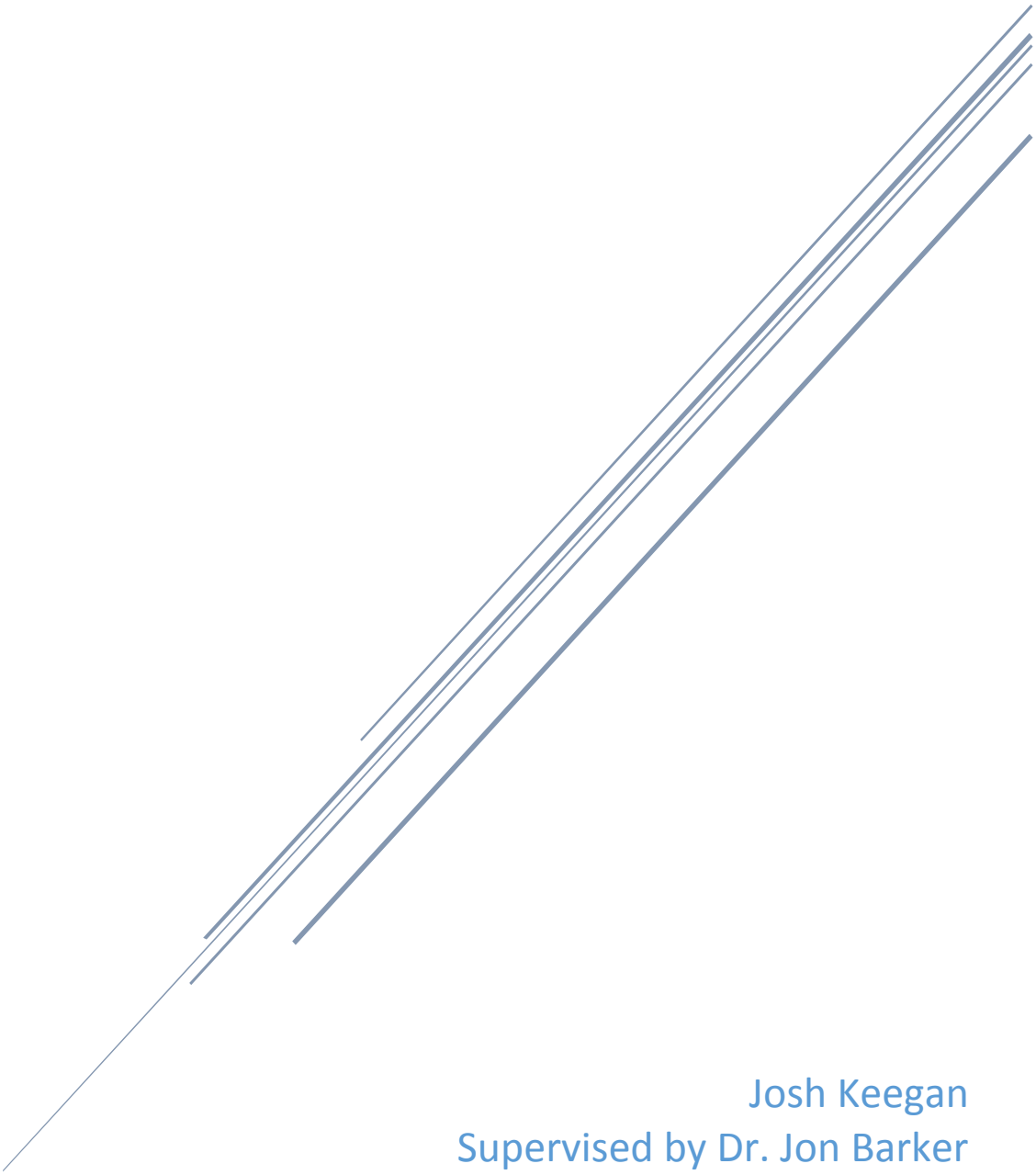
COMPUTER VISION WORD SEARCH SOLVER

COM3600: Individual Research Project



Josh Keegan

Supervised by Dr. Jon Barker

13/06/2014

This report is submitted in partial fulfilment of the requirement for the degree of Bachelor of Science with Honours in Computer Science by Josh Keegan.

Signed Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations which are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: Josh Keegan

Signature:

X _____

Date: 13th of June 2014

Abstract

This project aims to create a Computer Vision system capable of detecting word searches and then using Artificial Intelligence techniques to solve them. There is no one correct process to solve this problem, making the project very experimental in nature.

The system manages to detect and solve 88% of word searches that are surrounded by a bounding box in order to aid detection. Word search detection has been identified as the main area for further work; if a perfect word search detector were to be devised, the overall system performance would rise to 96% of word searches being solved correctly. There are no other computer vision systems detecting and solving word searches known to the author at this time for these results to be compared with.

Acknowledgements

I would like to thank my girlfriend Mel for her support throughout this project and time spent proof reading this report.

Thank you to all of my family who've been through some difficult times lately, your support is invaluable.

I also must thank Andrew Kirillov for his work on the AForge.NET framework, which this project uses extensively. It surpassed my expectations on several occasions and never gave me any problems, being a real pleasure to use.

Finally, I'd like to thank Dr. Jon Barker, my project supervisor, for thinking up this project to begin with. His feedback, expertise and time have contributed greatly to its success.

Contents

Chapter 1 Introduction

Word searches are popular puzzle games that can be very time consuming but are easy for people to solve, with no creative process or difficult problem solving being required. The player must simply find each word in a given list, in a grid of seemingly random characters. For computers however, this problem is not easy; whilst people have complex vision systems processing huge amounts of data in order to know where the word search is, what each character in each word to be found is and what each character that makes up the word search grid is, computers have none of this.

1.1 Aims and Objectives

The overall aim of this project is to create a computer vision system capable of detecting a word search in an image that may contain other data (i.e. it will be on a page, the desk that page is on might also be visible, as well as other objects not relevant to the word search), and then use artificial intelligence techniques to solve it. The system should work with pictures of word searches taken on the current generation of smartphones in order to allow for the software to later be deployed to these devices. A further aim of the system is to be robust to external factors such as lighting, and to work with as wide of a range of inputs as possible (e.g. a word search could be upside down and be skewed in the image).

In order to solve this problem it must be split up into a series of smaller problems that can be approached individually. These smaller problems will emerge throughout the initial chapters from surveying other work on problems with parallels to this and through analysing the overall problem, devising empirical solutions based on real world samples of word searches.

The initial thought might be that this is three separate problems: finding the word search, converting the image representation of it to a grid of characters (that the computer can work with) and finally solving the word search. This simple approach would have limited success because the data each stage would produce will not be perfect when used in the real world. As such, an aim of this project will be to build a system that is robust to errors wherever possible, which will require breaking down the barriers between these simple stages by sending additional data between them in order to allow for errors to be accounted for (e.g. character scores rather than actual characters being returned by classification).

1.2 Challenges

There are no existing computer vision word search solvers that are known to the author at this time, therefore there is no existing knowledge of what such a system should be comprised of and how difficult each problem will be. In order to identify each sub-problem as well as potential solutions, problems that show similarities to this and have already been extensively researched will be surveyed. Some of these problems with parallels to this one are Optical Character Recognition of scanned documents, searching for “found text” in images (e.g. reading the text on road signs from footage taken on board a vehicle) and Automatic Number Plate Recognition. The various solutions for these problems can generally have their stages categorised in to the following:

- Image Analysis
- Pre-Processing

- Feature Extraction
- Classification

Each step encompasses different problems for each domain that may or may not bear relevance to this project; by looking at what has already been done I will be more prepared for the problems this project may face. On top of these domains that share many of the aspects of this project, there are domains in which one or more of the same problems this project will face are tackled but large parts of the problem are different. An example of this would be QR code scanning, where the Pre-Processing and Image Analysis stages must be completed, but then there is no need for the Artificial Intelligence steps (Feature Extraction and Classification), since the transformed image of the QR code can just be sampled at the QR code pitch to see if a square is black or white. The stages of such systems that bear relevance to this project will still be considered.

1.3 Report Structure

The main content of this report follows on from this chapter and is split up into the following six chapters:

- Chapter 2 (Literature Survey) will review existing techniques applied to problems that show similarities to those found in this project.
- Chapter 3 (Analysis and Requirements) will analyse empirical evidence about the problem, as well as consider whether techniques seen in other domains (as identified in Chapter 2) can be used to solve this problem.
- Chapter 4 (Design) gives a detailed plan of how the overall problem will be split up into its individual stages and how these will all fit together. The design should be flexible enough to allow for the method used to solve each problem to be independent of other stages (where possible) and so the design will not consider possible implementations for each stage.
- Chapter 5 (Implementation and Testing) will document how a system meeting the design specified in Chapter 4 has been implemented and tested. Novel aspects of the system will be highlighted and pseudo-code for algorithms will be included where necessary.
- Chapter 6 (Evaluation, Results and Discussion) quantitatively evaluates the performance of the algorithms detailed in Chapter 5, both individually and as a whole system. There will also be discussion of any findings in this chapter and further work will be identified.
- Chapter 7 (Conclusions) summaries the findings of this project and identifies the main areas for further work.

Chapter 2 Literature Survey

This chapter will focus on researching what problems there are in locating and classifying text in computer vision systems and how they can be overcome. In Chapter 1 it was noted that the solutions to various problems that are found in other domains, but show similarities to that of this project can be split up into the following four stages:

- 2.1 Image Analysis – picking out data relevant to the problem from the input data
- 2.2 Pre-Processing – Normalising the input data, reducing variability in order to make the system more robust to external factors (e.g. lighting)
- 2.3 Feature Extraction – Capturing data about something to be classified. It should aim to capture as much data about the input as possible in a small number of independent features
- 2.4 Classification – Determine from the features what class the input data belonged to

This chapter will therefore focus on researching each of these stages independently of one another. In addition to this, Section 2.5 will consider some of the software choices available to the project.

2.1 Image Analysis

The Image Analysis stage is the process of finding the relevant text in an image. In this projects case this is finding where the word search grid lies in the image, and also where each individual character lies so that the image of it can be extracted for feature extraction and classification. This process can be quite bespoke to the application, but the basic idea of what is to be achieved remains the same; e.g. in Automatic Number Plate Recognition, this stage consists of finding the Number Plate in the image and then locating the individual characters in the number plate [1].

In OCR (Optical Character Recognition) document scanning, image analysis is the process of locating a structured body of text on a scanned page and locating individual characters within the body of text. Taking the popular OCR engine Tesseract [2] as an example, how an OCR engine performs image analysis will now be looked into.

OCR document scanning engines have to be able to cope with multiple lines of text, just as this project will. Tesseract approaches this problem by trying to find individual lines of text using a process it calls line construction [3] which attempts to fit the characters into a series of non-overlapping, parallel, but potentially sloping lines. By not requiring the lines to go horizontally across the page, Tesseract does not need to have rotated the image after performing image analysis, thus saving time.

Text with a curved baseline is a common problem in OCR document scanning, and one that this project will also face. It is caused by having multiple pages bound together; whether this is in a book or just a large number of pages stapled together, the page open will not always be completely flat leading to the text not forming completely straight lines. Tesseract was the first OCR engine to solve this problem by replacing the more traditional linear baseline with a quadratic spline that can approximate a curved baseline [3]. Figure 2.1 shows how a quadratic Bezier curve (an example of a quadratic spline whose cubic version is used extensively in computer graphics animation) defines a curve using three control points.

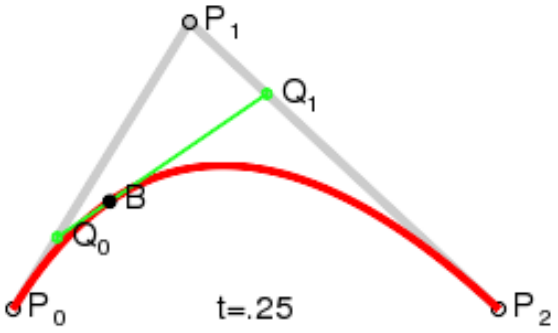


Figure 2.1: Constructing a Quadratic Bezier Curve (an example of a Quadratic Spline). Taken from [4]

Blob recognition/Connected Region Analysis has been used in conjunction with a filter on the shape of the blobs in order to locate quadrilateral objects in an image [5]. This may be relevant to this project because some word searches are bounded by a box and could therefore be located with this method.

OCR document scanning engines have to be able to cope with characters of variable width and joined-up characters, but they also have to be able to work with monospaced fonts. To find out if the font in an image is monospaced, OCR engines perform fixed pitch detection [3]. In order to split characters that are not joined up, the gap between them must be identified; Hoffman and McCullough's dissection method [6] uses a count of the number of black (character) pixels in each column along with the assumption that the text is of a fixed pitch to identify the start and end of each character [7].

2.2 Pre-Processing

Pre-Processing aims to reduce the variability in the input data in order to make the system more robust to external factors. Some of these external factors include lighting conditions, visibility (e.g. fog for Automatic Number Plate Recognition), colour, image blur and noise. The overall aim is to improve classification rates and decrease processing requirements in later stages. Different combinations of pre-processing techniques may be used on the input image for different stages (e.g. in Automatic Number Plate Recognition, colour might be important for finding the number plate, but not for determining what the individual characters are).

During pre-processing, the popular open source OCR engine Tesseract performs a process it calls blob filtering in order to remove noise from the region of an image containing text [3]. This process uses blob recognition to identify all of the connected regions in the image (e.g. a single character), and then discards all blobs whose height is less than that of some fraction of the median height. Essentially blob filtering makes the assumption that text in a region will be of roughly uniform size and uses that assumption to de-noise the image.

Colour image enhancement using the illumination-reflectance model to separate the actual colour of a surface from the effect of lighting [8] has previously been used in Automatic Number Plate Recognition. This technique was used to correctly identify license plates in 2466 / 2483 images [8], which is considered to be "an impressive detection rate" [1]. Recently there has been further work to further improve this model [9].

In a recent article by the author of AForge.NET [10] (a computer vision and artificial intelligence framework), a solution to the problem of glyph recognition is demonstrated [5]; where glyphs are each like a simple version of a QR code, with a grid of (large) black and white pixels identifying each glyph. Due to this project working with discrete shades of black and white, the pre-processing techniques are not concerned with preserving the shade or colour of each pixel values, but instead they just

determine whether it is light or dark. The techniques used are a greyscale filter to reduce the amount of data to process, edge detection to find the border between the white paper and the black border of the glyph, and thresholding to binarise the image.

The thresholding technique used in the Glyph recognition problem was Otsu Thresholding [11] which selects a single threshold value to binarise the entire image with. It selects the threshold value by constructing a histogram of the greyscale pixel values and assuming it to be bimodal (so there are a group of dark pixels, and a group of light ones). The histogram can then be split in to these two classes by minimising the variances of each class (or maximising the intra-class variance between them, which is equivelant); the threshold that splits the two classes is then used to binarise the image. This technique allows for images taken in different lighting conditions to be normalised. However, because it still uses the same threshold for the entire image it will not work for cases where the lighting varies over a single image; in these cases more advanced adaptive thresholding techniques are required, e.g. Bradley Local Thresholding [12].

2.3 Feature Extraction

Feature extraction is the process of converting the raw image data (pixel values) from an image of a found character to a collection of input values for a classifier. Feature extraction methods will vary greatly depending on the type of classifier that they are being used with and the input data they are trying to capture. As such, solutions for problems that do not deal with discrete printed characters are of little interest here as they are trying to capture very different information. The design goals of a feature extractor should be to capture as much of the information necessary to classify the data as possible in a small number of independent features.

Particularly complex feature extraction techniques are not always necessary; sometimes the raw pixel values may give reasonable classifier performance. This is the case for multi-layer neural networks, because of the way they learn: they apply weightings to the output of each node, and so the weightings of the first layer of nodes would effectively rank the pixels by their usefulness for classification. Then one or more hidden layers would be effectively performing feature extraction, working towards the end goal of outputting a probability for each class on the final layer. Neural networks can also be used purely for Feature Reduction; researchers have used them for this with a completely separate classifier [13]. On the other hand, neural networks can be used in combination with any feature extraction technique, as they can have any number of nodes in the first layer. Performing feature extraction separately to classification with neural networks will decrease the number of weights to be adjusted and therefore speed up the process of training the classifier.

Two of the simplest feature extraction techniques applied in handwriting character recognition are Discrete Cosine Transform (DCT) and Principal Component Analysis (PCA) [14]. It is noted that using the Cosine transform offers better information compression than using other unitary transforms (e.g. Sine and Slant transforms [15]) [14]. PCA determines which features to extract from learning the input data, this should lead to improved classifier performance with PCA over DCT, and less features should be required as the main information required to perform the classification should be compressed into fewer features.

2.4 Classification

Classification takes some features from the feature extraction stage and uses them to determine which character the original image represented. Generally this will involve a Machine Learning

algorithm working out where the boundaries are between classes based on some training data and then applying the general trends it has learned to determine the class of the newly acquired data (the character).

It is possible that software already exists that could be used to perform the classification stage. There at least 23 different off-the-shelf OCR engines [16] as of December 2013, of which 7 are open source [16] and so could potentially be used for this project. The popular open source Tesseract OCR engine was analysed when Hewlett Packard released the source code as open source [3] so the processes used by this engine can be reviewed. Whilst the engine contains a static character classifier, it is a very small process in a much larger system with images going through many steps before and after classification. Since Tesseract is designed to be used on scanned documents, it assumes that there is some linguistic structure, with sequences of characters forming words. This process is called Linguistic Analysis and OCR engines routinely use this technique to recover from individual character misclassifications; Tesseract is considered to contain relatively little linguistic analysis [3], but even this would cause problems for word searches where character combinations can be completely random.

An application that is similar to that of this project for classification purposes is Automatic Number Plate Recognition; in this domain researchers tend not to use off-the-shelf OCR software, but instead they train their own classifiers [1]. Researchers appear to be doing this because of the constraints of the problem domain they are working in and designing their own classifiers allows them to build these constraints into the classifier. An example of the constraints for a problem is that number plates are made up of non-linguistic combinations of upper case alphanumeric characters conforming to some pattern of character combinations that are in the same font. Many of these constraints also apply to this problem domain, with word searches being made up of a regularly spaced grid of upper case letters that do not necessarily have any linguistic meaning, all in a sans-serif font. Due to the similarities of the constraints between the two domains, this particular problem domain will be researched in greater detail.

A survey of published Automatic Number Plate Recognition techniques found that there are multiple popular classifiers with Statistical/Hybrid Classifiers utilising Hidden Markov Models and multi-layered feed-forward Neural Networks standing out as the most widely used [1]. It is difficult to establish which techniques actually give the best performance, as each system is designed to work on the number plates used in a specific country meaning that results cannot be directly compared. Despite the differences in the inner workings of these two most common methods, the both produce probabilistic outputs.

It is particularly useful for this project that algorithms producing probabilistic outputs have a proven track record and are widely used, as the probabilities can be used at a later stage to calculate the probability of a sequence of characters being a particular word, thus making the overall system highly robust to character misclassification.

Support Vector Machines have also been successful in the field of Automatic Number Plate Recognition, able to correctly classify up to 98.3% of upper case characters when used on South Korean number plates [17]. SVMs are non-probabilistic classifiers which may reduce the performance of the word finding algorithm (see Section 3.1.5.1). Recently, a probabilistic model has been devised based on SVMs; they showed “how the SVM can be viewed as a maximum likelihood estimate of a class of probabilistic models.” [18]. This particular idea should be treated with caution because there are no existing applications of it in domains similar to that of character classification that are known to the author.

Many researchers choose to use multiple classifiers with probabilistic outputs that are then combined to produce an overall score [1]. The actual methods used to combine the outputs of multiple classifiers are beyond the scope of this Literature Survey; there are papers analysing the different approaches [19]. Should it later be decided to use this technique, further research will be necessary.

2.5 Available Libraries and Programming Languages

The programming language and any libraries available will allow for many of the techniques discussed throughout this chapter to not have to be implemented from scratch by the author. The programming languages and libraries available will be surveyed in this section.

2.5.1 OpenCV Library

OpenCV is an open source C++ library with official interfaces for C, Python and Java and "was designed for computational efficiency and with a strong focus on real-time applications" [20]. It is an extremely popular computer vision library and has unofficial wrappers in many other popular programming languages, e.g. EmguCV for C# [21]. This popularity also leads to widely available community support and good documentation.

2.5.2 AForge.NET Library

AForge.NET is an open-source library written in C# that is "designed for developers and researchers in the fields of Computer Vision and Artificial Intelligence" [10]. It has good documentation for an open-source project (although it is not quite as well documented as OpenCV or MATLAB) and its source code is well commented meaning that it can be easily used for reference where documentation is lacks the required information. It is not as complete or as efficient as OpenCV, but its inclusion of artificial intelligence algorithms (such as classifiers) removes the dependency for another library. Due to it having been implemented entirely in C# it is platform independent thanks to the open-source Mono project [22].

