

Instance-Based Recognition of Screen-Rendered Text in a System for Pixel-Based Reverse-Engineering of Graphical Interfaces


By Stephen Joe Jonany
Advised By Professor James Fogarty

A senior thesis submitted in partial fulfillment of
the requirements for the degree of

**Bachelor of Science
With Departmental Honors**

**Computer Science & Engineering
University of Washington
June 2013**

Presentation of work given on June 6, 2013

Thesis and presentation approved by 

Date 6/13/2013

Abstract

Pixel-based systems for reverse engineering interfaces have made it possible to modify existing user interfaces without access to their underlying source code. Rather than waiting for a company to incorporate a new feature into an application, pixel-based systems empower *any* developer to make an application more accessible, usable, and useful. In contrast to standard computer vision techniques, these systems work by leveraging the fact that common interface components have a consistent appearance. Widgets are rendered procedurally, and so it is possible to exactly match their pixels for fast and accurate interpretation. Unfortunately, this same strategy fails when it comes to recognizing screen-rendered text, limiting the capabilities of pixel-based systems. Due to sub-pixel rendering and antialiasing, the appearance of a single character can vary in many subtle ways, which makes exact matching strategies ineffective. Moreover, off-the-shelf OCR technologies that are usually tailored for recognizing high-resolution handwritten text are generally ineffective because of the extreme low resolution of typical interface text. This paper discusses a scalable method that leverages the consistency of this type of text, overcomes the problems entailed by low-resolution images, and allows us to recognize screen-rendered text with reasonable speed and high accuracy.

1. Introduction

Current methods for implementing graphical user interfaces create fundamental challenges for HCI research and practice. Most interfaces do not usually expose their underlying source code to the public. Because of this, researchers are often unable to demonstrate or evaluate new techniques beyond small toy applications, and practitioners are often unable to adopt methods from the literature in new and existing applications. For the same reason, the development of new interfaces is also hindered with the duplication of efforts. Fortunately, pixel-based systems for runtime interface modification offer great potential to address these problems. Because all graphical interfaces ultimately consist of pixels, these methods can enable modification of interfaces without their source code and independent of their implementation. For example, prior pixel-based systems demonstrate a variety of promising runtime enhancements to existing interfaces, including accessibility enhancements, interface language translation, testing frameworks, interaction techniques, automation tools, and contextual help systems [5, 6].

Unfortunately, these pixel-based systems are limited by their inability to interpret interface text. Current pixel-based systems are at best capable of identifying the *presence* of text, but not interpreting its *value*. Existing pixel-based modifications either limit their functionality such that they can ignore text, or employ ad-hoc methods for interpreting text, which are difficult to reuse, manage, and scale [2]. Pixel-based systems are built on the insight that the pixels of a widget are procedurally defined, which is critical to their performance because it allows for exact or nearly exact matching of pixel values to detect interface elements. However, the appearance of textual content varies more substantially, and cannot be easily identified through exact matching. Modern toolkits often employ sub-pixel rendering and anti-aliasing techniques in text rendering, as can be seen in Figure 1. While these techniques improve readability, they also modify the pixel-level

appearance of text in unpredictable ways. Moreover, off-the-shelf OCR technologies that are usually tailored for recognizing high-resolution handwritten text are generally ineffective because of the extreme low resolution of typical interface text.



Figure 1. Sample Input
Challenges include low-resolution and anti-aliasing.

We address this problem with novel pixel-based methods for interpreting interface text. Instead of abstractly representing a collection of example characters with a complex model, we store all examples and directly compare their pixel values to a candidate image. This instance-based approach leverages the insight that despite the variation in a character’s raw pixel values, the *arrangement* of its pixels is highly consistent. Unlike previous instance-based attempts [3], our methods automatically expand the corpus of example characters with unsegmented images of words. This strategy enables us to improve the accuracy of the system while maintaining its performance. We provide initial evaluation of our methods using a corpus of over seven thousand images captured from a variety of common applications running in Microsoft Windows 7. Our methods achieved about 85% accuracy, and discounting single-character errors gave 95% accuracy. The results showed their potential for use in many practical applications.

