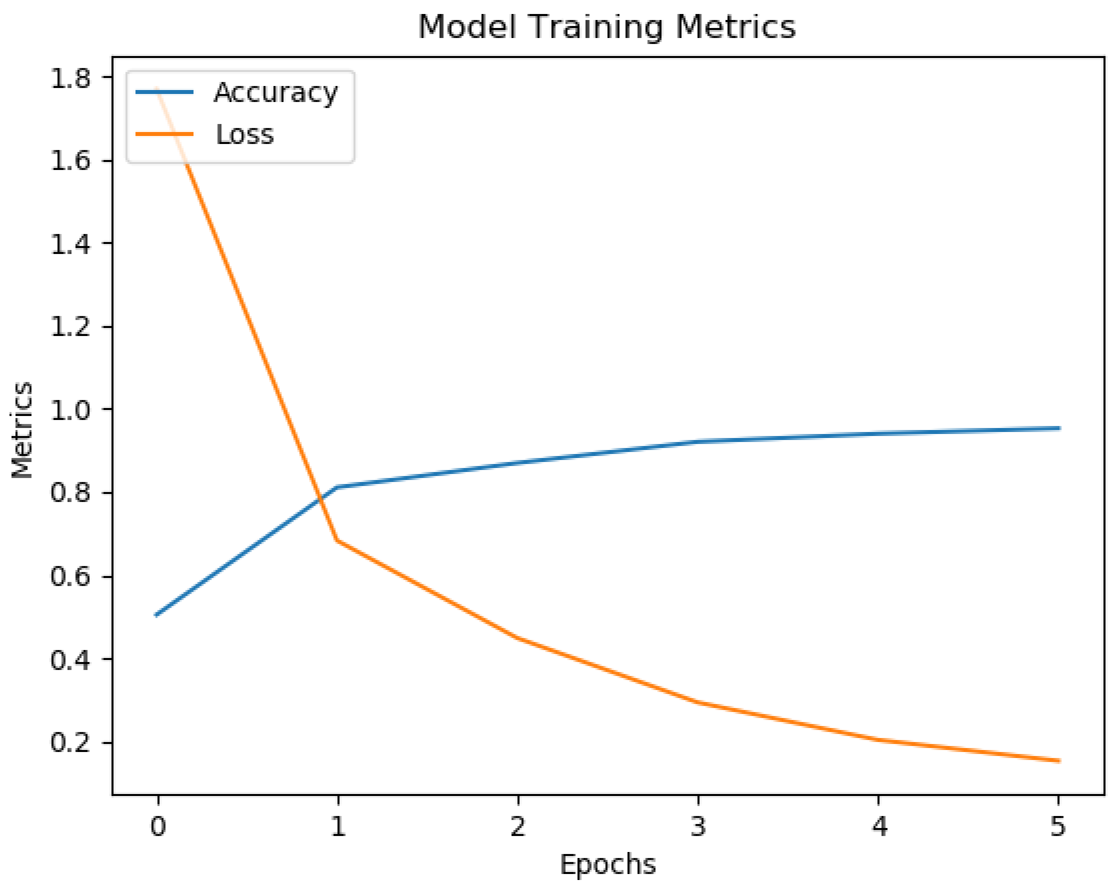


Figure 6.11: Graph of CAPTCHA 0.3's training metrics.

CAPTCHA	**Accuracy**	**Loss**	**Prediction Accuracy** ***(approx.)***
Really Simple CAPTCHA	97%	0.13	60%
CAPTCHA 0.3	95%	0.16	7%
Pastebin CAPTCHA	99%	0.09	70%

Table 6.1: Initial Keras metrics and Prediction Accuracies.

After extensive analysis and testing, table 6.2 displays the final prediction accuracies for each CAPTCHA and the probability that it will skip a CAPTCHA image. A CAPTCHA is skipped if segmentation validation fails.

CAPTCHA	**Prediction Accuracy** ***(approx.)***	**Skipped** ***(approx.)***
Really Simple CAPTCHA	97%	3%
CAPTCHA 0.3	33%	60%
Pastebin CAPTCHA	94%	0%

Table 6.2: Final Testing Accuracies.