2.5.3 MATLAB

MATLAB "is a high-level language and interactive environment for numerical computation, visualization, and programming" [23]. It is therefore a very different approach to having a separate programming language and computer vision library; instead with MATLAB it should be quicker to implement algorithms from scratch. In addition to this there are many MATLAB implementations of simple computer vision and image processing algorithms available freely online. The main advantages of MATLAB are that because it is such a high-level language, implementation time should be reduced and thanks to its extensive data visualisation tools it would be very easy to visualise the data for each stage of the system, helping to identify errors and aiding in demonstration. MATLAB is a commercial tool, but there is an open-source implementation of the MATLAB language called Octave [24]. However, Octave's performance (in terms of CPU runtime to perform various operations) falls a very long way behind MATLAB and the other lower-level languages surveyed.

Chapter 3 Analysis and Requirements

This chapter is split into three sections: Analysis, Requirements and Evaluation. Each section leads on from the last but there is no overlap between them. Section 3.1 (Analysis) will discuss what has been learned from the Literature Survey (Chapter 2) as well as what may have been already known about the problem and potential ways to go about solving it. The feasibility of the project will also be considered, based on evidence from the literature survey (Chapter 2). Section 3.2 (Requirements) will lay down a set of things the system should be able to do and prioritise them which will heavily influence the order in which features get implemented. Finally, Section 3.3 (Evaluation) will discuss how the system's performance will be evaluated; going into the details of what kind of data will be used, how it will be collected and marked up, and then how both the overall system and individual components will be evaluated using the collected data.

3.1 Analysis

In order to make observations about the population of word searches, three word search books, each containing approximately 100 word searches has been taken as a sample of the population. Characters in word searches are constrained by the following properties:

- All upper case
- No numbers or punctuation (A-Z only)
- Sans-serif fonts
- Always whitespace separating characters
- Characters equally spaced in a grid
- Consecutive characters do not necessarily form words (non-linguistic)

From the domains that have already been considered, Automatic Number Plate Recognition shares the largest number of these properties. The particular properties it shares leads to the feature extraction and classification methods from Automatic Number Plate Recognition being relevant to this project. It does not however share any properties that would allow techniques used in the image analysis stage to necessarily be applicable to this project.

3.1.1 Image Capture

Part of the analysis of this project will be to ensure that it is technically feasible. Since literature review has shown that similar problems in other domains have been tackled successfully, the selected hardware must be checked. The hardware that would be a limiting factor here is the camera, since all of the data used in the project will be captured with it.

The camera that will be used to capture the data used to train and evaluate the system is an Apple iPhone 4. It has been chosen because it is representative of the camera performance that can be expected from a modern smartphone (a device that a finished product based on this system could be deployed to). Further discussion of the camera choice can be found in Section 3.3 . The training data being collected with this camera should not prevent the system from working with other cameras of performance that is greater than or equal to the iPhone 4's 5 MP camera.

The resolution of images produced by this camera are 2,592px × 1,936px. If the assumption was made that a picture of a word search could be taken such that it filled the picture with nothing else in it, and

the reasonable number of rows and columns for a typical large word search (ascertained by surveying word search puzzle books) is 20x25 then the image size per character can be deduced. The image size for an individual character is 129px x 77px if the picture is taken in landscape orientation or 96px x 103px if taken in portrait mode. From here on it will be assumed pictures are taken in landscape mode since characters are taller than they are wide, more pixels should be used to encode this. Since the gap between one character beginning in a row and the next one starting is roughly equal to the width of each monospaced character, and the same applies vertically to the columns then the resolution of each actual character is halved to 64px x 38px. Character classification has been performed with a resolution as low as 60px x 25px [25] so it should be possible to classify each character. Therefore it must be required that for word searches with larger numbers of characters, the entire image must be filled, with this constraint being relaxed as the word search gets smaller (in terms of the number of rows and columns it is made up of).

3.1.2 Image Analysis

Image Analysis involves finding the word search in the image, extracting it and then splitting it up into individual characters. Many of the techniques used for finding objects discussed previously in Section 2.1 are very specific to their application (e.g. the use of fuzzy logic and knowledge of license plate colour). Here techniques that could be useful for the detection of word searches and splitting them up into their individual characters are identified.

3.1.2.1 Word search Recognition

Word search recognition is the problem of locating a word search in a wider image that may contain other items in the background. This is something that has largely not been helped by researching problems in similar domains. One possibility that was inspired by the approach for detecting and recognising QR style glyphs (discussed previously in Section 2.2) is to require the word searches to be surrounded by a bounding rectangle and then search for the bounding rectangle rather than the characters making up the grid. This approach could be implemented using connected region analysis/blob recognition to find the blobs in the image, followed by filtering out blobs that are quadrilaterals [5].

It would be preferred to not have to impose an artificial constraint on the problem such as having to surround the word search with a bounding box. One option would be to perform blob recognition on the image in order to find all of the characters (amongst other things), and then search through the positions of the blobs for a set of similarly sized blobs whose positions form a grid (allowing for skew and rotation).

3.1.2.2 Curved Baseline Correction

The problem of test having a curved baseline (generally caused by the spline of a book forcing one side of a page up preventing the page from being flat) has been observed in word searches when they are in books. A solution to this problem will be to fit a quadratic spline to some text in the image in order to approximate the curvature (as discussed previously in Section 2.1 and used in OCR document scanning [3]). Once the curvature of the page has been approximated, the image is corrected to remove it.

3.1.2.3 Word search Segmentation

Word search segmentation aims to split the image of a word search up into images of individual characters. The most promising approach to this problem appears to be the Hoffman and McCullough segmentation method [6] for individual words typed in a monospaced font, with this being applied both horizontally and vertically across the entire word search in order to create a regular grid from

which character images can be extracted. This has been selected due to its successful usage in OCR document scanning (as previously discussed in Section 2.1) for monospaced fonts.

3.1.3 Pre-Processing

Pre-processing techniques will be the part of the system that has the greatest impact on its ability to handle external factors (e.g. lighting). The Literature Survey (Section 2.2) has shown that some techniques (or variances of them) are used in multiple domains, whereas some of them are more restricted in their applications. This section will look at the problems that need to be solved with pre-processing and which techniques should give the best results in this domain.

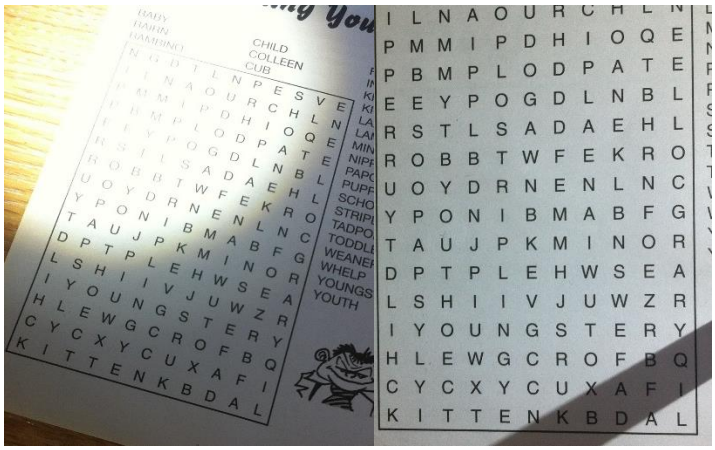


Figure 3.1 Pictures of word searches in varied lighting conditions. Left: a very bright spotlight being shone directly on to part of the word search. Right: a bright light is off and to the right of the word search, with the leg of the camera stand casting a shadow over the word search

Lighting must be corrected for in order to satisfy the System Requirements (Section 3.2); it must be able to cope with a variety of lighting conditions (within reason), including varied lighting across the same image. Figure 3.1 shows how varied lighting can affect pictures of word searches. Whilst advanced techniques such as the illumination-reflectance model are used for colour image enhancement in domains where colour information is useful (e.g. licence plate recognition based on their background colour), in this problem domain it is not necessary to preserve colour information during the pre-processing stage and so simpler thresholding techniques can be applied instead. Thresholding has the advantage of binarising the image which reduces the amount of data that needs to be processed in later stages. In order to be able to cope with varied lighting in a single image, adaptive thresholding techniques (such as Bradley Local Thresholding [12]) will be used. Before a threshold is applied to an image, it must be converted to greyscale; this is a simple process of summing the RGB components of each pixel after multiplying each one by a separate weight. Each of the RGB components is weighted in order to closer reflect the importance that each colour would be given by the human eye and there are industry standards defined for the weighting of each colour component (e.g. BT.709 [26]).

Once the word search has been recognised and extracted from the image, further noise reduction can be performed using the blob filtering technique as is used by the Tesseract OCR engine (as discussed in Section 2.2). In doing so, very large erroneous blobs that could interfere with the word search segmentation will be removed, and very small noise will also be removed, leading to a much cleaner data set going in to the feature extraction stage.

3.1.4 Feature Extraction

Feature extraction will take the pixel values of an image of an individual character, with it having already undergone some Pre-processing. Useful information that maximises the inter-class variance must then be generated. From the Analysis of what makes up a word search (Section 3.1), it can be seen that characters for classification will all be upper case, alphabetic, in a sans-serif font and surrounded by whitespace; this information will be used to make a decision about the feature extraction methods that should be used.

From the literature survey of feature extractors (Section 2.3) it can be expected that the best feature extractor for this project will be PCA (Principal Component Analysis). However, PCA is more complex than other techniques whose performance may come close such as DCT (Discrete Cosine Transform).

3.1.5 Classification

A classifier will be required for the system to be able to distinguish between different characters so that it can perform the final task of finding the words in the grid of characters.

3.1.5.1 Probabilistic vs. Non-Probabilistic

Classifiers can be split into two categories: ones that give probabilistic outputs and ones that do not. It is reasonable to assume that the chosen classifier will not be 100% accurate over all of the collected data (and of course other real world data), so it is important to consider what happens when a misclassification is made and how the system will recover from this error. The recovery options vary dependent upon the type of classifier chosen.

If the system were to naively assume that the classifier never misclassified any character, and simply search through the grid of characters for the words being looked for, then any mistakes made (on characters that make up part of a word) would prevent word(s) from being found. The system must therefore acknowledge that misclassifications can happen and have in place methods for recovery.

A system that would recover from misclassifications would require a probability to be assigned to each character (e.g. 90% sure it is a B, but 10% chance of it being a D). The system could then use the probabilities when going through the grid for each word to find the position that has the maximum probability of being the location of that word. Such an algorithm would guarantee a solution for each word. If desired, the system could still only generate solutions that are very likely to be correct; since it uses the character probabilities it could generate an overall word probability and use that to omit word locations that have particularly low probabilities. This would be the desired behaviour of the system in a production environment, so that the user could be notified of any words that were not found but for evaluation it is desirable to have a solution for every word.

One option for generating probabilities that would work with any type of classifier is to use a Confusion Matrix generated by evaluating the classifier on the cross-validation data set in order to accommodate for one character often being misclassified as another (e.g. B and D can look similar and therefore are going to be more commonly misclassified as each other than S and T). Once the classifier has determined which character an image contains, the probabilities from the confusion matrix for other characters being misclassified as that one will be used.

Probabilistic classifiers return a probability per class, so the classifier will recognise that a character is likely to be a B, but could be a D and provide the probability of it being each. Therefore, the output of a probabilistic classifier gives more information about the image being classified which can then be used to increase robustness to character misclassifications. It would also be possible to combine the

probabilities outputted with those from a confusion matrix generated on the cross-validation data, which could potentially further improve the accuracy of the probabilities assigned to each class. Therefore it is desirable to use a probabilistic classifier for the purposes of this project.

3.1.5.2 Classifier Selection

In Section 3.1.5.1 it was determined that a probabilistic classifier would suit the needs of this project best.

Due to the similarities between the character data used in this project and the characters seen in Automatic License Plate Recognition systems, it can be reasoned that classifiers that perform well in that domain should also perform well for this project. Therefore the type of classifiers that will be selected for use in this project will be the ones that have been shown to perform well in Automatic Number Plate Recognition. The most common classifiers used are multi-layer feed forward Neural Networks. This meets the requirement laid out in Section 3.1.5.1 that a probabilistic classifier would better suit the needs of this project as the probabilistic outputs could be used to help solve the word search.

3.2 Requirements

What the system is required to do is laid out here. Each requirement has an importance associated with it; mandatory requirements must be implemented in order for the project to be considered a success.

3.2.1 Functional Requirements

3.2.1.1 Mandatory

1. Find a word search present in an image
2. Split an image of a word search into individual characters
3. Classify characters
4. Automated system evaluation. Outputs accuracy scores for word search recognition, character classification and overall percentage of words found correctly

3.2.1.2 Desired

1. Demonstration GUI to load an image for processing
2. Demonstration GUI to show the solved word search (grid of characters with lines drawn on the solutions)

3.2.1.3 Optional

1. Demonstration GUI to manually adjust the bounding box for a word search found in an image
2. Progress bar for word search solver demonstration GUI, as it could take several seconds to process an image
3. Work on word searches in any orientation (so the picture could be taken upside down, or from the left or right)

3.2.2 Non-functional Requirements

3.2.2.1 Mandatory

1. Stability. The system must fail gracefully where possible, with errors being logged.
2. Work in different lighting conditions (image can be dull or bright)

3.2.2.2 Desired

1. User-friendly GUI
2. Work with varied lighting in the image

3.2.2.3 Optional

1. Robust to small bends in the paper the word search is on
2. Allow for multiple word searches in one image

In addition to what is required of the system, there are some requirements for the user when taking the photographs:

1. Hold the camera reasonably still, so as to not create a blurry photograph
2. Do not obstruct the view of the word search
3. Keep the word search as flat as possible
4. Have the word search fill as much of the photograph as possible

3.3 Evaluation

3.3.1 Evaluation Data

3.3.1.1 Data Sets

In order to evaluate the performance of the system, I will collect three sets of data:

- Training Data
- Cross-Validation Data (also known as Development Data)
- Evaluation Data

It would be possible to get a set of scores out using the cross-validation data, however it may have been used during the feature selection process for the classifier and therefore it would be possible to over-train a classifier to recognise that data set, rather than the general trends.

If the system were to be evaluated on the same data that was used to train it, there is no way of knowing if the system over-learned the data rather than the general trends of the population. Having an evaluation data set solves this problem and will allow me to reliably evaluate the performance of my system; the downside is the (time) cost of collecting, and even more so, the cost of labelling this additional data. Cross-Validation data is collected in order to aid in the classifier training process.

3.3.1.2 External Factors

It is important that the data collected is representative of the real world scenarios that the system could be used in, so that the system can be trained to cope with these external factors and later have its performance evaluated on them. In each of the three data sets there should be at least one image that contains a word search affected by each of the following potential external factors:

- Rotation of angles less than 45° so that the word search is still the right way up
- Rotation of angles between 45° and 315° so that word search is not oriented correctly
- Scale (examples of a word search taking up almost the whole image, and one where the word search takes up less than 1/8 of the image)
- Skew
- Multiple word searches in one image
- Uneven lighting

- Page not completely flat (only minor variations in this, as might be expected of a word search being in a bound book)
- Blur (as would be found if the camera were not held perfectly steady)

I will assume that each of these factors is independent and aim to have an example of each separately rather than having examples of every possible combination of them. Whilst it would be nice to have such an extensive set of data, the amount of time required to undertake a data collection task on that scale is prohibitive. I will however still have some data showing combinations of these external factors to show the system can cope with them when each factor is not isolated.

3.3.1.3 Data collection

Data will be captured using a typical smart phone camera in order to demonstrate the effectiveness of the system on a real world device to which the software could be deployed. An iPhone 4 has been chosen for this role because it is available to the author for data collection. It is a suitable candidate because of its popularity and age making it a representative baseline for smart phones, and its 5MP (mega-pixel) camera being of relatively low resolution by 2014 standards. This means that anything this project manages with a 5MP image, should also be possible on today's current-generation of smart phones with higher resolution cameras. Whilst capturing the data, the camera will be fixed in the position required for such an effect to be seen and then each set of data for that external factor will be captured.

3.3.1.4 Data Mark-up

Once all of the data has been captured, then the position of each word search in each image will have to be marked up and each character in each word search entered manually. In order to reduce the amount of data that has to be entered into the system multiple images will be taken of the same word search; this means that several images worth of characters can be entered at once as they are all pictures of the same set of characters. To help with the data entry process, a program that allows the user to select the regions containing word searches, enter the number of rows and columns and then type in each character individually will be developed. The system would then store the newly acquired metadata in an XML file for later use.

3.3.2 Evaluation Methods

The overall system will be evaluated in order to allow for the overall success of the project to be measured. Individual components of the system will be evaluated in order to provide feedback during development, to measure the success of the project in individual areas and to allow for areas of further work to be identified.

The overall system will be scored by iteratively providing it with the images from the evaluation data, and then checking if the positions it returns for each word are correct (by checking how many words are found correctly when the word positions returned are looked up against the labelled data). Since the system will aim to return a position for every word it is looking for, it does not matter whether a wrong answer is a false positive or a false negative; performance can simply be measured by:

$$\frac{\#\ words\ found\ correctly}{\#\ words\ looking\ for}$$

This is equivalent to the positive predictive value, or precision of the system [27].

Each classifier and feature extractor pair will be evaluated separately. This is because they are such key parts of the system and any problems should show in the evaluation results. All of the data that is

required for this will have already been captured for the overall system evaluation, which means there is no additional data collection required. The measure of performance that will be used for this evaluation is character accuracy as defined by the Information Science Research Institute's Annual tests of OCR Accuracy [28]:

$$\frac{n - \#\ errors}{n}$$

Where ***n*** is the total number of characters in the evaluation data. This measure is simple to understand and sums up the overall performance of the character classifier. The characters passed into this stage will have been correctly marked up by hand rather than relying on the word search recognition algorithm, so that the score is representative of the performance of this part of the system, not any stages beforehand. The evaluation may also include a Confusion matrix to show what errors the system is making (e.g. misclassifying a B as a D) to show where the system needs to be improved.

The word search recognition algorithm will also be scored separately because any failure in this early stage of the system is unrecoverable later; this means that the word search recognition algorithm will have a great impact on overall system performance. This part of the system will be evaluated by a two part pass/fail evaluation, with the first part testing whether the word search in the image has been located correctly and the second part testing whether the correct number of rows and columns have been detected. Detecting extra phantom word searches at this stage will not be penalised, as they will not negatively impact any word searches that have been identified correctly. In order to allow for different detection algorithms potentially detecting the corners of the word search in slightly different places and the hand mark-up not being pixel perfect, the corners of each word search will be checked whether they are within some radius (relative to the image dimensions) of the correct value.

Chapter 4 Design

This chapter is split into two sections that are fairly independent of one another. In Design Methodology (Section 4.1), a methodology for building the system will be selected. In the Core System Overview (Section 4.2), a design for the system that meets the requirements (as specified in Section 3.2) will be split up into a series of smaller problems that can be solved independently. The basic details of what each component should do and the data it should accept and return will be detailed.

4.1 Design Methodology

Design methodologies are used by Software Engineers to describe how they will go about building a system. There are a variety of different methodologies, designed to meet the needs of different businesses, their clients and the scale of the project being undertaken. This section will explain the methodology being used and the reasons for choosing it.

An Iterative and Incremental [29] methodology has been chosen for this project because it allows for the requirements to be flexible throughout the project life, is not focussed purely on larger projects where developers work in teams, and allows for features to be removed if there is not enough time to implement them without having already spent time planning and designing them.

In the Iterative and Incremental methodology, software engineers develop small parts of the system at a time (incrementally) and then improvements are made to each part and the system as a whole through complete iterations of the development cycle.

This approach means that there should always be a working version of the system available on the master branch (or equivalent for the VCS (Version Control System) being employed) albeit with some requirements potentially not yet implemented and to varying degrees where further iterations may be necessary. This gives the advantage of being able to stop development prematurely (most likely due to time constraints) and always have a working system to gain some results from.

4.2 Core System Overview

The core functionality of the system will be contained in one or more software libraries; it is the functionality provided by these libraries that will be discussed in this section. The system will be split up into modules at the top level so that the problem becomes several smaller problems. At this level, no time is spent considering how each problem will be solved, but rather that a problem exists and will need to be solved. Once a stage is being designed, it may become apparent that the approach being taken to solve it is dependent upon another stage, this will have an effect on the order in which stages get implemented.

Figure 4.1 is a Data Flow Diagram showing the system broken down into these individual modules, with each module labelled and having the data it takes as its input and produces as its output shown.