CAPTCHA 0.3 produced the weakest results in comparison to the other CAPTCHAs. An examination was undertaken to discover the reason for such a high skip rate and low prediction accuracy. CAPTCHA 0.3 produces multiple conjoined characters instead of just two. Figure 6.12 demonstrates an example of a multiple conjoined character. The current system can only handle two conjoined characters, thus the reasoning for the high skip rate. The low prediction accuracy is due to the amount of anti-recognition features built into CAPTCHA 0.3, however the tradeoff is usability. A survey was conducted amongst 100 random students with no vision impairments, asking if they could resolve five CAPTCHA 0.3 generated CAPTCHAs. The survey resulted with a fail rate of 27%. A high fail rate indicates that CAPTCHAs is too difficult. CAPTCHAs that are burdensome and time-consuming have the potential to lose users, thus a loss of potential sales and traffic (Nakaguro, et al, 2013).

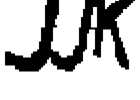
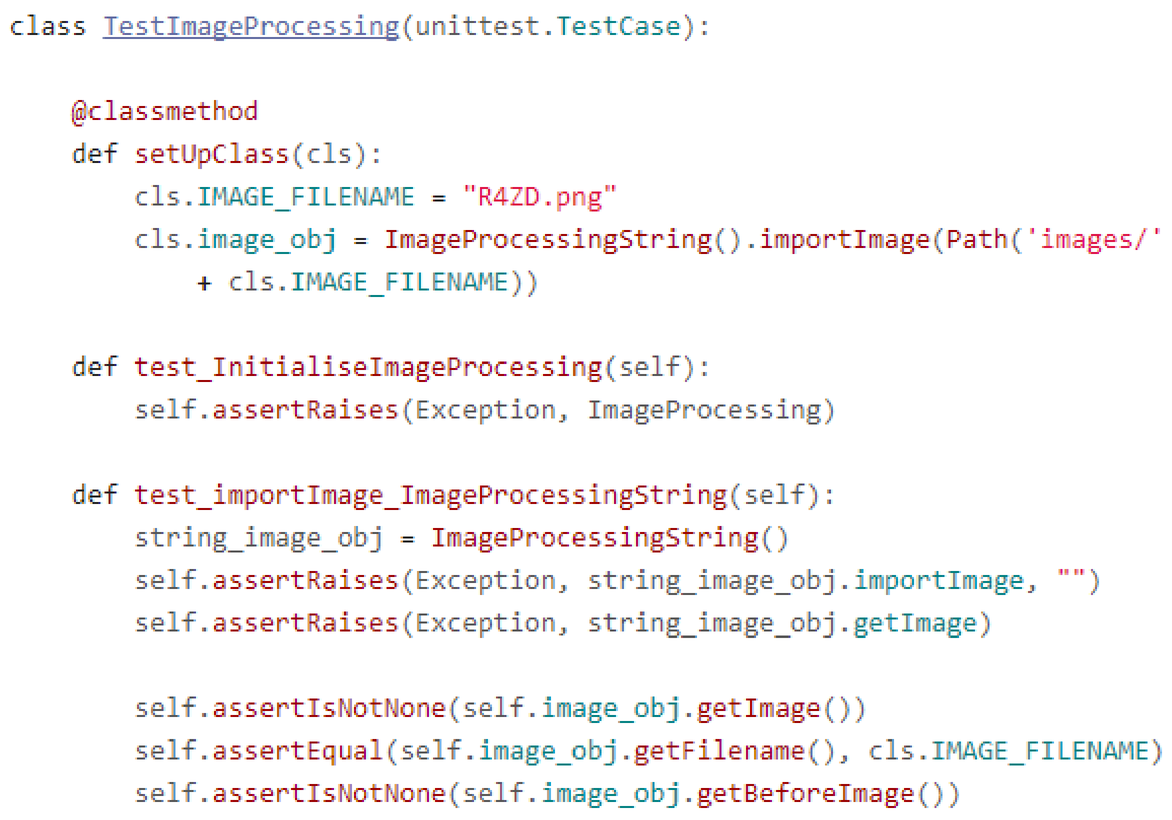


Figure 6.12: CAPTCHA 0.3 conjoined characters.

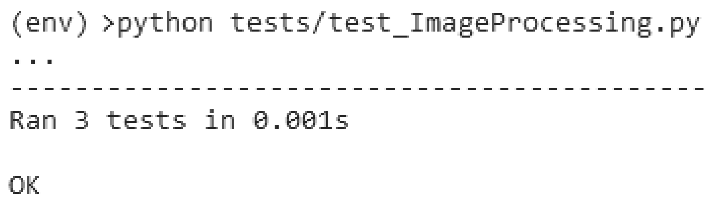
6.3 Application Testing

Testing helped the system conform to high-quality software assurance. Software can never be 100% defect free, making it difficult to determine when to stop testing. The chosen methods for determining when to stop testing were, when all high priority bugs were fixed and upon completion of the project duration (Mailewa, et al, 2015).

Unit tests were implemented to ensure that updates or changes applied to the system did not cause defects. Figure 6.13 is an example of a unit test setup for the ImageProcessing class. The function “test_importImage_ImageProcessingString” ensures that the functions of the ImageProcessingString class behave as expected.



```
class TestImageProcessing(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        cls.IMAGE_FILENAME = "R4ZD.png"
        cls.image_obj = ImageProcessingString().importImage(Path('images/'
            + cls.IMAGE_FILENAME))

    def test_InitialiseImageProcessing(self):
        self.assertRaises(Exception, ImageProcessing)

    def test_importImage_ImageProcessingString(self):
        string_image_obj = ImageProcessingString()
        self.assertRaises(Exception, string_image_obj.importImage, "")
        self.assertRaises(Exception, string_image_obj.getImage)

        self.assertIsNotNone(self.image_obj.getImage())
        self.assertEqual(self.image_obj.getFilename(), cls.IMAGE_FILENAME)
        self.assertIsNotNone(self.image_obj.getBeforeImage())
```



```
(env) >python tests/test_ImageProcessing.py
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

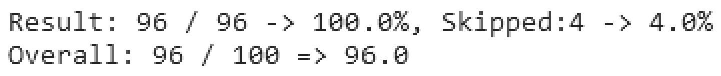
Figure 6.13: Unit test setup for ImageProcessing with results.

The Predict Accuracy test, as discussed in section 6.2.5 worked by matching the prediction output to the image filename. If the prediction output and image filename matched then the CAPTCHA was predicted successfully. The test accepts the following arguments, where an underlined argument is required:

- **config**: The CAPTCHA json configuration filename.

- **<u>folder</u>**: The path for the testing folder, which contains a minimum of 100 CAPTCHA images.
- **sample**: The number of random images selected from the testing folder to predict.
- **show**: Display the CAPTCHA image prediction if it was incorrect.
- **print**: Display each CAPTCHA prediction.

Upon completion of the test, the CAPTCHA prediction accuracy is outputted. Figure 6.14 is an example of the output. The result is a percentage of how many CAPTCHAs were predicted successfully, Skipped is the probability of skipping a CAPTCHA image and Overall is the overall percentage of how many CAPTCHAs were predicted successfully including those skipped.