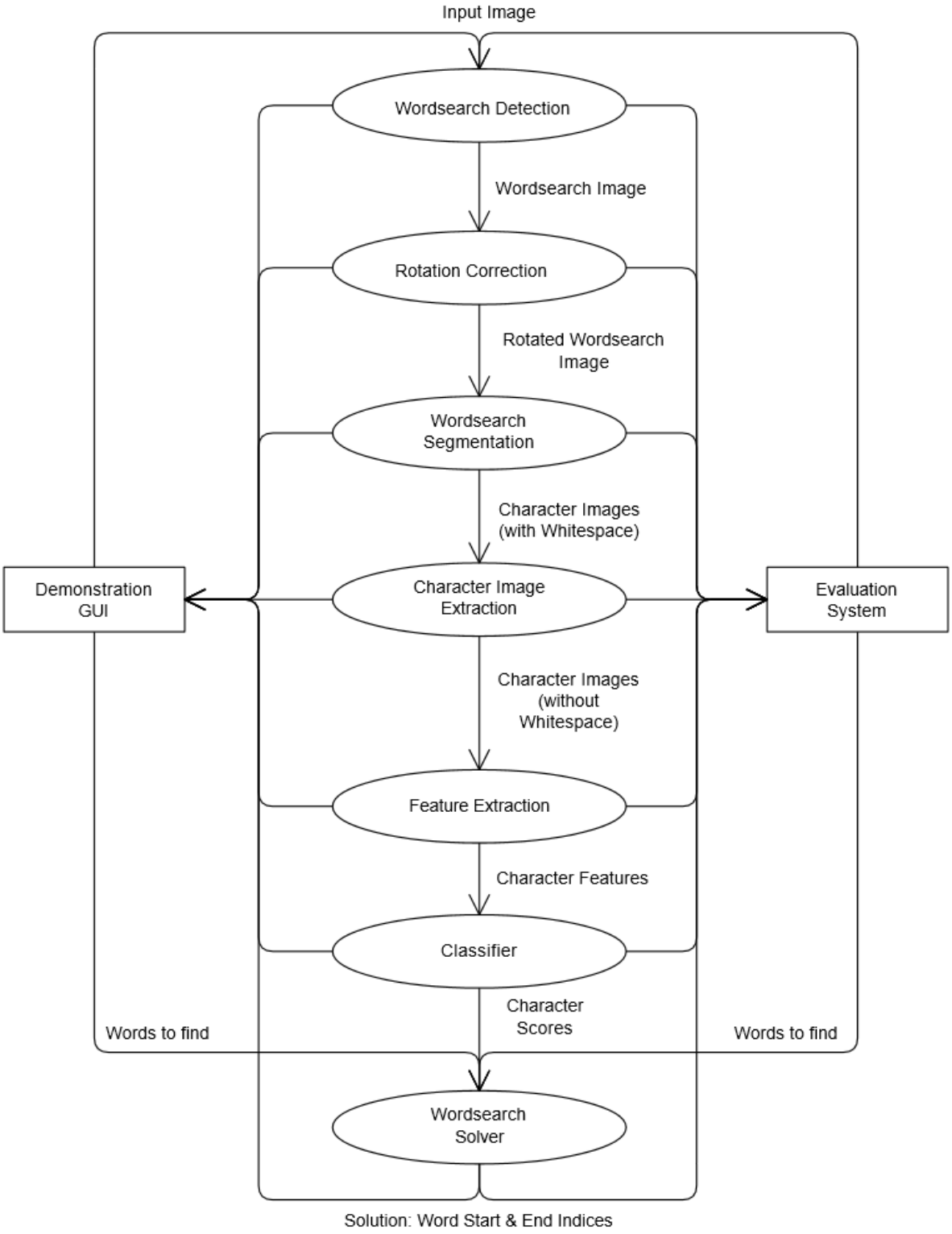


Figure 4.1 Data Flow Diagram for the Full System

When selecting what each module encompasses, care has been taken to ensure that each module represents the smallest portion of the system that should be evaluated independently of the rest of the system. In doing this, it will be possible to implement multiple solutions for each module and quantitatively evaluate their performance (both individually and as part of the larger system) before selecting the best approach to be used in the demonstration system.

By breaking the system up so much, it can lead to inefficiencies where there is potential for overlap between modules (e.g. the Rotation correction stage could need to segment the word search) and so ways for modules to share data beyond their own results will have to be considered at the Implementation stage. Even if there were to be some inefficiencies introduced into the system because of this, the benefits to the Evaluation (which in an experimental project such as this is more important than efficiency) should outweigh the additional processing demands.

4.2.1 Word search Detection

Input: Image containing a word search somewhere within

Output: Image containing only the word search grid

The aim of the Word search Detection module is to locate the word search in the provided image. Due to the varying sizes of word searches and the fixed aspect ratio of the camera being used, it would be impossible to take a picture that only ever contained the word search grid, as such the first step must always be to extract the grid before any further processing can take place.

4.2.2 Rotation Correction

Input: Image containing only the word search grid (in any of the four possible rotations)

Output: Image containing only the word search grid (in the correct rotation)

The Rotation Correction module performs the task of ensuring that the word search is rotated such that the bottom of the word search is aligned with the bottom of the image; this task is simplified by all of the word searches being considered in this project being rectangular. When the constraint on word searches being rectangular is combined with the knowledge that the input for this stage is an image containing only the word search and nothing else, then there are only four possible rotations of the input image that could produce the correct output (0°, 90°, 180° and 270°).

4.2.3 Word search Segmentation

Input: Image containing only the word search grid (in the correct rotation)

Output: Images of the characters making up the word search

The Word search Segmentation module must return an image of each character and its position in the grid (e.g. 3 in, 2 down). It does not have to necessarily filter out any artefacts or whitespace in the expected position of the character (or around the actual characters position, depending upon how the algorithm works), and each extracted character will not necessarily be in the exact centre of the image. Each character should be rotated such that the bottom of the character is in line with the bottom of the image, as this has already been done at the word search level (leaving only minor variations that may have to be corrected in later stages).

4.2.4 Character Image Extraction

Input: Images of the characters making up the word search (with surrounding whitespace and artefacts)

Output: Images of the characters making up the word search (without any surrounding whitespace or artefacts)

Character Image Extraction should take an image containing one character with some whitespace and potentially some artefacts and return an image containing only the character. Figure 4.2 shows an image that could be passed to the Image Extraction module with the area it would be expected to return highlighted.

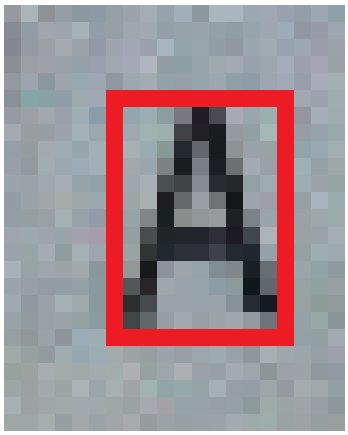


Figure 4.2 Character Image with the region to extract highlighted

4.2.5 Feature Extraction

Input: Images of the characters making up the word search (without any surrounding whitespace or artefacts)

Output: Character Features

The Feature Extraction stage will convert an image of a character to a collection of numerical values representing it. The simplest implementation of a feature extractor would be to return an array containing the pixel values in the image, however the features returned can have an effect on the performance of the classifier and so the simplest solution may not be best for this stage. Generally, the more features returned, the more training data will be required to learn the trends, and more CPU time will be required by the classifier to perform classification. For these reasons it may be desirable to return a small number of features. The features that get returned can also effect the output of the classifier; it is not necessary for the image to be perfectly reconstructed from the features so the feature extractor can discard a lot of the data in the image. A good feature extractor should select data that makes it easy to differentiate between different characters, e.g. a 'Z' has a diagonal line from the bottom left to the top right, but 'T' does not, so these classes could be separated by returning whether or not there is a line running diagonally from the bottom left to the top right of the image.

4.2.6 Classifier

Input: Character Features

Output: Character class scores

The Classifier must score the input for each possible class that the image could be (the 26 characters A-Z). How these scores are determined from the features input is completely up to the classifier, as is the range of values used.

4.2.7 Word search Solver

Input: Character class scores, Words to find

Output: Word search Solution (Row and Column indices for the beginning and end of each word)

The Word search Solver must find the location of each word to be found in the grid of characters and return a collection of these word positions (row and column indices for the beginning and end of each word) as the solution to the word search. This stage has a very simple solution (by simply scanning for each word in every possible position), but could be made more robust to errors made earlier in the system. E.g. account for character misclassifications or errors in the segmentation leading to additional rows and columns containing erroneous data in the grid.

4.3 User Interfaces

In addition to the core methods performing the individual stages listed previously, code must be written to use them. This will include:

- Quantitative Evaluation System
- Data Entry GUI (for marking up images with locations of word searches, the characters that make them up and any words to be found)
- Demonstration GUI
- Unit Tests

All of these separate projects will use the library containing the individual processing stages. They are all small, independent programs that do not require detailed designs to be specified at this stage.

4.4 Requirements Check

Before starting to implement any of this design, it must be checked against the requirements (as have previously been specified in Section 3.2). The non-functional requirements are concerned more with the quality of the finished product, as well as the conditions it functions in and so are largely unaffected by the design. The functional requirements must be checked off individually, this is done in Table 4.1.

Requirement	**Pass/Fail**
4.4.1.1 Mandatory	
Find a word search present in an image	Pass
Split an image of a word search into individual characters	Pass
Classify Characters	Pass
Automated system evaluation. Outputs accuracy scores for word search recognition, character classification and overall percentage of words found correctly	Pass
4.4.1.2 Desired	
Demonstration GUI to load an image for processing	Pass
Demonstration GUI to show the solved word search (grid of characters with lines drawn on the solutions)	Pass
4.4.1.3 Optional	
Demonstration GUI to manually adjust the bounding box for a word search found in an image	Fail – No GUI built for this, however the modular design allows for the detection stage to

	be overridden so this can be added
Progress bar for word search solver demonstration GUI, as it could take several seconds to process an image	Pass – Progress reported with stages being checked off once completed
Work on word searches in any orientation (so the picture could be taken upside down, or from the left or right)	Pass

Table 4.1 Functional Requirements Check

Overall, the design satisfies the requirements, only not meeting one of the optional requirements which were originally written to given the project extra scope and only to be done if time permitted.

Chapter 5 Implementation and Testing

This chapter goes through the implementation of a system that meets the design from Chapter 4 and discusses how it is tested. Firstly the tools (both software and hardware) that will be used to implement the system will be discussed. Then each module from the system design will have their implementations detailed. Finally, Section 5.9 details how the system was tested as it was implemented.

The design of the system (as laid out in Section 4.2) leaves the implementation of each module making up the core of the system to be as flexible as possible. This has been done in order to allow for several approaches to be trialled per module (where necessary) in order for them to be quantitatively evaluated against one another before one is selected for the final system. In this chapter, the specific algorithms implemented for each module will be discussed, complete with pseudo code. The modules being discussed and their sections are:

- 5.2 Word search Detection Implementation
- 5.3 Rotation Correction
- 5.4 Word search Segmentation
- 5.5 Character Image Extract
- 5.6 Feature Extract
- 5.7 Classifier
- 5.8 Word search Solver

In addition to these core modules there are executable pieces of software that need to implemented (as listed in Section 4.3) in order to run the system. Due to the limited space available in this document and their generic nature, details of their implementation will not be discussed. The system's source code should be used if the reader wishes to know any of the implementation details for these systems.

5.1 Tools

The tools used to implement and test the system must be decided upon before any work begins. This section will detail the tools selected and briefly give the reasons for choosing them.

5.1.1 Software

5.1.1.1 Libraries

Some available libraries that would be useful for this project have already been mentioned in Section 2.5 . I have decided to use the AForge.NET [10] Computer Vision and Artificial Intelligence library. The main reasons for choosing it are:

- It is "designed for developers and researchers in the fields of Computer Vision and Artificial Intelligence" [10], the main areas of this project.
- Its use of the LGPL v3 [30] open-source license that allows for the library to be freely used for both commercial and non-commercial applications [31].
- Its extensive documentation [32] (for a non-commercial software project)
- Its source code is freely available [33] (because it is open source), is well written and commented, meaning that the code can be read where documentation is not available or is missing the required information.

In order to provide some additional desired algorithms (AForge.NET lacks a good selection of Dimensionality Reduction algorithms), I have decided to also use Accord.NET [34] which is designed to be used in conjunction with AForge.NET and implements many common algorithms not found in AForge.NET. It shares all of the advantages of using AForge.NET thanks to using the same LGPL license, good documentation and open-source code (because of the LGPL license).

5.1.1.2 Programming Language

Whilst it can be possible to access methods of a library written in another programming language, this is often a problematic process that cause unnecessary development headaches and bloats the project codebase. It is therefore reasonable to limit the choice of languages to those that can directly use the selected libraries, or could use an existing wrapper that handles bridging the language barrier when using the library.

Both the AForge.NET and Accord.NET libraries that are being used by this project target the .NET platform. Due to the way .NET languages work (being compiled to Common Intermediate Language (CIL) and then run on an implementation of the Common Language Infrastructure (CLI) such as the Common Language Runtime (CLR)) these libraries can be used directly from any .NET programming language. The two best supported .NET languages are C# and Visual BASIC .NET. Putting aside any personal syntax preferences, there are a small list of key differences between the two languages [35]. Whilst many of the differences are really just adding syntactic sugar and not affecting the feasibility of the project, there is one that stands out: C# allows for what it calls unsafe code which "is required for any operation involving pointers" [36]. Having this low-level access to memory is particularly useful for this project as it gives highly efficient direct access to the raw contents of a Bitmap image, which will be required for several algorithms. There is a `GetPixel(int x, int y)` method that uses managed code to access a pixel (and would therefore be accessible from VB .NET and C#), however for each pixel it copies the full contents of the Bitmap to an array in memory and then selects the data for the specified pixel which is a very expensive operation and highly inefficient when performed for each of the pixels in an image. Due to C# providing direct access to memory through the use of pointers, C# will be the chosen programming language for this project.

5.1.1.3 Version Control System (VCS)

As this project is being developed just by one person, the typical reason for wanting to use a VCS (managing a code base being actively maintained by a team of developers) is not a problem, however some advantages of using a VCS still apply. In particular, the VCS will be used to maintain a history of the code and also to manage workflows through the Gitflow Workflow [37]. Whilst all VCSs maintain a history of the code, the second requirement means that a VCS with very cheap branching and merging is required; two commonly used (and therefore well supported) free VCSs meet this requirement: Git and Mercurial. Both are open source (licensed under GPL v2 [38]), actively maintained, have large active user communities and have free project hosting available online (including hosting for private repositories); therefore they can be considered equal for the purposes of this project and the choice becomes a matter of developer preference. I chose to use Git as I am more familiar with it.

5.1.1.4 Integrated Development Environment (IDE)

The two most popular IDEs for C# are Microsoft Visual Studio and the open-source MonoDevelop. Visual Studio has some advantages over its open source counterpart: code analysis, performance profiling, and code metric calculations. In addition to these features, Visual Studio has a more polished overall feel and also has a huge number of plugins adding a lot more functionality. MonoDevelop's main advantage is that it is cross-platform whereas Visual Studio requires a recent version of the

Microsoft Windows Operating System. In addition to these differences in the product itself, the Professional version of Visual Studio costs £506 (UK)/$499 (US), whereas MonoDevelop is free. There is a free version of Visual Studio (the Express edition), and as a student I get free access to Visual Studio Professional edition through Microsoft Dreamspark [39], however the cost should still be taken into consideration. It is possible to open the same solution files in both IDEs, so Microsoft Visual Studio Professional 2013 will be used, knowing that I have the option to switch to MonoDevelop at a later date.

5.1.1.5 Operating System

Due to the selection of Visual Studio 2013 as the IDE to be used for the development of this project, a recent version of the Microsoft Windows Operating System is required. Microsoft Windows 7 will be used.

5.1.2 Hardware

5.1.2.1 Camera

Apple's iPhone 4 has been selected as the input camera device because it is representative of the minimum capabilities of modern smartphones; a potential target platform for a commercial variant of this system. Full reasoning for this choice is explained in Section 3.3 .

5.1.2.2 Computer

The computer used for the development of this system will be a PC desktop with a 3rd generation Intel i5 quad core processor (based on the AMD64 architecture). The demonstration system will be designed to run on any modern PC with more than 256MB of RAM (due to the large number of images generated and stored in memory that are used to show the output of each stage visually).

5.2 Word search Detection Implementation

Word search detection was initially identified as being one of the main problems to be solved in this project, however as other areas proved more difficult than initially expected, time spent on word search detection was reduced. As such, there is a minimal solution implemented for this stage which assumes that all word searches are surrounded by a rectangular bounding box. This is the case for many of the word searches surveyed however not all word searches are bounded by a rectangle and so this stage will definitely not be able to find them.

This stage is made up of two smaller components and one method to bring them together. There is functionality to find quadrilateral blobs in the image, and also functionality to score how like a word search an image is.

Algorithm 5.1 finds quadrilaterals in a given Bitmap. This is used to find word search candidates, using the assumption that word searches will be bounded by a (quadrilateral) box.

```
1. Create a greyscale copy of the input Bitmap
2. Adaptive threshold the greyscale Bitmap (in place)
3. Perform Blob Recognition on the image, filtering out blobs smaller
   than the specified minWidth or minHeight
4. Initialise foundQuads = new List<List<IntPoint>>()
5. For each blob: If the blob approximately forms a quadrilateral,
   extract it's corner points and store them in foundQuads
6. Return foundQuads
```

Algorithm 5.1 Extract Quadrilaterals from Image - SharedHelpers.Imaging.ShapeFinder.Quadrilaterals(Bitmap, int, int)

Algorithm 5.2 scores a Word search candidate image. The model of a word search used to produce this score is its Segmentation (as generated by one of the algorithms defined in Section 5.4). The score is then calculated as the inverse of the average of the standard deviations of the row heights, column widths, gaps between rows and gaps between columns.

```
1. Segment the input image using the specified SegmentationAlgorithm
2. Instantiate a CandidateScorer object from the returned Segmentation
3. Fetch the WordsearchRecognitionScore from the CandidateScorer, by
   performing the following sub-routine:
      a. rowHeights = calculate the height of each row
      b. colWidths = calculate the width of each column
      c. rowGaps = calculate the gap between one row ending and the
         next starting
      d. colGaps = calculate the gap between one column ending and the
         next starting
      e. avgStdDev = (stdDev(rowHeights) + stdDev(colWidths) +
         stdDev(rowGaps) + stdDev(colGaps)) / 4
      f. return wordsearchRecognitionScore = (1 / avgStdDev)
```

Algorithm 5.2 Word search Candidate Scorer - SharedHelpers.ImageAnalysis.WordsearchDetection.CandidateScorer

Algorithm 5.3 uses algorithms Algorithm 5.1 and Algorithm 5.2 together to create a method able to extract a word search Bitmap (and it is coordinates in the original image) from a given Bitmap. It aims to return the single best word search candidate (if any are found) as this is desirable for the quantitative evaluation side of the system.

```
1. Find all quadrilaterals in image (of dimensions >= 10% image
   dimensions) using Algorithm 5.2.1.1 (Extract Quadrilaterals from
   Image)
2. If no quadrilaterals found, return null
3. Initialise bestScore = double.NegativeInfinity, bestBitmap = null
   and bestCoords = null
4. For each quadrilateral found:
      a. Create a Bitmap containing just that section of the input
         image (correcting for perspective)
      b. Score the Bitmap (for how like a word search it appears to be)
         using Algorithm 5.2.1.2 (Word search Candidate Scorer)
      c. If this Bitmaps score > bestScore, update bestScore,
         bestCoords and bestBitmap to the values for this Bitmap
5. Return the Coordinates of the best word search candidate, and the
   extracted Bitmap of it
```

Algorithm 5.3 Word search Candidate Selection - SharedHelpers.ImageAnalysis.WordsearchDetection.DetectionAlgorithm.ExtractBestWordsearch(Bitmap, SegmentationAlgorithm)

5.3 Rotation Correction Implementation

Once a Bitmap of a word search has been extracted, that Bitmap can have four possible orientations: 0°, 90°, 180° or 270° (clockwise). The correct orientation must be one of these possibilities because the word search is rectangular and the word search image (as returned by the Word search Detection stage) is object aligned (one of the four sides of the word search lines up with the bottom of the image). Rotation correction selects which of these orientations is the one that aligns the bottom of each character with the bottom of the image.