```
Result: 96 / 96 -> 100.0%, Skipped:4 -> 4.0%
Overall: 96 / 100 => 96.0
```

Figure 6.14: Really Simple CAPTCHA Prediction Accuracy output.

6.4 Conclusion

All these tools and technologies worked in parallel to produce a modular system that hides its complexity. The system was built so it can be used as a microservice, which a developer can easily implement. Pastebin and Really Simple CAPTCHA are insecure and exploitable from the prediction accuracy results. CAPTCHAs that are exploitable allow for automated attacks such as brute forcing a user’s password. The models developed for these CAPTCHAs were so accurate that it predicted the correct CAPTCHA resolution for incorrectly labelled testing CAPTCHAs.

CAPTCHA 0.3 is more secure but at the cost of usability, which in turn can potentially cause a user to leave.

7.0 Reflection

The robustness of each analysed CAPTCHA is as follows: Really Simple CAPTCHA, like its name, is really simple. The amount of installations this Wordpress plugin has is worrying. Wordpress is an extremely popular content management system and should warn its users that this plugin is not secure. Multiple ways that this CAPTCHA can increase its secureness are, a random length, random background noise and a more complex font. Pastebin CAPTCHA is not secure and has the potential of damaging the sites profits, as one of the key features of a Pastebin Pro account is that they will not be burdened with CAPTCHAs. This CAPTCHA should be improved by also utilising a random CAPTCHA length and larger background lines instead of many small ones. Finally, CAPTCHA 0.3 is somewhat secure but a difficult CAPTCHA to resolve. Usability is a key aspect of CAPTCHA and CAPTCHA 0.3 fails to meet that requirement. To improve its usability, characters should not overlap as much and character rotation should be slightly less, since at times that causes further overlapping. These CAPTCHAs are not secure or usable for modern day technology systems. It might even be possible that alphanumeric CAPTCHAs are not secure anymore due to the advancements in technology. Past research papers have proven their ability to break CAPTCHAs time and time again, including checkbox CAPTCHAs. Google reCAPTCHA version 3 is "invisible" but if it thinks you are a bot it will burden the user with a checkbox CAPTCHA. Currently, this CAPTCHA is not known to be broken publicly but previous versions have been proven broken, as discussed in section 2.3. Ultimately CAPTCHA breaking is a game of cat and mouse, as technology improves so must the security behind systems. Webmasters especially web commerce webmasters tend not to implement CAPTCHA into their site to prevent the loss of potential sales. They determined that automated attacks are less of a loss in comparison to losing customers and sales. Instead, bot like behavior is just blocked such as spamming login requests.

7.1 Future Work

7.1.1 RESTful Web Service

REST stands for Representational State Transfer (Wagh, and Thool, 2012). To improve the current system, it could be upgraded into a REST Application Program Interface (API). RESTful services are scalable, maintainable and lightweight. A REST API provides access to resources, where the REST client accesses these resources via a Universal Resource Identifier (URI) and modifies them to their needs. Resources are normally represented in JSON. The REST API would provide the ability for a user to request a CAPTCHA to be predicted by passing their authentication token and the image in base64 format. The reasoning for an authentication token is because the REST API could determine who made the request, thus the API can transition into a business model where each CAPTCHA prediction charges a fee smaller than other CAPTCHA resolution services.

7.1.2 Automated CAPTCHA Setup

Currently setting up a new CAPTCHA for prediction is a repetitive manual process. This can be automated using Python, where all the folders and files can be created automatically with a default JSON configuration. Also implementing a real time image processing user interface (UI) for determining which functions work best for processing a CAPTCHA image would help expedite CAPTCHA image analysis.

7.1.3 Continuous Integration

Continuous Integration (CI) is a practice where small code changes are merged frequently. This would help identify bugs at the start of the development cycle instead of at the end. The goal of CI is to build more robust software. CI works in harmony with agile methodologies since development is done in small increments. Travis CI would be the chosen continuous integration platform since it is “free”. Travis CI helps support the development process by automatically building and testing code changes. Travis CI provides feedback once building and testing are completed. For example, if a unit test failed, the integration will fail and that feedback will be provided.