Algorithm 5.4 scores a candidate word search rotation, where the score is highest for the correct orientation. It works by classifying the image from each word search grid cell using a probabilistic classifier and summing the highest class probability for each grid cell. At the grid cell level, the probability will be highest when the cell contains a character that is correctly oriented (although for some letters, e.g. 'I' the orientation will not matter). The entire grid is summed to ensure that anomalies caused by single cells have a minimal effect on the end result.

```
1. Using the probabilistic Classifier specified, classify each cell in
   the word search grid as a character, storing the probabilities
   returned for each character for each cell.
2. Initialise score = 0
3. For each grid cell character probabilities
      a. Largest = find the highest character probability for this cell
      b. score += Largest
4. Return score
```

Algorithm 5.4 Score a candidate word search rotation – SharedHelpers.ImageAnalysis.WordsearchRotation.WordsearchRotationCorrection.scoreBitmap(WordsearchRotation, Classifier)

Algorithm 5.5 returns the correct WordsearchRotation given a WordsearchRotation (which can be in any of the four possible orientations). The WordsearchRotation object contains the Bitmap of the word search, as well as its Segmentation. The object exists to provide methods to deep copy the Bitmap and Segmentation as one method call, as well as code to rotate both at once.

```
1. rotations = Deep copy the specified WordsearchRotation 4 times,
   rotating them (0°, 90°, 180°, 270°) respectively

2. Initialise bestScore = double.MinValue, bestIdx = -1

3. For i = 0; i < rotations.length; i++

      a. score = score rotations[i] with Algorithm 5.2.2.1 (Score a
         candidate word search rotation)

      b. if score > bestScore: update bestScore = score, bestIdx = i

4. return rotations[bestIdx]
```

Algorithm 5.5 Word search Rotation Correction - SharedHelpers.ImageAnalysis.WordsearchRotation.WordsearchRotationCorrection.CorrectOrientation(WordsearchRotation, Classifier)

5.4 Word search Segmentation Implementation

Word search Segmentation algorithms take a Bitmap as their only input and return a Segmentation object as their output. The Segmentation object stores the image dimensions and the pixel indices in the image that are between each row/column. These indices will be used to split the Bitmap of a word search up into Bitmaps of individual characters. Figure 5.1 shows where segmentation points would be when drawn on top of the input word search image. Segmentation objects may also store additional information and provide further functionality (e.g. rotation so that if the word search image gets rotated, the Segmentation need not be recalculated from scratch).

Figure 5.1 Segmented Word search

Word search Segmentation has been one of the main focus points of this project, as such there are several algorithms implemented for this stage, each allowing for different approaches to be evaluated. In order for the different algorithms to be interchangeable elsewhere in the system, all segmentation algorithms extend the abstract class SegmentationAlgorithm, which defines one abstract method:

```
public abstract Segmentation Segment(Bitmap image);
```

There is also an Interface (called ISegmentationAlgorithmOnBoolArr) defined for segmentation algorithms that work on a `bool[,]` representing the pixels in the image, rather than directly on the Bitmap.

Then there is an abstract class for segmentation algorithms that calculate the start and end indices of each row and column of characters. This class is called SegmentationAlgorithmByStartEndIndices and it extends SegmentationAlgorithm. It implements the `Segment` method from SegmentationAlgorithm and provides a new abstract method `doSegment` whose returned values are used to create the Segmentation object required to meet the `SegmentationAlgorithm.Segment` method signature. `doSegment` method signature:

```
protected abstract void doSegment(Bitmap image, out int[,] rows, out int[,] cols);
```

All of the word search segmentation algorithms that made their way into the final library are children of SegmentationAlgorithmByStartEndIndices, with some implementing ISegmentationAlgorithmOnBoolArr. For each algorithm, it will be specified whether they operate on the bool[,] or Bitmap representation of the word search image. The full list of segmentation algorithms and the sections where they can be found is:

- 5.4.2 Manually Selected Single Threshold
- 5.4.3 Mean Single Threshold
- 5.4.4 Median Single Threshold
- 5.4.5 Percentile Two Thresholds

- 5.4.6 Histogram Single Threshold
- 5.4.7 Histogram and Percentiles Two Thresholds
- 5.4.8 Blob Recognition

Before the implementation of each of these algorithms is discussed, some of the methods that are common to several of them will be discussed separately in Section 5.4.1 . In addition to the individual segmentation algorithms, there is a method that can be applied to the result of any of the segmentation algorithms that aims to remove rows and columns in the Segmentation that are erroneously small. This process is detailed in Section 5.4.9 .

5.4.1 Shared Methods

Many segmentation algorithms work in similar ways, making use of the same methods. These methods will be explained here and referred back to later as they are used.

Algorithm 5.6 calculates the number of dark pixels per column in a given image. The pseudo-code here assumes the image has already been thresholded and single pixels evaluate to boolean values. In the actual implementation there is another function that does the same for rows, it will not be shown here because they are essentially the same.

```
1. Initialise colPixelCounts = new int[img.Width]
2. for i = 0; i < img.Width; i++
      a. colPixelCounts[i] = 0
      b. for j = 0; j < img.Height; j++
            i.  if img[i, j] //If this pixel is dark
                  1. colPixelCounts[i]++
3. return colPixelCounts
```

Algorithm 5.6 Count number of dark pixels per column - SharedHelpers.ImageAnalysis.WordsearchSegmentation.WordsearchSegmentationHelpers.CountNumDarkPixelsPerCol(bool[,])

Algorithm 5.7 takes the number of dark pixels per row (or column), along with a threshold of dark pixels to enter a row of characters and a threshold to exit. From this it calculates the start and end pixel indices for each row (or column) of characters. Many of the segmentation algorithms use this function, with their differences being how they select the thresholds. A simplified version of this algorithm that only used a single threshold was initially implemented, but the need for two thresholds was soon realised. Early segmentation algorithms using a single threshold now use this function, passing it the same threshold for both entry and exit.

```
1. Initialise chars = new List<uint[]>(), charStartIdx = 0, inChar =
   false
2. For i = 0; i < darkPixelsCounts.Length; i++
      a. if inChar:
            i.  if darkPixelCounts[i] < exitThreshold: //If leaving a
                character row/col
                   1. thisChar = { charStartIdx, I } //Store this
                      row/col start & end indices
                   2. chars.Add(thisChar)
                   3. inChar = false
      b. else if darkPixelsCounts[i] > enterThreshold: //If entering a
         character row/col
            i.  charStartIdx = i, inChar = true
3. if inChar && charStartIdx != (darkPixelCounts.Length - 1) //If
   ending in a character, and it didn't start on the previous row/col
   (which would yield a row/col of width 0 which isn't possible)
      a. thisChar = { charStartIdx, darkPixelCounts.Length – 1 }
         //Store this last row/col up to the edge of the image
      b. chars.Add(thisChar)
4. Convert chars to uint[,] and return
```

Algorithm 5.7 Find character row/column start and end indices – SharedHelpers.ImageAnalysis.WordsearchSegmentation.SegmentationAlgorithmHelpers.FindCharIndices(uint[], double, double)

5.4.2 Manually Selected Single Threshold

This segmentation algorithm uses Algorithm 5.6 to extract the number of dark pixels per row and per column. Histograms of this data (one of which is shown in Figure 5.2) were used to manually select a threshold value that would split the data into columns containing characters and those containing whitespace only. This threshold was then converted from a measure of number of pixels to a percentage of the number of pixels in each column (image height); doing this allows for the same threshold to be used for both rows and columns and the segmentation algorithm to work on images of any dimensions.

Once the number of dark pixels per row and column were calculated, this data was then fed in to Algorithm 5.7 (along with the pre-selected threshold value) which finds the start and end indices of the rows containing characters.

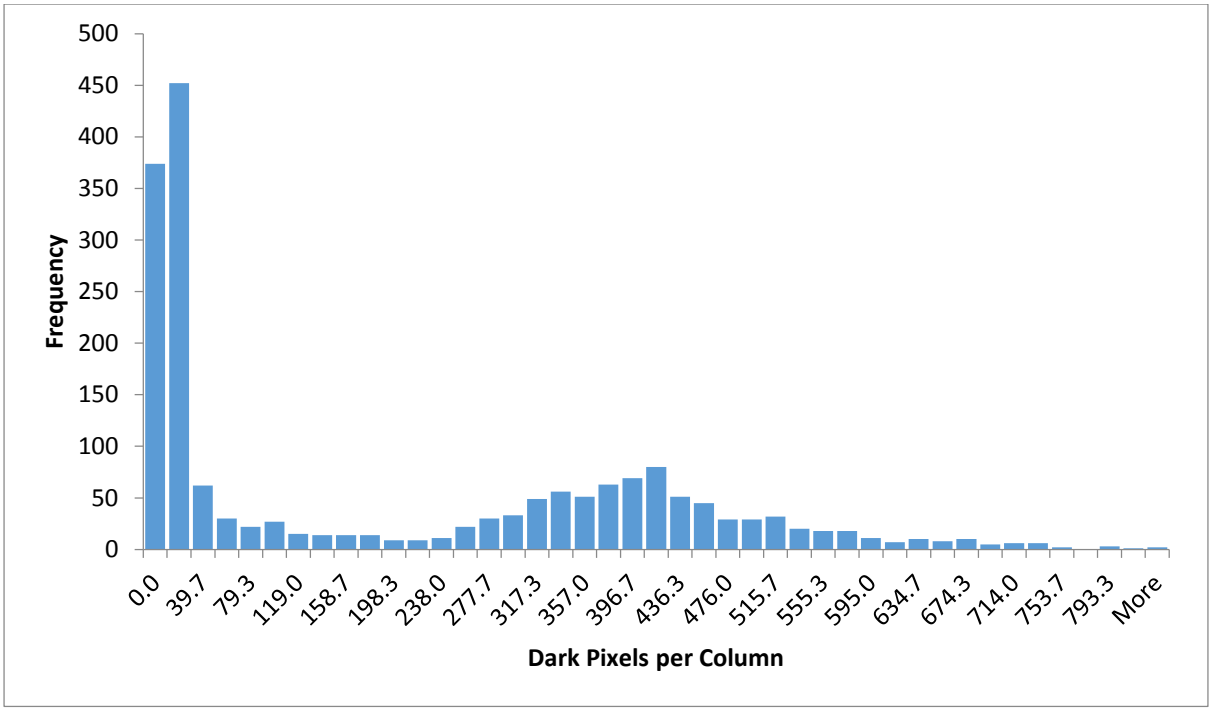


Figure 5.2 Histogram of Dark Pixels per Column in a thresholded Word search Image

5.4.3 Mean Single Threshold

This segmentation algorithm uses Algorithm 5.2.3.1 to extract the number of dark pixels per row and per column. It then calculates the mean number of dark pixels per row and per column and uses these values as the thresholds for Algorithm 5.7 to find the start and end indices of the rows containing characters.

5.4.4 Median Single Threshold

This segmentation algorithm is the same as the Mean Single Threshold, except it takes the median rather than the mean. It uses Algorithm 5.2.3.1 to extract the number of dark pixels per row and per column. It then calculates the median number of dark pixels per row and per column and uses these values as the thresholds for Algorithm 5.7 to find the start and end indices of the rows containing characters.

5.4.5 Percentile Two Thresholds

All segmentation algorithms up until this point suffer from over-segmentation (as shown in Figure 5.3) created by their use of a single threshold to detect both the start and end of each row and column. This algorithm explores the idea of using two thresholds as the input to Algorithm 5.7; one threshold for entering a row/column and another for exiting it. The idea here is to set the threshold to enter a column higher than the threshold to exit one. This prevents cases where a column of pixels may contain just enough to hit the single threshold and so it is recognised as a column of characters only to stop a few columns of pixels later when it drops below the single threshold again; once the higher entry threshold is hit you can be fairly certain that you have entered a column of pixels containing characters.

It also builds upon previous algorithms that select the threshold based upon the data collected from the number of dark pixels per row or column, except it uses percentiles. Percentiles allow for a greater

level of control over the threshold being chosen by the programmer, whilst still leaving the actual value to be determined from the data. If the 50th percentile were used for both thresholds, this algorithm would be equivalent to the Median Single Threshold segmentation algorithm (as specified in Section 5.4.4). However, the thresholds should be different to one another, with the start threshold higher than the end threshold. The values used by default in the actual implementation are the 60th and 40th percentiles respectively.

5.4.6 Histogram Single Threshold

This segmentation algorithm aims to replicate how a person would go about selecting a single threshold manually, as was done in the Manually Selected Single Threshold segmentation algorithm (as specified in Section 5.4.2) using the histogram in Figure 5.2. In order to do this, a class representing a histogram was created (called Histogram) and a sub-class of this was created to represent a bimodal histogram (called BimodalHistogram) which has the functionality to find the best place for a threshold to split the data with a distinct peak on either side.

In order to split the bimodal histogram the algorithm used in Otsu thresholding (discussed in Section 2.2) was used. Otsu thresholding works by essentially making a histogram of the image and then selecting a threshold that will split the image into two distinct groups (foreground and background); it is how Otsu selects this threshold from the histogram that will be applied here.

Once the threshold has been selected from the histogram, it will be passed to Algorithm 5.7 along with the number of dark pixels per row and column in order to find the start and end indices of the rows and columns containing characters.

5.4.7 Histogram and Percentiles Two Thresholds

This segmentation algorithm builds on the techniques from the two previous algorithms (Percentile Two Thresholds and Histogram Single Threshold) to create an algorithm that can select start and end thresholds based on the data from the input image.

Selecting two thresholds from the bimodal histogram using Otsu's method could be done by selecting the maximum and minimum values for the bin it selects as the best threshold, however this would not give any control over how much of the data fell inside the window between the two thresholds. Instead, the minimum and maximum thresholds should be chosen in a way that controls how much data will lie inside the window between the thresholds. In order to do this, the single threshold calculated from thresholding the bimodal histogram with Otsu's method has its percentile rank calculated (the percentile rank is the inverse of percentile, so if the 37th percentile of some data set is 120.5 then the percentile rank of 120.5 in the same data set would be 37). Once we know the percentile rank of a threshold, we can add x to it for the rank of the start value and subtract x for the rank of the end value. These ranks are then used to calculate actual threshold values with the percentile function.

Once the two thresholds have been selected, they are passed as arguments to Algorithm 5.7 along with the number of dark pixels per row and per column. This finds the start and end indices of the rows and columns containing characters. Algorithm 5.8 shows how this algorithm is implemented.

```
//Count the number of dark pixels per row and column
uint[] colDarkPixelCounts =
SegmentationAlgorithmHelpers.CountNumDarkPixelsPerCol(image);
uint[] rowDarkPixelCounts =
SegmentationAlgorithmHelpers.CountNumDarkPixelsPerRow(image);

//Calculate the Histograms to select a threshold
BimodalHistogram colHist = new BimodalHistogram(colDarkPixelCounts.ToDoubleArr());
BimodalHistogram rowHist = new BimodalHistogram(rowDarkPixelCounts.ToDoubleArr());

//Find the percentile rank corresponding to the threshold selected from the Histograms
Percentile colPercentile = new Percentile(colDarkPixelCounts.ToDoubleArr());
Percentile rowPercentile = new Percentile(rowDarkPixelCounts.ToDoubleArr());
double colThresholdRank = colPercentile.CalculateRank(colHist.ThresholdValue);
double rowThresholdRank = rowPercentile.CalculateRank(rowHist.ThresholdValue);

//Look up the percentile values for the ranks x either side of the auto-selected ones
double colStartRank = Math.Min(100, colThresholdRank + PERCENTILE_EITHER_SIDE);
double colEndRank = Math.Max(0, colThresholdRank - PERCENTILE_EITHER_SIDE);
double rowStartRank = Math.Min(100, rowThresholdRank + PERCENTILE_EITHER_SIDE);
double rowEndRank = Math.Max(0, rowThresholdRank - PERCENTILE_EITHER_SIDE);

double colStartVal = colPercentile.CalculatePercentile(colStartRank);
double colEndVal = colPercentile.CalculatePercentile(colEndRank);
double rowStartVal = rowPercentile.CalculatePercentile(rowStartRank);
double rowEndVal = rowPercentile.CalculatePercentile(rowEndRank);

//Determine the start & end indices of all the rows & cols, using the calculated start
& end threshold vals
uint[,] colChars = SegmentationAlgorithmHelpers.FindCharIndices(colDarkPixelCounts,
colStartVal, colEndVal);
uint[,] rowChars = SegmentationAlgorithmHelpers.FindCharIndices(rowDarkPixelCounts,
rowStartVal, rowEndVal);
```

Algorithm 5.8 Segment word search by Bimodal Histogram threshold selection and Percentile Rank to generate two thresholds - SharedHelpers.ImageAnalysis.WordsearchSegmentation.VariedRowColSize.SegmentByHistogramThresholdPercentileRankTwoThresholds.DoSegment(bool[,], out int[,], out int[,])

Despite the thresholds being selected in more and more advanced ways, there is still a danger of selecting unsuitable thresholds. Figure 5.3 shows the consequences of selecting an end threshold that is too low. When segmenting the image into rows, the number of dark pixels per row has gone above the entry threshold for the beginning of a row, but then not dipped back below the exit threshold until it has gone over 5 rows of characters. This image shows no signs of over-segmentation (as would be expected if the entry and exit threshold were too close together) so performance here would be better with a smaller difference in percentile rank chosen to add and subtract from the percentile rank of the threshold selected from the bimodal histogram.

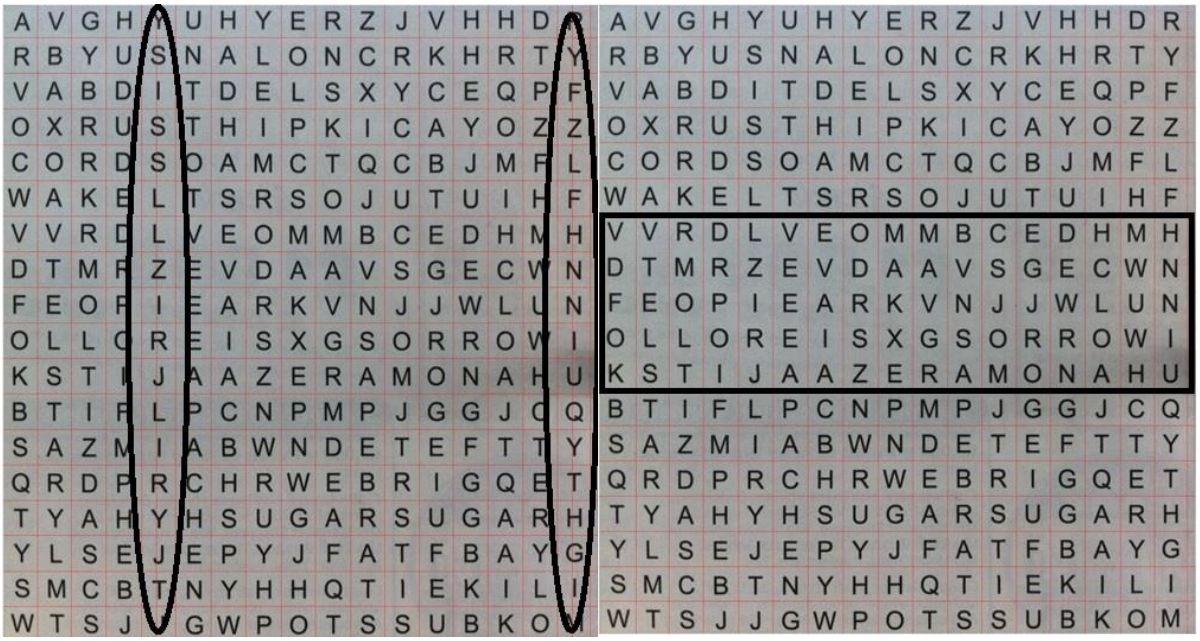


Figure 5.3 Examples of over-segmentation (left) and under-segmentation (right) in word search images. Segmentation problems are highlighted

5.4.8 Blob Recognition

Each of the previous segmentation algorithms work on the same basic idea, derived from the work of Hoffman and McCullough (discussed in Section 2.1). The limits of this method appear to be being reached, with the gains from each increasingly complex algorithm diminishing. Having learned from these previous algorithms, Blob Recognition is a new different approach to this problem.

Whilst the most computationally efficient solution to the problem should be working with the data at a pixel level, as had been attempted up until this point, it was proving to be challenging to ignore the noise that this low-level data includes. In order to solve the noise problem, I decided that an attempt should be made to work on the data at the highest level possible: the characters themselves. To do this, the size and location of each character in the input word search image would have to be identified, which is itself a challenge. The input image is made up of lots of blobs of varying shapes and sizes (Figure 5.4 shows all blobs found in a word search image); some of these will be characters, some background noise and the image may sometimes include the bounding rectangle that is used to find the word search in the original image, or any combination of its sides. It is possible that part of these sides will be occluded, meaning that they may appear to be the same width or height that you might expect a character to be. Importantly, even though each character in a given word search is always given the same amount of space on the page, their dimensions will vary between word searches due to the varying number of rows and columns. Because it is known that characters always come in a grid formation, it is alright if we are not able to identify all characters, as long as enough are identified to detect the pattern of rows and columns. It is therefore required that characters are detected with a high level of precision, but we can relax the requirements for recall considerably.

Figure 5.4 Word search Image with each recognised blob highlighted in a different colour

The process for finding characters in the given word search image is therefore to perform blob recognition with some sensible default minimum and maximum dimensions (e.g. blobs must make up at least 1% of the width and height of the image and at most 15%). Once these blobs have been extracted, the overwhelming majority of them should be characters. In order to remove the few that are not, the data set is cleaned up by removing characters whose width and height are either considerably smaller or larger than the mean width and height. Algorithm 5.9 shows the post-blob recognition filtering process that is the key to the high performance of this segmentation algorithm.

```
uint[] widths = new uint[blobs.Length];
uint[] heights = new uint[blobs.Length];

for(int i = 0; i < blobs.Length; i++)
{
    widths[i] = checked((uint)blobs[i].Rectangle.Width); //Throws OverflowException if
val < 0
    heights[i] = checked((uint)blobs[i].Rectangle.Height);
}

//Filter out blobs that are too big or too small
//Only remove blobs that are too big or too small in both dimensions, to allow for
thin chars like I's
double meanWidth = widths.Mean();
double minWidth = meanWidth * BLOB_FILTER_MEAN_MULTIPLIER_MINIMUM;
double maxWidth = meanWidth * BLOB_FILTER_MEAN_MULTIPLIER_MAXIMUM;
double meanHeight = heights.Mean();
double minHeight = meanHeight * BLOB_FILTER_MEAN_MULTIPLIER_MINIMUM;
double maxHeight = meanHeight * BLOB_FILTER_MEAN_MULTIPLIER_MAXIMUM;

List<Blob> chars = new List<Blob>();
foreach(Blob b in blobs)
{
    int width = b.Rectangle.Width;
    int height = b.Rectangle.Height;

    //Is this Blob allowable
    if((width <= maxWidth && width >= minWidth) || //Width check
        (height <= maxHeight && height >= minHeight)) //Height check
    {
        chars.Add(b);
    }
}
return chars;
```

Algorithm 5.9 Filter out blobs that do not appear to be characters – SharedHelpers.ImageAnalysis.WordsearchSegmentation.VariedRowColSize.SegmentByBlobRecognition.filterBlobs(Blob[])

Once we have the location and dimensions of the characters we can calculate where each row and column starts and ends. For columns this is done by grouping characters together that overlap in the x dimension (across the width of the image), then the highest and lowest x values from each group are used to determine start and end indices of each column. The same process is repeated in the y dimension in order to calculate row indices. The actual implementation of this idea is slightly different (as it is slightly more optimised); it does not actually cluster the characters, but rather searches through the existing data structure. Algorithm 5.10 is the optimised version of this algorithm as implemented. The same algorithm can be used to find the rows, except it needs to work in the y domain, using the y coordinates and the height of each blob's bounding box.

```
//Sort blobs left to right (by left edge co-ord)
Blob[] sorted = blobs.OrderBy(b => b.Rectangle.X).ToArray();

//Now find cols in the sorted blobs by looking for overlapping blobs
List<Tuple<int, int>> liCols = new List<Tuple<int, int>>();
int colStart = sorted[0].Rectangle.X;
int colEnd = sorted[0].Rectangle.X + sorted[0].Rectangle.Width - 1; //-1 because start
& end indices used for segmentation should be inclusive & the width makes it exclusive
of the end index
for(int i = 1; i < sorted.Length; i++)
{
    int start = sorted[i].Rectangle.X;
    int end = start + sorted[i].Rectangle.Width - 1; //-1 because start & end indices
used for segmentation should be inclusive & the width makes it exclusive of the end
index

    //If we have reached the start of a new col
    if(start > colEnd)
    {
        //Add the current col to list of cols
        liCols.Add(Tuple.Create(colStart, colEnd));

        //Set the column start & end values to that of this character
        colStart = start;
        colEnd = end;
    }
    else //Otherwise we're still in the same col
    {
        //Update the column end index if this character goes further to the right
        if(end > colEnd)
        {
            colEnd = end;
        }
    }
}
//We'll have finished in a column, add this col to the found cols
liCols.Add(Tuple.Create(colStart, colEnd));
```

Algorithm 5.10 Find columns in an array of Blobs – SharedHelpers.ImageAnalysis.WordsearchSegmentation.VariedRowColSize.SegmentByBlobRecognition.findRowsAndCols(List<Blob>, out int[,], out int[,])

5.4.9 Remove erroneously small Rows and Columns

This is not an implementation of a segmentation algorithm, it is an algorithm that can be applied to the result of running a segmentation algorithm. It aims to remove rows and columns from the Segmentation that appear to be too small to be actual rows or columns and are therefore erroneous. This is possible because of the fixed width and height of each row and column that makes up a word search. The result of this should be to reduce the problem of over-segmentation (as seen in Figure 5.3).

It works by identifying rows and columns in the Segmentation object that are too small to actually be rows and columns in the word search by calculating the mean row/column height/width and then removing those that are smaller than some multiple of the mean value (where the multiple is less than 1). In the actual implementation, the multiple of the mean value is set to 0.5. Algorithm 5.11 shows the code responsible for this.

```
List<int> keepIndices = new List<int>();
double threshold = meanColWidth(colStartEnds) *
REMOVE_SMALL_COLS_THRESHOLD_MEAN_MULTIPLIER;

//Determine which cols to keep
for (int i = 0; i < colStartEnds.GetLength(0); i++)
{
    int width = colStartEnds[i, 1] - colStartEnds[i, 0];

    if (width > threshold)
    {
        keepIndices.Add(i);
    }
}

//Store the indexes we decided to keep
int[,] toRet = new int[keepIndices.Count, 2];
for (int i = 0; i < toRet.GetLength(0); i++)
{
    toRet[i, 0] = colStartEnds[keepIndices[i], 0];
    toRet[i, 1] = colStartEnds[keepIndices[i], 1];
}
return toRet;
```

Algorithm 5.11 Remove rows/columns that are smaller than half the mean value – SharedHelpers.ImageAnalysis.WordsearchSegmentation.Segmentation.removeSmallCols(int[,])

5.5 Character Image Extraction Implementation

The character image extraction algorithm works by simply extracting the biggest blob in the image. Once the biggest blob has been extracted in to a new image, it is resized to a constant width and height so that the input for the next stage can expect a consistent input size.

Extracting the biggest blob is highly effective and very simple, it is only downfall is when it is extracting the blob for the character in a corner of the word search and if both sides of the bounding box are included in the image. If this is the case then the biggest blob will be the bounding box rather than the character.

It may be possible to improve on this algorithm in the future by calculating the area contained inside the blob using the Shoelace algorithm [40], and then selecting the blob with the largest contained area. This would select the character rather than the two sides of the bounding box, because the bounding box lines would lead to a very small area contained in the blob, despite its larger overall width and height. However, the decision was made to not spend more time on this stage as this is a minor problem that can only ever effect single characters in the corners of a word search, and these single character errors should be made up for by probabilistic Word search Solver algorithms (Section 5.8).

5.6 Feature Extraction Implementation

Feature Extraction algorithms are implemented as sub-classes of the abstract FeatureExtraction class that defines the abstract method:

```
public abstract double[][] Extract(Bitmap[] charImgs);
```

It takes an array of thresholded Bitmap images that are the resultant images of the previous stage (only containing the character, and all the same dimensions). It returns an array of double arrays, each containing the feature values for one image. The range, number of features and how they are calculated depends on the algorithm.

5.6.1 Pixel Values

This first feature extraction algorithm is extremely simple and is not expected to perform particularly well; it is implemented purely as a baseline for more advanced feature extraction techniques to be evaluated against. The value of each pixel is mapped one-to-one onto the values to be returned, i.e. each pixel is used as a feature. Pixels are thresholded so there can only be two possible values and they are mapped so that if the pixel is dark it becomes 0.5 and if it is light it is -0.5. Any values could have been assigned to true or false, values around zero were selected due to a slight performance gain with the specific classifier being used and I decided to place the values fairly close to zero because there are higher levels of floating point precision around zero due to the way floating point numbers are stored [41].

5.6.2 Discrete Cosine Transform

As previously discussed in Section 2.3 , DCT is a widely used feature extraction algorithm and it should offer reasonable performance for this project. The Accord.NET framework implements both standard DCT and 2D DCT (as used in lossy image compression algorithms such as JPEG). It expects a 2D double array as its input and so this is generated in the same way as the Pixel Values algorithm, except it stores the values in a 2D array with the same dimensions as the image. Once the 2D DCT algorithm has been run, its output is converted to a single dimension double array for each character image to be used as the input for a classifier.

5.6.3 Principal Component Analysis

PCA has previously been discussed in Section 2.3 and should be more than good enough for the relatively simple classification problem here. PCA requires that the data set be learned before it can actually preform feature extraction on any data. In order to accommodate for the possibility of adding additional feature extraction algorithms that required training, an abstract class called TrainableFeatureExtractionAlgorithm was written that extends FeatureExtractionAlgorithm and implements the ISerializable interface. The ISerializable interface is implemented in order to allow for a final demonstration system to be shipped without requiring the training data, instead the trained PCA feature extractor's values must be able to be saved to and loaded from the file system. The most important methods this class therefore adds to the base FeatureExtractorAlgorithm class are:

```
public void Train(Bitmap[] charImgs)
public static TrainableFeatureExtractionAlgorithm Load(string filePath)
public void Save(string filePath)
```

The PCA feature extraction algorithm is a concrete implementation of the TraininableFeatureExtractionAlgorithm, with methods for training, extracting features, and saving/loading to/from the file system. In addition to the standard constructor, it also offers one with a parameter for the number of dimensions the feature extraction method should return. Should an object be instantiated with this parameter specified, the top *n* features will be returned rather than all of them.

The PCA algorithm (as implemented by Accord.NET) takes an array of double arrays, with each double array containing the pixel values of one character image. Each pixel's value is mapped in the same way as is done for the Pixel Values feature extraction algorithm (true -> 0.5, false -> -0.5).

It is worth noting that the Accord.NET PCA implementation has the option to normalise the training data by either subtracting the mean or subtracting the mean and dividing by the standard deviation. The method selected for use in this system is to just subtract the mean as due to the discrete input values, the standard deviation should be a fairly useless metric for normalising the data set.

5.7 Classifier Implementation

The classifier selected for the system is a single layer Neural Network using a Bipolar Sigmoid function for node activation and trained with Backpropagation learning. There is a network trained for each of the different feature extraction algorithms (as the network must learn the different features each extraction algorithm produces separately); in addition to this there must also be some networks that work with a reduced number of dimensions as can be produced by PCA (e.g. top 20 dimensions). The networks may learn very different weights, and have different numbers of inputs, but they will all have 26 outputs (one per class) and they will all be trained in the same way.

5.7.1 Training

In order for neural networks to produce the desired outputs and therefore be useable as a classifier they must first be trained to do so. The learning algorithm will adjust the weights of each neuron based on how close its output was to the desired output, but this must be done repeatedly in order to make the network increasingly accurate. Simply choosing a number of iterations and running the learning algorithm that many times is not good enough because it may be run too many times, over-learning the data (where the classifier learns to recognise minor details in the training data rather than the overall trends leading to a decline of performance on real data). Another possibility is that the network could not have finished learning the data and performance would be poor on both training and real data.

In order to tell when the network has learned the training data, we can measure how close the outputs the network gave were to the perfect/desired outputs we are using. On every iteration, we now continue only while the error value is below some constant and the iteration number is less than the maximum number of iterations.

Preventing the network from over-learning the data is slightly more complex. As previously discussed, the data has been split into three sets (Training, Cross-Validation, and Evaluation). Only the training data set has been used so far to train the classifier, but only the evaluation data set will be used to evaluate it; this leaves the cross-validation data set free to be used to aid the training process. The cross-validation data will never be used to actually train the classifier, so it can be used to benchmark performance on actual data for each training iteration. So whilst the error to the training data will continue to drop (supposedly showing increased performance), the performance of the network on the cross-validation data should begin to plateau (and then even drop off as the network over-learns the training data). The network can therefore be prevented from over-learning the data by measuring the network performance on the cross-validation data during each iteration and monitoring for a drop in performance. If the trained network from the previous iteration is stored, as soon as performance drops off, the previous network (from before the drop off in performance) can be used and training stops. This is used in Algorithm 5.12 to train a neural network.

```
1. Initialise prevNetwork = null, prevNetworkNumMisclassified =
   uint.MaxValue, prevNetworksNumMisclassified = new Queue<uint>(),
   iterNum = 1
2. prevNetworksNumMisclassified.Enqueue(uint.MaxValue) n times, where n
   is the number of previous iterations to check to see if performance
   has plateaued
3. Initialise neuralNet to a new network
4. Randomise neuron weights in neuralNet
5. prevNetwork = DeepCopy(neuralNet)
6. do:
      a. Run an iteration of training on neuralNet with the training
         inputs and expected outputs
      b. error = sum of: difference between neuralNet output and
         desired output squared, then divided by two
      c. networkNumMisclassified = evaluate network on cross-validation
         data
      d. if networkNumMisclassified >
         prevNetworksNumMisclassified.Mean(): //Check if performance on
         cross-validation data has dropped off
            i. neuralNet = prevNetwork //Use the previous network
           ii. break
      e. prevNetwork = DeepCopy(neuralNet)
      f. prevNetworkNumMisclassified = networkNumMisclassified
      g. prevNetworksNumMisclassified.Dequeue();
         prevNetworksNumMisclassified.Enqueue(prevNetworkNumMisclassifi
         ed)
      h. if prevNetworksNumMisclassified.Distinct().Count() == 1:
         //Check if performance has plateaued on cross-validation data
            i. break //Use this network
      i. if iterNum > MAX_LEARNING_ITERATIONS:
            i. break
      j. iterNum++
   while(error > LEARNED_AT_ERROR)
7. return neuralNet
```

Algorithm 5.12 Train a Neural Network – QuantitativeEvaluation.EvaluateNeuralNetworks.evaluateSingleLayerActivationNetworkWithSigmoidFunctionBackPropagationLearning(double[][], double[][], double[][], char[], double[][], char[], double, string)

It is still possible for networks to become stuck in localised solution that may not necessarily be the best solution to the classification problem globally. This is because we do not try all possible combinations of network weights, but rather optimise them from some initial values. In order to reduce the chances of this being a problem, many networks will be trained using different starting neuron weights (chosen randomly) and once trained they will be evaluated on the cross-validation data with the one that performs best being selected for use in the final system. As more and more networks are trained, the chance of them all getting stuck in a localised solution approaches zero. 10,000 neural networks will be trained for each feature extraction algorithm meaning we can be fairly confident in the weights used representing the global solution.

5.7.2 Usage

In order to make the classifier as simple to use as possible, the decision was made to encapsulate both the classifier and feature extractor in an object whose sole purpose is to wrap them both up, providing a single method to classify an array of character image Bitmaps. This makes classifying character images as simple as loading the trained classifier and feature extractor (if it requires training) from the file system and then calling the classify method. E.g.:

```
        FeatureExtractionAlgorithm featureExtractor =
TrainableFeatureExtractionAlgorithm.Load("pcaAllFeatures.featureExtraction");
        Classifier classifier = new
AForgeActivationNeuralNetClassifier(featureExtractor, "pcaAllFeatures.neuralnet");
        double[][][] charProbabilities = classifier.Classify(charImgs);
```

5.8 Word search Solver Implementation

If the input to this stage is assumed to be correct then the solution here is trivial; this extremely simple solution is implemented for benchmarking purposes as the Non-Probabilistic Solver. However, none of the previous stages will produce the perfect output in every possible case, meaning that this final stage should also attempt to work with bad data to still find the best solution. Obviously this will not always be possible, but multiple algorithms have been devised and implemented to attempt to allow for errors in the earlier stages.

5.8.1 Non-Probabilistic Solver

The Non-Probabilistic word search solver is very simple, it takes a 2D array of characters and an array of strings that are the words to be found and then looks for the words in the word search. Because the input for this stage is actually a double[] per character rather than a char, the first thing that must be done is to convert the scores per character to actual characters. This is done by iterating over each grid position and selecting the character that scored highest in that position. The pseudo-code for this algorithm is in Algorithm 5.13.

1. Initialise chars = new char[networkOutputs.Length, networkOutputs[0].Length]
2. For i = 0; i < networkOutputs.Length; i++ //Col
    a. For j = 0; j < networkOutputs[i].Length; j++ //Row
        i. Initialise maxIdx = 0 //Find the character with the highest score
        ii. For k = 1; k < networkOutputs[i][j].Length; k++
            1. If networkOutputs[i][j][k] > networkOutputs[i][j][maxIdx]
                a. maxIdx = k
        iii. chars[i, j] = (char)('A' + maxIdx) //Convert the index (0..25) to a char (A..Z)
3. return chars

Algorithm 5.13 Convert classifier outputs to 2D char array - SharedHelpers.ClassifierInterfacing.NeuralNetworkHelpers.GetMostLikelyChars(double[][][])

The words are then found in the word search by simply iterating over every grid position and trying to fit each word in each of the eight possible directions that it could go in.

5.8.2 Probabilistic Solver

The main advantage of using a probabilistic classifier is to allow for this stage to use the probabilistic outputs (discussed in Section 3.1.5). The way this is done is that for each word, every position in the word search grid is tried as a starting position, and each of the eight possible directions will yield a possible placement for that word (except when the word cannot fit in that direction because the starting position is too close to the edge). Now for every possible word position, a score can be calculated by summing the classifier scores (at the used grid position) for each character that makes up the word. We know that the word search must contain the word we are looking for, so the solution for each word is the one that scores the highest.

5.8.3 Probabilistic Solver, preventing character discrepancies

A problem with the Probabilistic Solver is that it is possible for two words to use the same grid position as different characters; in reality of course it is impossible for a grid position to be two characters at once. This discrepancy between the usages of that character and the expected character in the words using it must somehow be detected and resolved. The solution to this is to generate a score for each word in each position (as was done in the Probabilistic Solver). Each word will now have a list of possible positions and scores associated with them. The aim is now to find the combination of word positions that scores highest (where the combination score is the sum of the individual word scores), with all words having been placed and no two words using a single grid position as different characters. Searching all possible combinations would be a classic example of an intractable problem, so some heuristic method must be devised to return what is likely to be the correct solution in a timely manner.

A breadth-first state space search is used to determine the order in which candidate solutions are evaluated. Each candidate solution is a combination of word positions. The order in which word positions are tried is based upon their score, with the position scoring highest going first. So the starting state (root node of the search) is generated by selecting the position with the best score for each word (i.e. the solution generated by the Probabilistic Solver).

In order for the state space search to proceed, there must be a method for checking if a solution is valid (i.e. no two words use a single grid position as different characters), and a method for generating the successors (next level in the tree of candidate solutions) for a given node. Both of these depend on knowing where any discrepancies/collisions have occurred; Algorithm 5.14 generates a “Collision Table” (a lookup table for each cell in the word search that shows each word using it and what character that word uses it as), implemented as a 2D array of dictionaries where a string (the word) implies a char (the character the word uses this position as). Once the Collision Table has been generated, the locations of any collisions are identified by Algorithm 5.15. Checking if a solution is valid is now as simple as checking if the list of collisions contains any elements. Generating the successors to an invalid node is done by trying different combinations of word positions for words involved in collisions; this is done in Algorithm 5.16.

```
//Build up a representation of the Wordsearch and what characters each word uses
Dictionary<string, char>[,] collisionTable = new Dictionary<string, char>[cols, rows];
… <Removed initialisation loop for space>

//Iterate over each word, populating the wordsearch
foreach(KeyValuePair<string, WordPosition> kvp in wordPositions)
{
    string word = kvp.Key;
    WordPosition position = kvp.Value;

    //Foreach character the word goes through
    int charIdx = 0;
    foreach(IntPoint charPosition in position.CharIndices)
    {
        //Store that this word goes through this character
        collisionTable[charPosition.X, charPosition.Y].Add(word, word[charIdx]);

        charIdx++;
    }
}

return collisionTable;
```

Algorithm 5.14 Generate Collision Table - SharedHelpers.WordsearchSolver.SolverProbabilisticPreventCharacterDiscrepancies.generateCollitionTable(Dictionary<string, WordPosition>, int, int)

```
for(int i = 0; i < collisionTable.GetLength(0); i++) //Cols
{
    for(int j = 0; j < collisionTable.GetLength(1); j++) //Rows
    {
        Dictionary<string, char> words = collisionTable[i, j];
        //If there is more than one word making use of this character, check that they
are all using it as the same character
        if (words.Count > 1)
        {
            char? firstChar = null;
            foreach (char c in words.Values)
            {
                //If first iter, store char
                if (firstChar == null)
                    firstChar = c;
                else //Otherwise this is a later iteration, check that the char this
word is using this position as is the same as previous words
                {
                    //If this word is using this charcter differently to how the first
word used it, there's a collision
                    if (c != firstChar)
                        collisions.Add(new IntPoint(i, j));
                }
            }
        }
    }
}
```

Algorithm 5.15 Find Collisions - SharedHelpers.WordsearchSolver.SolverProbabilisticPreventCharacterDiscrenancies.findCollisions(Dictionary<string, char>[,])

```
List<Tuple<int, Dictionary<string, int>>> successors = new List<Tuple<int,
Dictionary<string, int>>>();

//At each place there is a collision
foreach (IntPoint collisionPoint in collisions)
{
    //Try moving each word involved in that collision to every combination of their
positions
    //  up until one higher than the current position
    foreach (string word in collisionTable[collisionPoint.X, collisionPoint.Y].Keys)
    {
        //The Word Position index currently being used for this word
        int currentPositionIdx = selectedPositions[word];

        for (int newPositionIdx = 0; newPositionIdx <= currentPositionIdx + 1;
newPositionIdx++)
        {
            //Don't repeat the current position index. Saves some work for vet
successors
            if (newPositionIdx != currentPositionIdx)
            {
                //Clone the node
                Dictionary<string, int> nextSelectedPositions = new Dictionary<string,
int>(selectedPositions);

                //Update the selected position for this word
                nextSelectedPositions[word] = newPositionIdx;

                //Add this node to the successors
                successors.Add(Tuple.Create(level + 1, nextSelectedPositions));
            }
        }
    }
}
```

Algorithm 5.16 Generate Node Successors – SharedHelpers.WordsearchSolver.SolverProbabilisticPreventCharacterDiscrepancies.evaluateNode(Tuple<int, Dictionary<string, int>>, Dictionary<string, List<Tuple<double, WodPosition>>>, int, int)

Now that methods exist to find collisions, check solutions and generate successors, the breadth-first search can be performed. Usually in a state space search you are interested in returning the first valid solution, however because some solutions can be better than others with this problem (using the scoring system defined previously in this section) I decided to always evaluate complete levels of the tree and then pick the best solution (if there are any) rather than just returning the first one found. This means that if at level three of the tree there were to be two solutions; one with some words in incorrect places (but still valid because there are no collisions) and the other to be the correct answer, instead of just returning whichever the search reaches first, it will find both and then evaluate them against one another before returning the better solution.