7.1.4 Multiple Conjoined Characters

Multiple conjoined characters are not handled by the current system due to time constraints. But it is possible for a future version of the system to handle multiple conjoined characters. One possible solution is counting the number of contours after imaging processing and decide based on that amount. For example, if there are two contours after image processing and one contour has an aspect ratio greater than the z-score result and the other does not then divide evenly the contour with the greater aspect ratio *n* times, where *n* is the maximum CAPTCHA length minus 1. This solution has not been implemented therefore its accuracy is uncertain.

7.2 Conclusion

To conclude, this CAPTCHA breaking system has been a success, as each CAPTCHA has achieved an automation resolution accuracy greater than the 1% accuracy determined by researchers to consider a CAPTCHA broken. Implementing CAPTCHA into a system comes down to determining whether automation prevention is more important than the potential loss of users. Arguably, the weakness of alphanumeric CAPTCHAs along with the burden they place on the user and constant security upgrades raises the question as to whether they are a suitable security feature anymore. A possible alternative to CAPTCHAs could be the requirement of users to input their phone number and enter the received code to verify they are not a bot, similar to two step authentication. Of course, this process can also be automated but would come with a far greater cost. Nonetheless, CAPTCHA breaking has helped computer vision and security improve throughout the years and will continue to until CAPTCHA becomes obsolete.

8.0 References

1. Von Ahn, L., Blum, M. and Langford, J., 2002. *Telling humans and computers apart (automatically): or how lazy cryptographers do AI*. School of Computer Science, Carnegie Mellon University.
2. Von Ahn, L., Blum, M., Hopper, N.J. and Langford, J., 2003, May. CAPTCHA: Using hard AI problems for security. In *International Conference on the Theory and Applications of Cryptographic Techniques* (pp. 294-311). Springer, Berlin, Heidelberg.
3. Chen, J., Luo, X., Guo, Y., Zhang, Y. and Gong, D., 2017. A Survey on Breaking Technique of Text-Based CAPTCHA. *Security and Communication Networks*, *2017*.
4. Turing, A. M. (2009). Computing machinery and intelligence. In *Parsing the Turing Test* (pp. 23-65). Springer, Dordrecht. [USE]
5. Moradi, M., & Keyvanpour, M. (2015). CAPTCHA and its Alternatives: A Review. *Security and Communication Networks*, *8*(12), 2135-2156.
6. Kolupaev, A., & Ogijenko, J. (2008). Captchas: Humans vs. bots. *IEEE Security & Privacy*, *6*(1).
7. Stark, F., Hazırbas, C., Triebel, R. and Cremers, D., 2015. Captcha recognition with active deep learning. In *GCPR Workshop on New Challenges in Neural Computation*.
8. Chellapilla, K. and Simard, P.Y., 2005. Using machine learning to break visual human interaction proofs (HIPs). In *Advances in neural information processing systems* (pp. 265-272).
9. Banday, M.T. and Shah, N.A., 2011. A study of captchas for securing web services. *arXiv preprint arXiv:1112.5605*.

10. Mithe, R., Indalkar, S. and Divekar, N., 2013. Optical character recognition. *International journal of recent technology and engineering (IJRTE)*, *2*(1), pp.72-75.

11. Kopp, M., Nikl, M. and Holena, M., 2017. Breaking CAPTCHAs with Convolutional Neural Networks. In *Proceedings of the 17th Conference on Information Technologies-Applications and Theory*.

12. Sivakorn, S., Polakis, J. and Keromytis, A.D., 2016. I'm not a human: Breaking the Google reCAPTCHA. *Black Hat*.

13. C. Hong, B. Lopez-Pineda, K. Rajendran, and A. Recasens, "Breaking Microsoft's CAPTCHA," 2015.

14. Krizhevsky, A., Sutskever, I. and Hinton, G.E., 2012. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

15. WordPress.org. (2018). Really Simple CAPTCHA. [online] Available at: https://wordpress.org/plugins/really-simple-captcha/.

16. Darnstädt, M., Meutzner, H. and Kolossa, D., 2014, December. Reducing the cost of breaking audio captchas by active and semi-supervised learning. In Machine Learning and Applications (ICMLA), 2014 13th International Conference on (pp. 67-73). IEEE.

17. Sano, S., Otsuka, T. and Okuno, H.G., 2013, November. Solving Google's continuous audio CAPTCHA with HMM-based automatic speech recognition. In International Workshop on Security (pp. 36-52). Springer, Berlin, Heidelberg.

18. Hughey, K.B.D.P.G. and Levin, D., 2017. unCaptcha: A Low-Resource Defeat of reCaptcha's Audio Challenge.

19. Bursztein, E., Moscicki, A., Fabry, C., Bethard, S., Mitchell, J.C. and Jurafsky, D., 2014, April. Easy does it: more usable CAPTCHAs. In Proceedings of the 32nd annual ACM conference on Human factors in computing systems (pp. 2637-2646). ACM.