In the event of the word search solver receiving really bad data (as would be the case if the word search were not found properly in the Word search Detection stage, Section 5.2) then no amount of searching can ever find the correct answer (although it might find a valid answer, i.e. one with no collisions). In these cases the system must detect when a solution is beyond repair. State space search allows for a very simple way of doing this; a maximum depth is specified and the algorithm will not generate any successor nodes for deeper into the tree than this level. If the search has not found any valid solutions before reaching this level (by which time it may have still checked tens of thousands of candidate solutions) then it is safe to call the search off. When this is the case the system returns the

best invalid solution (the one that scores highest, i.e. the same one that would have been returned by the Probabilistic Solver). This is so the system always returns its best answer for evaluation purposes; if this were to be used in some sort of production system then it could instead notify the user that it had failed to solve the word search due to unrecoverable errors made in previous stages.

5.9 Testing

Testing for this project has been largely informal (as is often the case for research projects), with methods being tested interactively with both perfect (made up) and real world data collected from images of word searches. However, many methods and objects have been unit tested where I felt it was necessary to ensure their correct operation.

Unit Testing has aimed to cover the parts of the system where the results being incorrect would be very difficult for a person to notice from the final system or through informal testing. This has helped greatly to eliminate many deep bugs that would probably otherwise have made it into the final system unnoticed. Code coverage analysis shows that 78.47% of the blocks of code in the BaseObjectExtensions library are unit tested, and 22.32% of the blocks of code in the SharedHelpers library are unit tested. The SharedHelpers library figure is slightly misleading, with the majority of namespaces within it having 0% coverage, but namespaces likely to contain deep bugs having fairly complete test sets, e.g. the Maths and Maths.Statistics namespaces both have more than 80% code coverage.

Chapter 6 Evaluation, Results and Discussion

In this chapter the performance of various parts of the system will be evaluated and discussed. For each part of the system being evaluated, the motivation for performing the evaluation will be explained, as well as how the evaluation will be carried out and any limitations that this may place on the evaluation and its results, what the data set will be and finally its results and any conclusions that can be drawn from them.

Data has been collected for the evaluation as described in Section 3.3 . This gives three data sets: training, cross-validation and evaluation data. The evaluation data will generally be what is used here, however in some cases (where the module being evaluated hasn't been trained on any data) it may be possible to use data from the other data sets. Where this is the case, the use of the additional data will be explained. The data sets are of images containing word searches; each image is marked up with the location of each word search, its number of rows and columns, each character that makes up the word search and the words to be found in it.

The parts of the system to be evaluated and the sections they are evaluated in are:

- 6.1 Word search Detection Evaluation
- 6.2 Word search Segmentation Evaluation
- 6.3 Feature Extraction and Classification Evaluation
- 6.4 Rotation Correction Evaluation
- 6.5 Full System Evaluation
- 6.6 Post-Detection Evaluation

6.1 Word search Detection Evaluation

The word search detection module (as implemented in Section 5.2) will be evaluated here. This modules utilises the word search segmentation module (as implemented in Section 5.4) for which there are several different implementations. The main aim of evaluating this stage of the system (in addition to measuring its performance) will be to select which segmentation algorithm the word search detection system uses in later evaluations.

The word search detection system will always produce one of the following two possible outputs:

- Null – signifying it could not detect anything resembling a word search in the input image
- Four coordinates – if any quadrilaterals were found in the image, it will calculate a score for each image representing how much each image appears to be a word search and then return the coordinates of the candidate that appears to be most like a word search

Due to these two possible results, this detection system doesn't function as a standard detection system would be expected to (it assumes there is always one word search to be found, meaning it will not return all candidates, and may return a candidate even if it does not match very closely with the model of a word search). This means that there is no information to be gained from evaluating it on images that contain no word searches in order to see if it finds false positives.

6.1.1 Evaluation Data and Method

The evaluation of this module will be done using images that contain one word search for which the coordinates of its corners are known. Each image will be passed to the module, and the returned coordinates will be compared to the known correct solution. In the event of no coordinates being returned, the system will have failed to find the word search in that image. There is no one correct solution for the coordinates, with the known correct solution being just one possible solution. However, each coordinate in a correct solution should be relatively close to one of the coordinates of the known correct solution. Also, the candidate and known correct coordinates should map one-to-one onto one another, i.e. if the points are ordered 1, 2, 3, 4 in the correct solution then a correct candidate solution must contain points all of these points, not having detected one corner twice and therefore missing part of the word search.

In order to allow for the returned solution to deviate from the known solution, the Euclidean distance between each coordinate pair that map onto one another in the two solutions (i.e. top-left to top-left and top-right to top-right etc…, or top-left to top-right and top-right to bottom-left etc… and so on for each of the four possible coordinate pair mappings) is measured and then checked to see if it is less than the acceptable distance between points. The acceptable distance between points in the return solution and the known correct solution is defined as being the mean dimension size of the mean cell in the word search (using the image coordinates and number of rows and columns from the known correct solution).

Due to the word search detection system's dependence on the word search segmentation system, it will be evaluated using each of the word search segmentation systems. For each word search segmentation system used the evaluation will be run twice, once using the output of the standard algorithm and once using the Remove Erroneously Small Rows and Columns algorithm (as implemented in Section 5.4.9) in addition to the standard algorithm. This will show how the use of the word search model (which is what the segmentation algorithms are being used as) effects the performance of the overall module.

The evaluation data set that will be used to evaluate this module will not be the standard evaluation data set. This is because this module has not been trained, or previously had its performance measured on any of the data (training, cross-validation or evaluation) meaning that all of these data sets can be used for evaluating it. Using this extra data increases the size of the evaluation data set without altering its quality, leading to a higher confidence in the results. Images in any of these data sets that contain more than one word search will be removed so that there always be one word search to be found in each image. This leaves a data set of 77 images, each containing one word search to be found.

6.1.2 Results

Full results are provided in Appendix A. (see table A-1). Figure 6.1 graphs the percentage of word searches detected correctly by the word search detection system for each of the available segmentation algorithms.

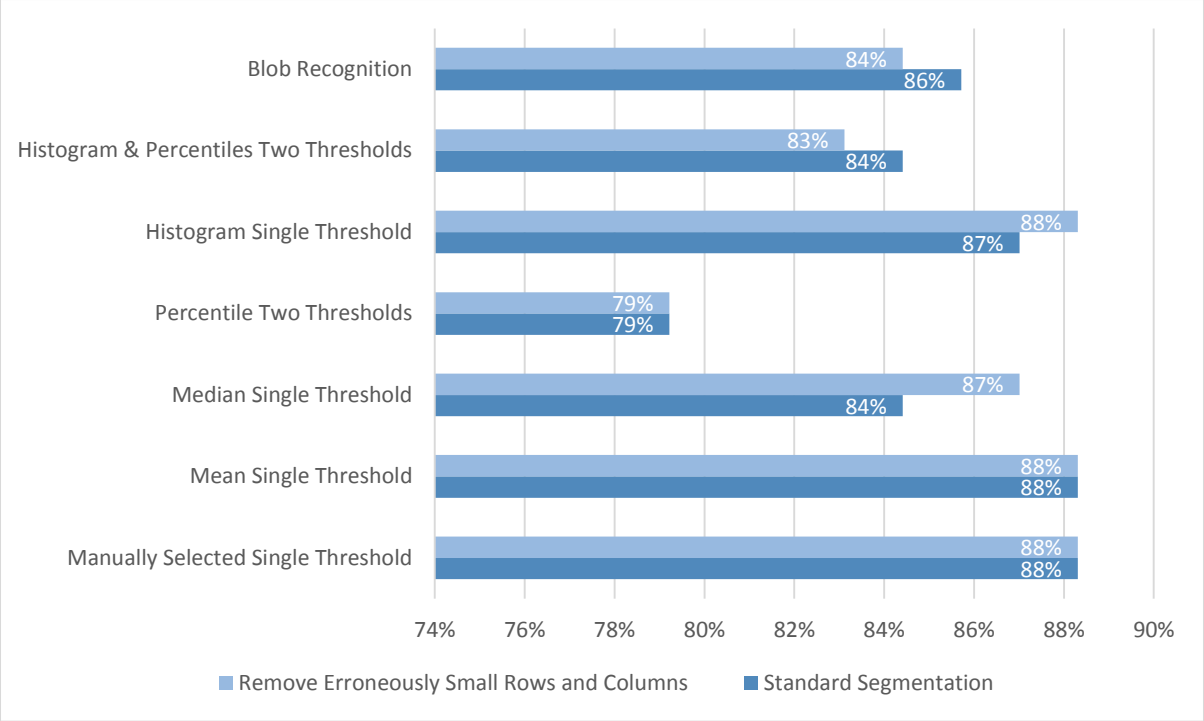


Figure 6.1 Word search Detection Performance - Percentage of word searches detected correctly. Performance broken down by segmentation algorithm (which is used as a model of a word search) and whether the calculated segmentation then has erroneously small rows and columns removed. Raw data in Appendix A. (see table A-1).

6.1.3 Discussion

The evaluation of this module has multiple limitations. Firstly, how changing the acceptable distance between each point in the candidate solution and the known correct solution effects the evaluation is unknown. It is possible that the current acceptable distance is either too large or too small. If it is too large, it may lead to some solutions that should have been rejected actually being accepted and if it is too small then it may lead to valid solutions being rejected. Further work is needed to measure the impact of this variable on the results of this evaluation.

If further work were to be done on this module, more detailed evaluation would be required. Separate evaluations could be done for the candidate ranking part of the system (i.e. does the candidate that is actually a word search get ranked as being most like a word search), and for the candidate selection part of the system. For the candidate selection part of the system, there could be the simple test of is the correct solution one of those found, and then there could also be more advanced measures such as precision that would penalise the system for finding candidates that are not word searches. This further evaluation work would highlight the part of the detection system that requires the additional work.

It is also worth noting that this evaluation was designed to work with a limitation of the implemented module. The module only works on word search images that are surrounded with a rectangular bounding box. Therefore, all of the evaluation data used contains word searches surrounded by bounding boxes; this is not representative of the actual population of word searches.

6.1.4 Conclusions

The data (as seen in Figure 6.1) suggests that the Mean Single Threshold, Manually Selected Single Threshold and Histogram Single Threshold (after erroneously small rows and columns have been

removed) segmentation algorithms are the best models of a word search. This was to be expected, since the more advanced segmentation algorithms are designed to less susceptible to erroneous data and noise, leading to them being more lenient models of what a word search looks like.

However, the results are all very close and the sample size is too small (at just 77 images) to draw any solid conclusions from these results.

6.2 Word search Segmentation Evaluation

Several word search segmentation algorithms have been implemented over the course of this project. The main aim of this evaluation is to compare the performance of the different algorithms, allowing for the best one to be fed-forward into later evaluations. In addition to performance of the segmentation algorithms, how their performance is affected by the Remove Erroneously small Rows and Columns functionality (described in Section 5.4.9) will also be evaluated.

6.2.1 Evaluation Data and Method

The evaluation will work by segmenting word search images with each segmentation algorithm and seeing if it calculates the correct number of rows and columns.

The evaluation data set that will be used to evaluate this system will not be the standard evaluation data set. This is because this module has not been trained, or previously had its performance measured on any of the data (training, cross-validation or evaluation) meaning that all of these data sets can be used for evaluating it. Using this extra data increases the size of the evaluation data set without altering its quality, leading to a higher confidence in the results.

The evaluation of this module requires images of the word search having been extracted and with perspective corrected (such that the word search is always rectangular). The evaluation data will therefore be images of word searches (extracted from the raw images based on the coordinates of the word search corners) and the corresponding number of rows and columns for each word search image. This gives a data set of 81 word search images, all marked up with the number of rows and columns that make up the word search within.

6.2.2 Results

Full results are provided in Appendix A. (see table A-1). Figure 6.2 graphs the percentage of word search images that get segmented into the correct number of rows and columns by the word search segmentation module for each of the implemented segmentation algorithms.

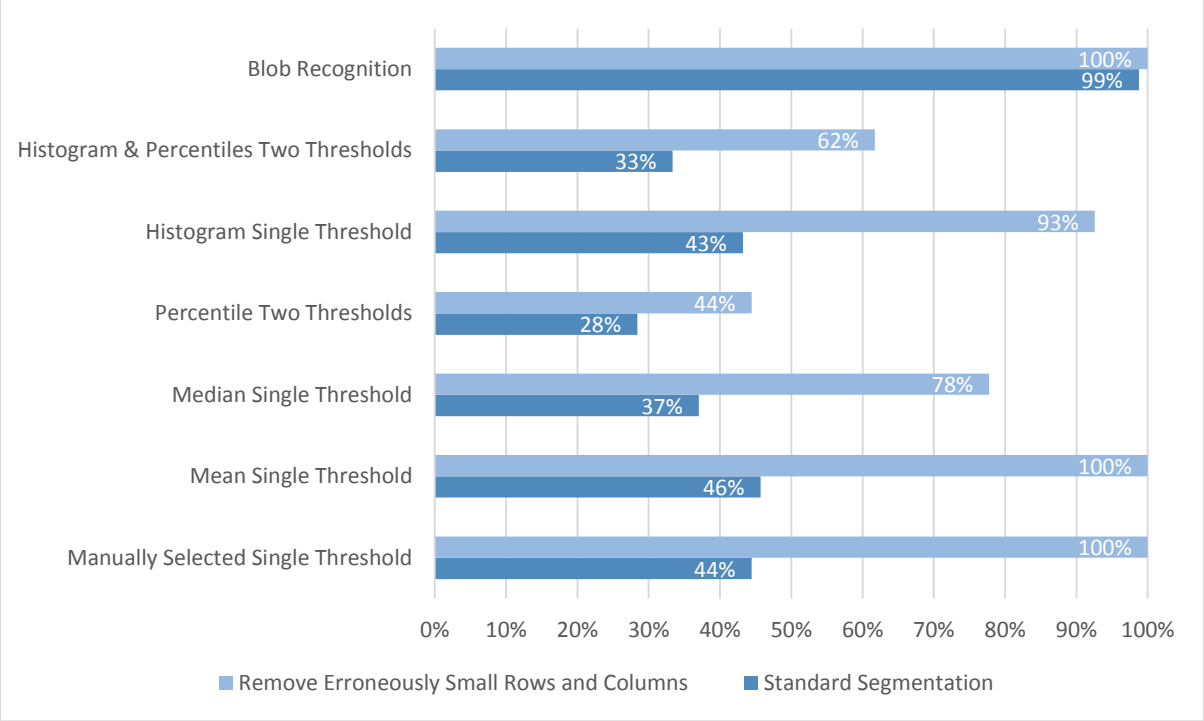


Figure 6.2 Word search Segmentation Performance - Percentage of word search images that get segmented into the correct number of rows and columns. Raw data in Appendix A. (see table A-2).

6.2.3 Discussion

The method used for evaluating whether a segmentation returned by a segmentation algorithm is overly-simple. It checks whether the total number of rows in a returned segmentation is the same as the actual number of rows from the marked up evaluation data, and does the same for columns. It makes no attempt to check that each row and column is in the correct position. An example of how this could affect the results is if there was an additional row in the segmentation at the top of the image, and a row at the bottom was missed. This gives the correct number of rows overall, even though the word search image hasn't been segmented correctly.

This simple check for the number of rows and columns may also be causing the results where the Segmentation has had erroneously small rows and columns removed to be artificially high. This is because if a row is over-segmented into two rows then either one or both of these smaller rows would be removed leading to the segmentation for this row being in the right position. However, the segmentation indices can be anywhere within the whitespace between rows, so it would have to have moved considerably in order to cause problems for later stages in the system.

Also, the evaluation classes each segmentation as either correct or incorrect. This leads to solutions that are almost correct (e.g. have just one over-segmented column) being scored exactly the same as ones that are completely incorrect. This means that scores increase considerably for algorithms that are prone to over-segmentation when they have erroneously small rows and columns removed from their Segmentations.

6.2.4 Conclusions

The results suggest that the Blob Recognition, Mean Single Threshold and Manually Selected Single Threshold segmentation algorithms give the best performance and do so when used in conjunction with Remove Erroneously Small Rows and Columns. However, due to some of the limitations of this

evaluation (as highlighted in Section 6.2.3) that may lead to artificially high performance for algorithms using the Remove Erroneously Small Rows and Columns method, it would appear that Blob Recognition is the best segmentation algorithm.

That is not to say that the other two algorithms capable of scoring 100% aren't as good, but that further evaluation work is required to ensure that the huge jump in performance given by using the Remove Erroneously Small Rows and Columns method is genuine.

Despite the higher uncertainty in the results that have had their erroneously small rows and columns removed, there is a clear increase in performance seen in the results that suggests that the Remove Erroneously Small Rows and Columns method greatly reduces the problem of over-segmentation. This is further backed up by the performance gains on algorithms using one threshold being higher than those using two (because two thresholds should reduce the problem of over-segmentation, but should make the problem of under-segmentation more prominent).

6.3 Feature Extraction and Classification Evaluation

Evaluating the performance of the different feature extraction methods used is done in order to select one for use in the final system. It is also done in order to see whether this area of the system requires further work and in order to speculate as to whether or not it could benefit from more advanced techniques. The performance of each feature extractor will be evaluated by comparing the performance of neural networks trained to work with each feature extractor; all networks will have undergone exactly the same training routine.

6.3.1 Evaluation Data and Method

The evaluation will be performed by taking a data set of character images (from word searches) that have been marked up with their corresponding character and then performing feature extraction and classification on each character and checking if the most likely character from the classifier output scores is the same as what the character is marked up as being. In order to remove any whitespace from the character image before passing it to the feature extractor, the Character Image Extraction module (as implemented in Section 5.5) will be used on all images before they have their features extracted.

The evaluation data for this module will be the standard evaluation data set. This is because both the training and cross-validation data sets have already been used whilst training the classifier, and the training data may have been used to train the feature extractor (if the feature extractor must first learn the data).

The images of individual characters required for this evaluation will be derived from the standard data set by extracting an image of each word search in the data set and then segmenting each word search in to the known (correct) number of rows and columns. Then each grid entry in the word search will be a single character and the image of it can be extracted. In order to remove any whitespace surrounding characters, each character image will be passed through the Character Image Extractor (as implemented in Section 5.5). This gives an evaluation data set of 11,452 character images taken from word searches and all marked up with their corresponding class labels (the characters A-Z).

6.3.2 Results

The output of the evaluation system for each feature extraction technique is a confusion matrix which shows how the classifier classified the samples for each class. These confusion matrices are too large to include here, and have been placed in Appendix A:

- PCA (All Features) – Appendix A. Table A-4.
- PCA (Top 20 Features) – Appendix A. Table A-5.
- DCT – Appendix A. Table A-6.
- Pixel Values – Appendix A. Table A-7.

In order to compare the performance of the classifier when trained with each feature extraction technique, the number of correct classifications (where the predicted class is the same as the actual class) was summed and placed in Appendix A. (see table A-3). These results were then graphed in Figure 6.3.

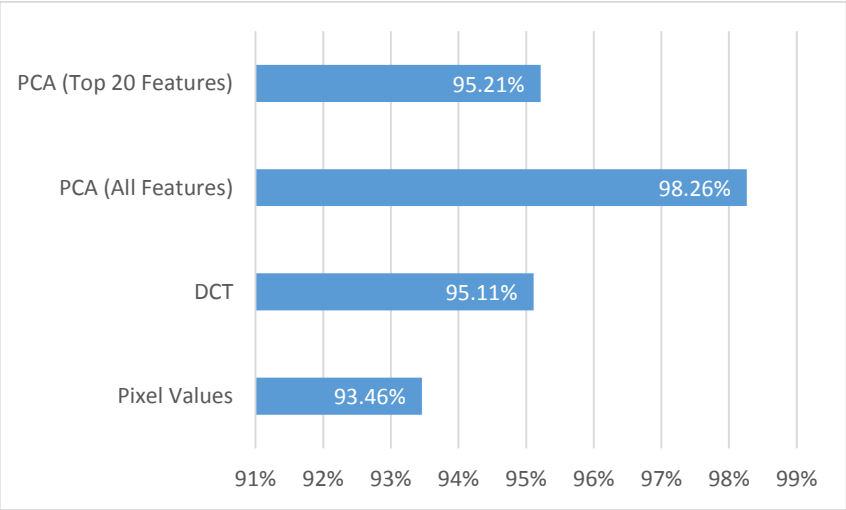


Figure 6.3 Classifier (single-layer neural network) performance when trained with different feature extraction techniques – Percentage of characters classified correctly. Raw data in Appendix A. (see table A-3).

6.3.3 Discussion

A limitation of this evaluation is that the character images had their whitespace removed by the Character Image Extractor rather than it being done manually. It cannot be assumed that the Character Image Extractor will produce the correct result 100% of the time (a possible improvement is actually detailed alongside the current implementation in Section 5.5), and so some of the misclassifications may be caused by errors in the Character Image Extractor.

An additional limitation of this evaluation is that the word search characters that the system was evaluated on were all from word searches in the same collection of books that the system was trained on. Whilst word searches are generally printed in all block capitals in a sans-serif font it is possible that the performance could be lower on word searches that use fonts visually distinct to the range of fonts the system was trained to recognise.

6.3.4 Conclusions

The results suggest that the best feature extraction technique for this classification task (when used in conjunction with the neural network as implemented in Section 5.7) is PCA, and all of the features it produces should be used. It was expected that PCA would be the best feature extractor, as it is the

only method used that learns the trends of the training data in order to determine how to generate features. It was also expected that PCA would perform best when all of the features it produced were used by the classifier, this is because the classifier is a neural network which doesn't treat all inputs as being equally important. Instead, the neural network will learn weights to apply to each input and so the network is capable of learning for itself which of the features it receives should be weighted very highly and which should have little impact on the classification result.

The confusion matrix for using the pixel values in Appendix A. (see table A-7) shows that it didn't manage to classify a single example of the character 'S' correctly. The same problem can be seen in the confusion matrix for DCT, also in Appendix A. (see table A-6) where it only manages to classify two examples of the character 'G' correctly. There is no similar scenario of complete lack of performance for a single character for either classifiers using PCA. This result suggests the features produced by PCA capture the data significantly better than either DCT or the image pixel values manage to. This again suggests PCA is the better feature extraction technique to be fed-forward for use in the final system.

6.4 Rotation Correction Evaluation

The rotation correction module is being evaluated in order to measure how much the performance of the overall system is being affected by this module, and to identify if any further work is required.

The rotation correction module relies on the segmentation, character image extraction, feature extraction and classification modules. The dependency on the segmentation module will be removed for this evaluation by constructing a correct Segmentation object from the known number of rows and columns for each word search in the evaluation data set. However, the character image extraction, feature extraction and classification stages will all be used in this evaluation.

Rather than evaluating this module for each of the implemented feature extractors, the best feature extraction technique and classifier (as determined in Section 6.3) will be fed-forward into this stage. The reason for this is that the classifier that performs best overall should also yield the best performance for this stage, so there is no reason to evaluate the module's performance with the other classifier and feature extractor combinations.

6.4.1 Evaluation Data and Method

The evaluation will be performed by passing a word search image to the rotation correction system. In each case, the output bitmap will be compared to a copy of the input bitmap that has been rotated in order to give the correct solution. The system will have produced the correct solution if the bitmaps are identical, otherwise the solution will be deemed incorrect.

The evaluation data for this module will be the standard evaluation data set. This is because both the training and cross-validation data sets have already been used whilst training the classifier, and the training data was used to train the PCA feature extractor.

The rotation correction module takes an image of a word search as its input, so only the region containing each word search in the evaluation images will be extracted and used for this evaluation. For each extracted word search image there are four possible orientations that it can be in. Each of the word search images will be rotated by each of 0°, 90°, 180° and 270° clockwise in order to increase the size of the evaluation data set fourfold. This yields an evaluation data set of 200 word search images (50 word search images, all rotated through their four possible orientations). This also means that there will be an even distribution of input images in each of the four possible rotations.

6.4.2 Results, Discussion and Conclusions

Of the 200 input images, all 200 were returned in the correct orientation. This result suggests that this is a very strong module in the system requiring no further work. It also suggests that this module is not negatively impacting the score of the overall system.

A limitation of this evaluation is that no attempt has been made to evaluate its performance with incorrect segmentations. This has been done due to the strong performance of the segmentation module (evaluated in Section 6.2) meaning that this module can expect to be working with a correct segmentation. Should further evaluation of the segmentation module reveal that it doesn't perform as well as the current evaluation suggests, then this module could be evaluated with varying degrees of under-segmentation and over-segmentation.

6.5 Full System Evaluation

The system will now be evaluated as a whole; the main aim of doing this is to quantify the performance of the current system and therefore see how close it is to achieving the goal of building a computer vision system that can solve word searches. In addition, the evaluation of the full system will be used to evaluate the effects of changing the algorithms or parameters of a single stage of the system on the overall performance.

6.5.1 Algorithm Combinations to Evaluate

There are many possible combinations of algorithms and their parameters that can be used for the evaluation, each algorithm and parameter that can be changed are:

- Segmentation Algorithm used for Word search Detection
- Does the Segmentation used in Word search Detection have erroneously small rows and columns removed?
- Segmentation Algorithm
- Does the Segmentation have erroneously small rows and columns removed?
- Segmentation Method. There are the following three options that determine how the generated segmentation gets used to split the word search up into images of individual characters:
    - Fixed Width – New segmentation is generated from the number of rows and columns forcing all rows to be of equal height and columns to be of equal width.
    - Varied Width, No Resize – Allow rows and columns to have varied dimensions and do not resize character images to constant dimensions before passing them to the character image extractor
    - Varied Width, With Resize – Allow rows and columns to have varied dimensions, but resize the character images to be of constant size (as would be found if all rows and columns were of equal size)
- Feature Extractor
- Classifier
- Word search Solver

In order to reduce the runtime for this evaluation, only specific combinations selected because they are of interest will be evaluated. For each module of the system that has already been evaluated, the algorithm that performed best for that module independent of the rest of the system will be fed-forward into this evaluation. That will be the following algorithms:

- Mean Single Threshold segmentation algorithm for word search detection
- Blob Recognition segmentation algorithm for word search segmentation
- PCA (All features) for feature extraction

The other algorithms and parameters may be changed for each evaluation in order to see the effect they have on the overall system; the defaults will be:

- Erroneously small rows and columns will not be removed from the word search detection segmentation
- Erroneously small rows and columns will not be removed from the segmentation
- Fixed Width Segmentation method
- Non-Probabilistic word search solver

These defaults have been chosen in order to make the system as simple as possible, allowing for a baseline performance to be measured. Then, each of the defaults can be changed for more complex methods in order to see the effect they have on overall system performance.

6.5.2 Evaluation Data

The evaluation data used here will be the standard evaluation data set. This is because multiple modules that are involved in this evaluation have use the training and cross-validation data to learn the data and so they cannot be used during this evaluation. This gives 49 images to be used for evaluating the full system.

6.5.3 Evaluation Method

It was originally planned (in Section 3.3) to measure the performance of the overall system based on the number of words it finds in the correct position. However, I decided to make the test harder by changing it to be based on the number of word searches for which every word to be found was placed in the correct position. This means that if a word search solution has one word in the wrong position, it will count as a failure. This was done because I think the aim of the system should be to produce a completely correct solution rather than just trying to get most of the words in the correct positions.

6.5.4 Results

Full results of the evaluation are provided in Appendix A. (see table A-8), and are graphed in Figure 6.4.

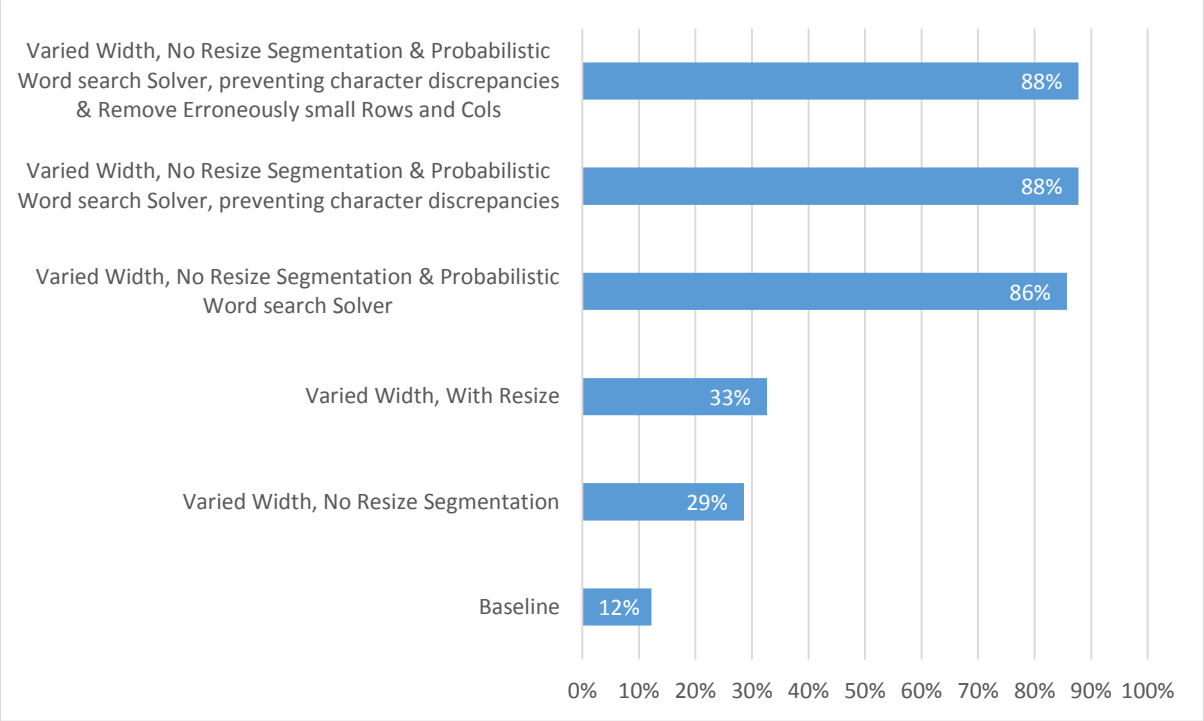


Figure 6.4 Percentage of word searches solved correctly by different combinations of algorithms. Baseline algorithms detailed in Section 6.5.1 and raw data in Appendix A. (see table A-8).

6.5.5 Discussion

Whilst having the full system evaluated based on the number of complete word searches it solves correctly produces a good indicator of the performance of the overall system, it doesn't reflect the performance of the word search solvers particularly well. For a better measure of their respective performances the full system could be evaluated again and scored using the F-Measure for words in the word searches.

The greatest limitation of this evaluation is the small sample size, which creates significant uncertainty in the results being representative of how the system would actually perform on the entire population of word searches.

6.5.6 Conclusions

The first conclusion that can be drawn from these results is that the best combinations of algorithms are capable of finding and correctly solving a word search in 88% of the images in the evaluation data set. The system has a similar overall performance to that of the word search detection module (evaluated in Section 6.1). This combined with the knowledge that other areas of the system that have been evaluated separately indicates that most of the failures are being caused by the word search detection module. This hypothesis will be tested in Section 6.6 by testing the system from the point of having found the word search correctly onwards.

The data also suggests that allowing for rows and columns to be of varied widths and heights leads to a significant increase in performance over forcing all rows and columns to be of equal size. However, due to uncertainty caused by the relatively low sample size it cannot be said whether resizing the image of each character to some constant dimensions before performing character image extraction has any impact on the overall system performance.

The use of the probabilistic word search solver over the non-probabilistic solver offers a large increase in performance (increasing from 29% to 86%). This result was to be expected and the reason for this large jump in performance is explained in Section 6.5.5 but this does not affect the conclusion that the probabilistic solver offers greater performance than the non-probabilistic solver.

The probabilistic solver that also prevents character discrepancies outperforms the standard probabilistic solver. Whilst the performance increase is only from 86% to 88%, this is still a clear indication of this algorithm having higher performance. The reasoning for this is because the solver that prevents character discrepancies will always return the same solution as the standard probabilistic solver if the first candidate solution is valid (as all correct ones must be), and only change the solution in the event of it detecting a character discrepancy. Therefore any performance gain from using this algorithm is a clear indicator that it does perform better on the entire population of word searches.

Removing erroneously small rows and columns from the Segmentation object generated by the blob recognition segmentation algorithm showed no increase in overall system performance. This suggests that it may offer no performance advantage over segmentation by blob recognition alone. This cannot be said with any certainty though since the sample size is too small to measure the very small performance gain it could potentially offer (based on the segmentation evaluation results in Figure 6.2).

6.6 Post-Detection Evaluation

As previously noted in Section 6.5.6 , the highest performance of the overall system is very similar to that of the word search detection module, whilst all other modules of the system that have evaluated independently perform significantly better. This leads to the hypothesis that the majority of the failures in the overall system are being caused by the word search detection module. Here this hypothesis will be tested by evaluating the system from word search segmentation onwards (effectively skipping the detection stage by using the marked up data, artificially giving it 100% performance).

The algorithm combinations will work in the same way as was used for the full system and are explained in Section 6.5.1 with the only difference being that the first algorithm and parameter both relate to word search detection and are therefore ignored for this evaluation. System performance will also be measure by the same method, as described in Section 6.5.3 . Due to the same methodology being employed, the same limitations that are found in the full system evaluation (detailed in Section 6.5.5) are also present here.

6.6.1 Evaluation Data

The evaluation data used here will be the evaluation data set. The input for this evaluation will be images of word searches, so for each image in the standard evaluation data set, an image will be extracted for each word search it contains. This yields 50 word search images to be used in the evaluation of these stages.

6.6.2 Results

Full results are provided in Appendix A. (see table A-9), and are graphed in Figure 6.5.

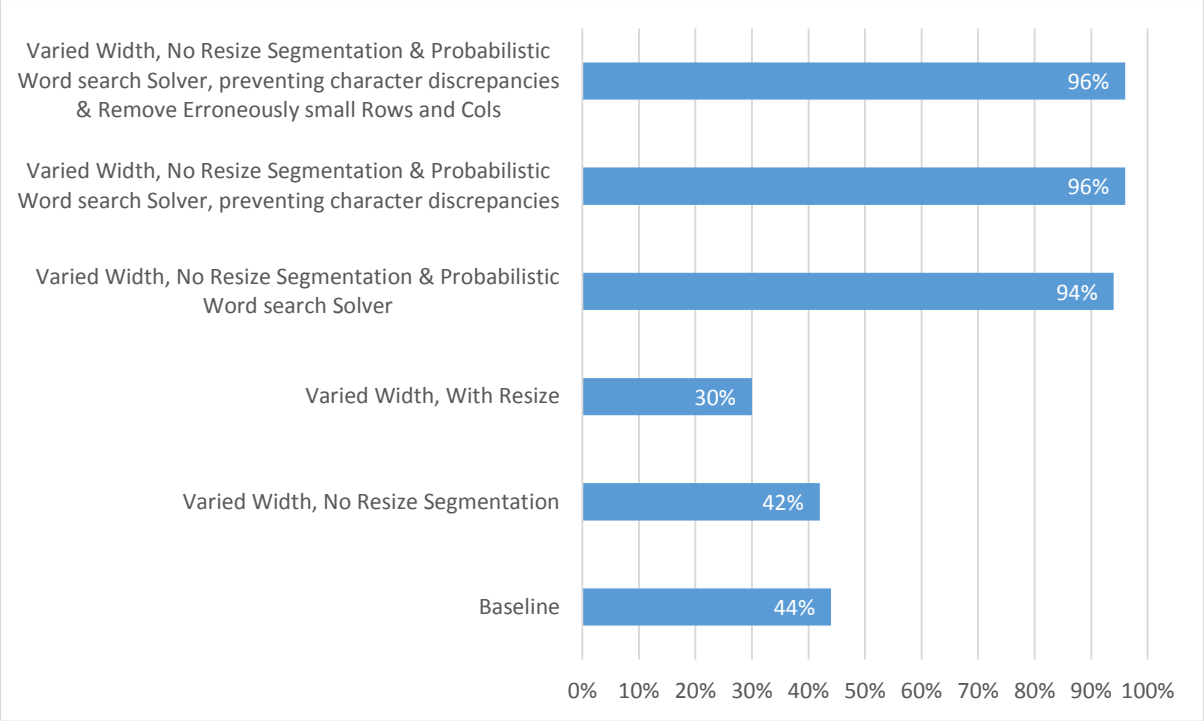


Figure 6.5 Percentage of word searches solved correctly from the Segmentation stage onwards by different combinations of algorithms. Baseline algorithms details in Section 6.5.1 and raw data in Appendix A. (see table A-9).

6.6.3 Conclusions

The data suggests that the majority of the failures in the overall system are caused by the word search detection module, with the highest performance increasing from 88% to 96% of word searches being solved correctly when the word search detection module is removed from the evaluation. The main conclusion here is therefore that the original hypothesis is correct, and the word search detection module is the weakest part of the system, requiring further work.

The data from this evaluation also appears to suggest that not resizing character images before they are input into the character image extractor increase performance. In order to come to this conclusion each of the results for rows and columns with varied widths and heights can be compared to each other. However, due to the small sample size and the conflicting result from the Full System evaluation, more data and further evaluation would be required to determine this with any certainty.

It is worth noting that the baseline algorithm combination performing better than either of the algorithm combinations using varied width segmentation is to be expected. This is because the evaluation data being used is the word search images as they are marked up in the database and when these images are marked up they are all done so that their rows and columns are distributed evenly over the entire image. This means that the results for the baseline algorithm combination are artificially high.

Chapter 7 Conclusions

This aim of this project was to create a Computer Vision system capable of detecting word searches and then using Artificial Intelligence techniques to solve them. This has been achieved, with 88% of images containing a word search having it detected and solved correctly.

The problem of segmenting a word search into rows and columns proved to be much harder than initially anticipated, with the initial segmentation algorithms performing quite poorly. After much further work on this area, the problem has been solved, with the best algorithm (whose main difference is its use of blob recognition) capable of getting the segmentation correct 100% of the time when evaluated.

Word search detection was an area initially identified as requiring a lot of work, and unfortunately time constraints combined with other areas of the system being more difficult than initially anticipated lead to this part of the system not getting much time spent on it. The solution used imposes an artificial constraint on the word searches being found in that they must be bounded by a rectangle. Whilst this is the case for many word searches, it is not the case for them all and the system will fail to detect any word searches without the bounding box. Even with this artificial constraint, the word search detection module accounts for the majority of instances where the system cannot return the correct solution for an image. The percentage of images the system can produce the completely correct solution for jumps from 88% to 96% when the word search detection module is bypassed.

The feature extraction and classification part of the system performed well, managing to correctly classify 98.26% of 11,452 images of characters correctly. This high performance was largely thanks to the neural network and its training method, although a more complex network and considerably more training data would have been required if it weren't for PCA feature extraction. In addition, the rotation correct module which utilises the feature extraction and classification system exceeded expectations, correcting the orientation of word search images 100% of the time in evaluation.

The word search solver module has also been a success with the probabilistic solver making up for many errors made in earlier stages. The probabilistic solver that prevents character discrepancies takes this even further, able to work with poorer quality character probabilities and still find the correct solution. Its heuristic state space search approach to the intractable problem of selecting the optimal position for each word without having a single grid space being used as more than one different character goes well beyond what was initially expected for this stage and is a highlight of the project.

The main areas for further work are word search detection and the related problem of finding the list of words that are to be found in the word search elsewhere on the page, which hasn't been attempted during this project. The main aim of further work on the word search detection module would be to detect word searches that are not bound by a rectangle.