20. reCAPTCHA: Easy on Humans, Hard on Bots. 2018. reCAPTCHA: Easy on Humans, Hard on Bots. [ONLINE] Available at: https://www.google.com/recaptcha/intro/v3.html.

21. Pope, C. and Kaur, K., 2005. Is it human or computer? Defending E-commerce with Captchas. IT professional, 7(2), pp.43-49.

22. Sathya, R. and Abraham, A., 2013. Comparison of supervised and unsupervised learning algorithms for pattern classification. International Journal of Advanced Research in Artificial Intelligence, 2(2), pp.34-38.

23. Fidas, C.A., Voyiatzis, A.G. and Avouris, N.M., 2011, May. On the necessity of user-friendly CAPTCHA. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (pp. 2623-2626). ACM.

24. Keras.io. (2019). Home - Keras Documentation. [online] Available at: https://keras.io/ [Accessed 10 Jan. 2019].

25. Yang, J.I.A., Wang, F.A.N., Chen, Z.H.A.O. and Jungang, H.A.N., 2018, An Approach for Chinese Character Captcha Recognition Using CNN.

26. CRACKING GOOGLE RECAPTCHA WITH DEEP LEARNING - Li Shen, Wei Li and Xiaojin Jiao. (2019). [video] Available at: https://www.youtube.com/watch?v=XeA_BlAItCM [Accessed 10 Jan. 2019].

27. K. Chellapilla, K. Larson, 2005, "Building segmentation based human friendly human interactive proofs", In: Proceedings of the Second International Workshop on Human Interactive Proofs, Springer-Verlag, pp. 1-26.

28. Python.org. (2019). What is Python? Executive Summary. [online] Available at: https://www.python.org/doc/essays/blurb/ [Accessed 12 Jan. 2019].
29. Packaging.python.org. (2019). Analyzing PyPI package downloads — Python Packaging User Guide. [online] Available at: https://packaging.python.org/guides/analyzing-pypi-package-downloads/ [Accessed 27 Mar. 2019].
30. Recorded Future. (2016). Hiding in Plain Paste Site: Malware Encoded as Base64 on Pastebin. [online] Available at: https://www.recordedfuture.com/base64-pastebin-analysis/ [Accessed 27 Mar. 2019].
31. Similarweb.com. (2019). [online] Available at: https://www.similarweb.com/website/pastebin.com [Accessed 27 Mar. 2019].
32. Mahato, S., Pati, P.R., Tiwari, P. and Mishra, R.G., 2015. Implementation of Advanced Captcha based Security System. International Journal of Technolo-gy Innovations and Research, 15, pp.1-7.
33. Docs.opencv.org. (2019). OpenCV: Morphological Transformations. [online] Available at: https://docs.opencv.org/3.4/d9/d61/tutorial_py_morphological_ops.html [Accessed 28 Mar. 2019].
34. Hayes, A. (2019). What a Z-Score Tells Us. [online] Investopedia. Available at: https://www.investopedia.com/terms/z/zscore.asp [Accessed 3 Apr. 2019].
35. Wang, Y., Huang, Y., Zheng, W., Zhou, Z., Liu, D. and Lu, M., 2017, March. Combining convolutional neural network and self-adaptive algorithm to defeat synthetic multi-digit text-based CAPTCHA. In 2017 IEEE International Conference on Industrial Technology (ICIT) (pp. 980-985). IEEE.

36. Nielsen, M.A., 2015. Neural networks and deep learning (Vol. 25). USA: Determination press.

37. Agarwal, R. and Umphress, D., 2008, March. Extreme programming for a single person team. In Proceedings of the 46th Annual Southeast Regional Conference on XX (pp. 82-87). ACM.

38. Huda, M., 2018. Empirically Validated Simplicity Evaluation Model for Object Oriented Software. International Journal of Software Engineering & Applications (IJSEA), 9(6).

39. Stencel, K. and Węgrzynowicz, P., 2008, November. Implementation variants of the singleton design pattern. In OTM Confederated International Conferences" On the Move to Meaningful Internet Systems" (pp. 396-406). Springer, Berlin, Heidelberg.

40. Python.org. (1997). Comparing Python to Other Languages. [online] Available at: https://www.python.org/doc/essays/comparisons/ [Accessed 10 Apr. 2019].

41. Docs.python.org. (2019). collections — Container datatypes — Python 3.7.3 documentation. [online] Available at: https://docs.python.org/3/library/collections.html#collections.OrderedDict [Accessed 14 Apr. 2019].

42. Rayana, R.B., Killian, S., Trangez, N. and Calmettes, A., 2016, November. GitWaterFlow: a successful branching model and tooling, for achieving continuous delivery with multiple version branches. In Proceedings of the 4th International Workshop on Release Engineering-RELENG 2016.

43. Nakaguro, Y., Dailey, M.N., Marukatat, S. and Makhanov, S.S., 2013. Defeating line-noise CAPTCHAs with multiple quadratic snakes. computers & security, 37, pp.91-110.

44. Mailewa, A., Herath, J. and Herath, S., 2015. A Survey of Effective and Efficient Software Testing. In The Midwest Instruction and Computing Symposium.

45. Wagh, K. and Thool, R., 2012. A comparative study of soap vs rest web services provisioning techniques for mobile host. Journal of Information Engineering and Applications, 2(5), pp.12-16.