In conclusion, this project set quite an ambitious goal for the given timeframe, but it has succeeded in producing robust, high performance solutions in almost all areas of problems it set out to solve. For this reason the project can be considered a success, with some areas for requiring further work.

Finally, now this project has been completed it is my intention to open-source all of the software that has been created for this project. This is to allow it to be used as an example of the capabilities of the AForge.NET framework, and in the hope that someone somewhere will learn from it.

References

[1] C.-N. Anagnostopoulos, I. Anagnostopoulos, I. Psoroulas, V. Loumos and E. Kayafas, "License Plate Recognition From Still Images and Video Sequences: A Survey," *IEEE Transactions on Intelligent Transportation Systems,* vol. 9, no. 3, pp. 377-391, September 2008.

[2] "Tesseract-OCR," Google, [Online]. Available: https://code.google.com/p/tesseract-ocr/. [Accessed 12 June 2014].

[3] R. Smith, "An Overview of the Tesseract OCR Engine," in *Proceedings of the Ninth International Conference on Document Analysis and Recognition*, Washington DC, 2007.

[4] "Bézier curve," [Online]. Available: http://en.wikipedia.org/wiki/B%C3%A9zier_curve. [Accessed 06 December 2013].

[5] A. Kirillov, "From glyph recognition to augmented reality," [Online]. Available: http://www.codeproject.com/Articles/258856/From-glyph-recognition-to-augmented-reality. [Accessed 10 December 2013].

[6] R. L. Hoffman and J. W. McCullough, "Segmentation Methods for Recognition of Machine-Printed Characters," *IBM Journal of Research and Development,* pp. 153-165, March 1971.

[7] R. G. Casey and E. Lecolinet, "A Survey of Methods and Strategies in Character Segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 18, no. 7, pp. 690-706, July 1996.

[8] H. Mahini, S. Kasaei, F. Dorri and F. Dorri, "An Efficient Features–Based License Plate Localization Method," in *The 18th International Conference on Pattern Recognition (ICPR'06)*, Hong Kong, 2006.

[9] N. Baddiri, B. N. K. Christu, B. S. Kumar and S. Zaheeruddin, "IR Based Color Image Preprocessing Using PCA with SVD Equalisation," in *12th International Conference on Intelligent Systems Design and Applications (ISDA)*, Kochi, 2012.

[10] A. Kirillov, "AForge.NET," [Online]. Available: http://www.aforgenet.com/. [Accessed 20 May 2014].

[11] N. Otsu, "A Threshold Selection Method from Gray-Level Histograms," *IEEE Transactions on Systems, Man, and Cybernetics,* vol. 9, no. 1, pp. 62-66, 1979.

[12] D. Bradley and G. Roth, "Adaptive Thresholding Using the Integral Image," *Journal of Graphics Tools,* vol. 12, no. 2, pp. 13-21, 2007.

[13] G. E. Hinton and R. R. Salakhutdinov, "Reducing the Dimensionality of Data with Neural Networks," *Science,* vol. 313, pp. 504-507, 28 July 2006.

[14] Ø. D. Trier, A. K. Jain and T. Taxt, "Feature Extraction Methods for Character Recognition - A Survey," *Pattern Recognition,* vol. 29, no. 4, pp. 641-662, 1996.

[15] W. Pratt, W.-H. Chen and L. R. Welch, "Slant Transform Image Coding," *IEEE Transactions on Communications,* vol. 22, no. 8, pp. 1075-1093, 1974.

[16] "Comparison of optical character recognition software," [Online]. Available: http://en.wikipedia.org/wiki/Comparison_of_optical_character_recognition_software. [Accessed 2 December 2013].

[17] K. K. Kim, K. I. Kim, J. B. Kim and H. J. Kim, "Learning-based approach for license plate recognition," in *Neural Networks for Signal Processing X, 2000. Proceedings of the 2000 IEEE Signal Processing Society Workshop*, Sydney, 2000.

[18] V. Franc, A. Zien and B. Scholkopf, "Support Vector Machines as Probabilistic Models," in *28th International Conference on Machine Learning (ICML-11)*, Bellevue, Washington, 2011.

[19] L. Xu, A. Krzyzak and C. Y. Suen, "Methods of Combining Multiple Classifiers and Their Applications to Handwriting Recognition," *IEEE Transactions On Systems, Man, And Cybernetics,* vol. 22, no. 3, pp. 418-435, May/June 1992.

[20] "OpenCV," [Online]. Available: http://opencv.org/. [Accessed 12 June 2014].

[21] "EmguCV," [Online]. Available: http://www.emgu.com/wiki/index.php/Main_Page. [Accessed 12 June 2014].

[22] "Mono Project," [Online]. Available: http://www.mono-project.com/Main_Page. [Accessed 12 June 2014].

[23] "MATLAB," MathWorks, [Online]. Available: http://www.mathworks.co.uk/products/matlab/. [Accessed 12 June 2014].

[24] "GNU Octave," [Online]. Available: http://www.gnu.org/software/octave/. [Accessed 12 June 2014].

[25] Y.-P. Huang, S.-Y. Lai and W.-P. Chuang, "A Template-Based Model for License Plate Recognition," in *Proceedings of the 2004 IEEE International Conference on Networking, Sensing & Control*, Taipei, 2004.

[26] "BT709 Field," AForge.NET, [Online]. Available: http://www.aforgenet.com/framework/docs/html/e4559be6-5778-dc89-75b1-507cc63c96c7.htm. [Accessed 09 December 2013].

[27] "Positive Predictive Value," [Online]. Available: http://en.wikipedia.org/wiki/Positive_predictive_value. [Accessed 1 December 2013].

[28] S. V. Rice, F. R. Jenkins and T. A. Nartker, "The Fifth Annual Test of OCR Accuracy," Information Science Research Institute, Las Vegas, 1996.

[29] “Iterative and Incremental Development,” [Online]. Available: http://en.wikipedia.org/wiki/Iterative_and_incremental_development. [Accessed 29 April 2014].

[30] GNU, “GNU Lesser General Public License,” GNU, 29 June 2007. [Online]. Available: http://www.gnu.org/licenses/lgpl.html. [Accessed 20 May 2014].

[31] A. Kirillov, “AForge.NET Framework's License,” [Online]. Available: http://www.aforgenet.com/framework/license.html. [Accessed 20 May 2014].

[32] A. Kirillov, “AForge.NET Online Documentation,” [Online]. Available: http://www.aforgenet.com/framework/docs/. [Accessed 20 May 2014].

[33] A. Kirillov, “AForge.NET Framework Source Code,” [Online]. Available: https://code.google.com/p/aforge/source/browse/. [Accessed 20 May 2014].

[34] C. R. Souza, “Accord.NET Framework,” [Online]. Available: http://accord-framework.net/. [Accessed 20 05 2014].

[35] “Comparison of C# and Visual BASIC .NET,” [Online]. Available: http://en.wikipedia.org/wiki/Comparison_of_C_Sharp_and_Visual_Basic_.NET#Features_of_Visual_Basic_.NET_not_found_in_C.23. [Accessed 20 May 2014].

[36] Microsoft, “MSDN: unsafe (C# Reference),” [Online]. Available: http://msdn.microsoft.com/en-us/library/chfa2zb8.aspx. [Accessed 20 May 2014].

[37] Atlassian, “Git Workflows: Gitflow Workflow,” Atlassian, [Online]. Available: https://www.atlassian.com/git/workflows#!workflow-gitflow. [Accessed 20 May 2014].

[38] “GNU General Public License, version 2,” GNU, June 1991. [Online]. Available: http://www.gnu.org/licenses/gpl-2.0.html. [Accessed 20 May 2014].

[39] Microsoft, “Microsoft Dreamspeak - Students,” Microsoft, [Online]. Available: https://www.dreamspark.com/Student/Default.aspx. [Accessed 20 May 2014].

[40] “Shoelace formula,” [Online]. Available: http://en.wikipedia.org/wiki/Shoelace_formula. [Accessed 05 June 2014].

[41] IEEE, “IEEE Standard for Floating-Point Arithmetic,” *IEEE Std,* 2008.

[42] “Word search,” [Online]. Available: http://en.wikipedia.org/wiki/Word_search. [Accessed 2 December 2013].

[43] “Waterfall Model,” [Online]. Available: http://en.wikipedia.org/wiki/Waterfall_model. [Accessed 29 April 2014].

[44] J. Highsmith and A. Cockburn, “Agile Software Development: The Business of Innovation,” *Computer,* vol. 34, no. 9, pp. 120-127, 2001.

[45] “Manifesto for Agile Software Development,” 2001. [Online]. Available: http://agilemanifesto.org/. [Accessed 29 April 2014].

Appendix A

Segmentation Method	Standard Segmentation	Remove Erroneously Small Rows and Columns
Manually Selected Single Threshold	88.31%	88.31%
Mean Single Threshold	88.31%	88.31%
Median Single Threshold	84.42%	87.01%
Percentile Two Thresholds	79.22%	79.22%
Histogram Single Threshold	87.01%	88.31%
Histogram & Percentiles Two Thresholds	84.42%	83.12%
Blob Recognition	85.71%	84.42%

A-1 Data from Figure 6.1

Segmentation Method	Standard Segmentation	Remove Erroneously Small Rows and Columns
Manually Selected Single Threshold	44.44%	100.00%
Mean Single Threshold	45.68%	100.00%
Median Single Threshold	37.04%	77.78%
Percentile Two Thresholds	28.40%	44.44%
Histogram Single Threshold	43.21%	92.59%
Histogram & Percentiles Two Thresholds	33.33%	61.73%
Blob Recognition	98.77%	100.00%

A-2 Data from Figure 6.2

Feature Extractor	Correct Classifications	Percentage of Characters Classified Correctly
Pixel Values	10,703/11,452	93.46%
DCT	10,892/11,452	95.11%
PCA (All Features)	11,253/11,452	98.26%
PCA (Top 20 Features)	10,904/11,452	95.21%

A-3 Data from Figure 6.3

		Predicted Class																								
	label	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Actual Class	A	760	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
	B	0	392	0	0	0	0	0	1	5	0	0	0	1	0	1	0	0	0	2	0	0	0	6	0	0	0
	C	0	0	490	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	D	0	1	0	417	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0
	E	0	0	0	0	881	0	3	0	2	0	0	0	0	0	0	0	2	0	0	0	0	2	14	0	0	0
	F	0	0	0	0	1	382	0	1	1	0	0	0	0	0	2	0	0	0	0	0	0	0	1	0	0	0
	G	0	0	1	0	0	1	349	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0
	H	0	0	0	0	0	0	0	386	1	1	0	0	2	0	1	0	0	0	0	0	1	0	0	1	0	0
	I	0	0	2	0	0	6	0	0	457	0	0	1	0	10	1	1	0	1	0	10	0	0	1	3	0	0
	J	0	0	0	0	0	0	0	0	1	173	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	K	2	0	0	0	0	3	0	1	0	0	349	0	0	0	0	0	0	3	0	0	0	0	3	1	0	0
	L	0	0	0	0	0	0	0	0	0	0	0	559	0	0	0	0	1	0	0	0	0	0	0	0	0	0
	M	0	0	0	0	0	0	0	0	4	0	0	0	403	0	0	0	0	0	0	0	0	4	0	0	0	0
	N	1	0	0	0	0	0	0	0	2	0	0	0	1	550	0	0	0	0	0	0	0	0	0	0	0	0
	O	0	0	1	0	0	0	0	0	0	0	0	0	0	0	587	0	7	0	0	0	0	5	2	0	0	0
	P	1	0	0	0	0	2	0	0	0	0	0	0	0	0	3	383	0	0	0	0	0	0	0	0	0	0
	Q	0	0	0	1	0	0	0	0	0	1	0	0	0	0	5	0	184	0	0	0	0	0	0	0	0	0
	R	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	706	0	0	0	0	10	1	0	0
	S	0	1	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	535	0	0	1	0	0	0	0
	T	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	734	0	1	0	0	0	0
	U	0	0	0	0	0	0	0	0	1	3	0	2	0	0	0	0	0	0	0	0	413	4	0	0	0	0
	V	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	217	0	0	0	0
	W	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	269	0	0	0
	X	0	0	0	0	0	0	0	0	0	0	0	3	0	0	1	1	0	0	0	0	0	0	0	190	0	0
	Y	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	6	0	0	340	0
	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	147

A-4 Confusion Matrix for classification with PCA feature extraction (All Features)

Appendix A

	Predicted Class																									
Actual Class	*label*	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
	A	749	0	0	0	0	1	0	0	9	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	1
	B	0	386	0	0	1	0	0	3	3	0	0	0	0	0	0	0	1	3	10	0	0	0	1	0	0	0
	C	0	0	476	0	0	0	0	0	0	0	0	0	0	0	14	0	0	0	0	0	0	0	0	0	0	0
	D	1	9	0	389	0	0	0	0	0	0	0	0	0	0	11	0	1	0	0	0	5	1	3	0	0	2
	E	0	0	0	0	871	5	0	0	6	0	0	0	0	0	0	0	14	0	5	0	0	0	1	0	0	2
	F	1	0	0	0	0	368	0	0	0	0	0	1	0	0	0	9	0	3	0	3	0	0	2	0	0	1
	G	4	0	2	0	0	0	273	0	3	0	0	1	0	0	49	0	2	0	20	1	0	0	0	0	0	0
	H	1	0	0	0	0	0	0	381	0	0	3	1	1	2	0	1	0	0	0	0	0	0	3	0	0	0
	I	3	0	2	1	0	1	0	0	453	0	7	0	1	1	0	4	0	0	0	12	0	1	4	0	0	4
	J	0	0	0	0	0	0	0	0	4	150	0	0	0	0	0	0	0	0	0	1	19	0	0	0	0	0
	K	0	0	0	0	0	0	0	0	0	0	357	1	0	0	0	0	2	0	0	2	0	0	0	0	0	0
	L	0	0	0	0	0	0	0	0	0	0	0	560	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	M	0	0	0	0	0	0	0	0	4	0	5	0	394	0	0	0	1	0	2	0	0	5	0	0	0	0
	N	0	0	0	0	0	0	0	0	0	0	0	0	0	553	0	0	0	0	0	1	0	0	0	0	0	0
	O	5	0	0	0	0	0	0	0	0	0	0	0	0	0	581	0	0	0	0	0	13	1	1	0	0	1
	P	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	386	0	0	0	0	0	0	1	0	0	0
	Q	14	0	0	0	0	0	0	0	15	0	0	45	3	3	56	0	14	11	0	2	18	0	6	0	0	4
	R	7	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	701	0	0	0	1	7	1	0	0
	S	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	537	0	0	0	0	0	0	0
	T	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	734	0	0	0	0	0	0
	U	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	422	0	0	0	0	0
	V	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	217	0	0	0	0
	W	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	269	0	0	0
	X	0	0	0	0	0	0	0	0	0	0	2	2	0	0	0	1	0	0	0	2	0	0	0	188	0	0
	Y	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	348	0
	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	147

A-5 Confusion Matrix for classification with PCA feature extraction (Top 20 Features)

		Predicted Class																									
	label	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
Actual Class	***A***	758	0	0	0	0	0	0	2	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	B	0	399	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	3	3	0	0	0	1	0	0	0
	C	0	2	481	0	0	0	0	0	0	0	0	0	0	0	6	0	0	0	1	0	0	0	0	0	0	0
	D	0	1	0	412	0	0	0	0	0	0	0	0	0	0	6	0	0	0	0	0	3	0	0	0	0	0
	E	0	0	0	0	868	15	0	1	6	0	0	4	0	0	0	0	0	1	3	1	0	0	5	0	0	0
	F	0	0	0	0	0	386	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
	G	18	6	2	0	4	28	2	0	4	0	13	0	0	10	130	0	19	0	114	0	1	1	3	0	0	0
	H	0	0	0	0	0	0	1	382	0	0	0	0	2	3	1	1	0	0	0	0	2	0	0	0	1	0
	I	1	0	2	1	0	6	0	0	465	0	0	5	0	6	0	3	0	0	0	1	0	0	2	0	0	1
	J	0	0	0	0	0	0	0	0	1	173	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	K	0	0	1	0	0	3	0	3	0	0	350	0	0	0	0	0	0	4	0	0	0	0	0	1	0	0
	L	0	0	0	0	0	0	0	0	0	0	0	560	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	M	0	0	0	0	0	0	0	0	2	0	0	0	405	0	0	0	0	0	0	0	2	0	0	0	2	0
	N	0	0	0	0	0	0	0	1	0	0	0	0	1	551	0	0	0	0	0	0	0	0	0	0	1	0
	O	0	0	0	1	0	1	0	0	0	0	0	0	0	0	598	0	0	0	0	0	0	1	1	0	0	0
	P	0	0	0	0	0	9	0	0	0	0	0	0	0	0	1	373	0	0	0	0	0	0	0	0	0	0
	Q	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11	0	180	0	0	0	0	0	0	0	0	0
	R	4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	714	0	0	0	0	0	0	0	0
	S	0	1	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	1	534	0	0	0	0	0	0	0
	T	0	0	0	0	0	0	0	0	15	0	0	0	0	0	0	0	0	0	0	719	0	0	0	0	1	0
	U	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	417	3	0	0	0	0
	V	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	217	0	0	0	0
	W	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	269	0	0	0
	X	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	1	0	0	0	190	1	0
	Y	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	3	0	2	0	0	343	0
	Z	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	146

A-6 Confusion Matrix for classification with DCT feature extraction

		Predicted Class																								
Actual Class	label	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
	A	757	0	0	0	0	1	0	0	2	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	B	0	347	0	5	31	4	2	7	7	0	0	0	0	0	3	0	0	2	0	0	0	0	0	0	0	0
	C	0	0	490	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	D	0	0	0	415	0	0	1	0	0	0	0	0	0	0	6	0	0	0	0	0	0	0	0	0	0	0
	E	0	0	0	0	891	3	5	1	3	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	F	0	0	0	0	0	385	0	0	0	0	0	1	0	0	2	0	0	0	0	0	0	0	0	0	0	0
	G	0	0	0	0	0	3	352	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	H	0	0	0	0	0	1	0	385	0	0	0	1	1	2	2	0	0	0	0	0	0	0	0	1	0	0
	I	0	0	2	0	0	3	3	0	465	0	0	5	0	5	1	3	0	1	1	1	0	1	0	0	0	3
	J	0	0	0	0	0	0	0	0	1	171	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
	K	0	0	2	0	0	2	0	0	0	1	349	0	0	0	0	0	0	4	0	0	0	0	0	4	0	0
	L	0	0	0	0	0	0	0	0	0	0	0	560	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	M	0	0	0	0	0	0	0	0	3	0	0	0	405	0	0	0	0	0	0	0	0	2	0	0	1	0
	N	0	0	0	0	0	0	0	0	1	0	1	0	1	550	0	0	0	0	0	0	1	0	0	0	0	0
	O	0	0	0	0	0	0	0	0	0	1	0	0	0	0	601	0	0	0	0	0	0	0	0	0	0	0
	P	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	388	0	0	0	0	0	0	0	0	0	0
	Q	0	0	0	0	1	0	0	0	0	0	0	0	0	0	8	0	182	0	0	0	0	0	0	0	0	0
	R	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	9	0	707	0	0	0	0	0	1	0	0
	S	15	7	20	2	49	31	352	0	10	15	1	0	0	12	4	0	0	8	0	0	0	0	0	0	13	0
	T	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	731	0	0	0	0	0	2	0
	U	0	0	4	0	1	0	0	2	1	5	0	0	0	0	1	0	0	0	0	407	0	2	0	0	0	0
	V	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	217	0	0	0	0	0
	W	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	268	0	0	0
	X	0	0	0	0	0	0	0	0	0	1	0	2	0	0	0	2	0	0	0	0	0	0	0	190	0	0
	Y	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	4	0	1	343	0
	Z	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	147

A-7 Confusion Matrix for classification using the image's pixel values

Algorithm Combination	Percentage of word searches solved correctly
Baseline	12.24%
Varied Width, No Resize Segmentation	28.57%
Varied Width, With Resize	32.65%
Varied Width, No Resize Segmentation & Probabilistic Word search Solver	85.71%
Varied Width, No Resize Segmentation & Probabilistic Word search Solver, preventing character discrepancies	87.76%
Varied Width, No Resize Segmentation & Probabilistic Word search Solver, preventing character discrepancies & Remove Erroneously small Rows and Cols	87.76%

A-8 Data from Figure 6.4

Algorithm Combination	Percentage of word searches solved correctly
Baseline	44%
Varied Width, No Resize Segmentation	42%
Varied Width, With Resize	30%
Varied Width, No Resize Segmentation & Probabilistic Word search Solver	94%
Varied Width, No Resize Segmentation & Probabilistic Word search Solver, preventing character discrepancies	96%
Varied Width, No Resize Segmentation & Probabilistic Word search Solver, preventing character discrepancies & Remove Erroneously small Rows and Cols	96%

A-9 Data from Figure 6.5