2. Component Overview

Our goal is to efficiently and accurately provide labels for images of word-length text. Figure 1 illustrates an example of a typical image input into our system. Because state-of-the-art pixel-based technologies, such as Prefab [2], already have methods for reliably identifying text, we assume that the input images have already been cropped to only the region containing relevant content. Given this assumption, there are two major tasks left to address. One is to segment an image into characters, and the other is to assign labels to those characters.


Our framework consists of 2 main components, illustrated in Figure 2. First, we have an example set which contains images of characters labeled with their corresponding text values. We call the process of building this set of example glyphs the *extraction step* (Section 4). Second, we have an algorithm that simultaneously segments and classifies input images. We call this process the *recognition step* (Section 3). This algorithm works by searching through potential segmentations and then comparing those segments to glyphs in our example set. In summary, the *extraction step* serves to build the example set, which will guide the *recognition step*, whereas the *recognition step* is the main component that will perform the actual recognition.

Extraction

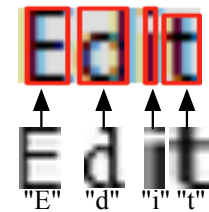
A "A" B "B"
A "A" B "B"
... ..

Populate the
Example Set

Recognition

 Cost = 1000
Cost = 10

Simultaneous
Classification and
Segmentation



Pick the
Best Segmentation

Figure 2. Component Overview

3. Recognition Step

As mentioned, the recognition step leverages the corpus of example images in order to generate a label for an input image. The algorithm works by (1) normalizing the input image to reduce potential variations in its appearance (2) searching through possible segmentations of the image, (3) assigning labels to each segment, and (4) choosing the best combination of segments and corresponding labels according to a cost function. The search algorithm is an efficient dynamic programming solution that reuses segmentation and classification of portions of the image. To obtain a label for a possible segment, our algorithm finds the character in our example set with an appearance that is most similar to the segment. Finally, a cost function based on Euclidean distance is used to rank each segmentation. This section describes each of these steps in more detail.

3.1 Normalization

To reduce potential inconsistencies in the appearance of an image, we normalize input images by converting them to greyscale. We performed background removal by first creating the monochrome version of the same image, and then removing pixels in the greyscale image that correspond to white pixels in its monochrome version, while leaving the remaining pixels unchanged. The monochrome algorithm used is Otsu's method, because our need fits in nicely with the assumption of the algorithm, which is that the pixels in the image are divided into two classes, namely foreground and background. This might leave an expanse of white pixels around the foreground pixels. The final step to the background removal was to crop the image down to the smallest rectangle such that all foreground pixels are captured.



Figure 3. Background Removal

The cropped image is then resized to a constant height to speed up the dynamic programming algorithm. Finally, the color range of the greyscale image is linearly

stretched so it ranges from 0 to 255. This is because there are some images that are darker or lighter than most of the images. In our distance calculation between two different greyscale images, we want to only capture the notion of how a pixel is relatively darker than the other, instead of the absolute difference.

3.2 Dynamic Programming

The core of the recognition step is dynamic programming, where we try to find a way to segment the image into several non-overlapping rectangles such that the maximum over all costs incurred by each of these rectangular regions is minimized. After we find such segmentation, we generate the predicted text content by concatenating the characters represented by each region in order from left to right.

Before showing the algorithm, there are some notations that have to be introduced:

1. Img = The input image that has been normalized
2. $bestMatch(i,j)$ = The lowest cost to match a segment spanning column i to j of Img
3. $MaxNumChar$ = The maximum number of characters Img can represent

The computation of $bestMatch(i,j)$ involves computing the euclidean distance between the greyscaled, 10x10 version of the region bounded by columns i and j of Img , and each of the samples in the example set. The final result is the minimum across all the computed distances, and represents the best cost of identifying the subregion as a single character.

The algorithm can be more formally specified with the notions like so

Input : Sample Image Img

Output: $\{region[1], region[2], ..., region[n]\}$

i.e. segmentation of Img into n regions such that

$Max\{bestMatch(region[1]), ..., bestMatch(region[n])\}$ is minimized.

We need to first determine the number of characters to segment the image into, and for this, we approximate $MaxNumChar$, which is the maximum number of characters. We apply the heuristics that it is unlikely for all the characters in the image to all have heights which are greater than 4 times its width. By guessing that the maximum aspect ratio of any character in the image is 4.0, we can compute the minimum character width by dividing the height of the image with maximum aspect ratio. $MaxNumChar$ is then obtained by dividing the width of Img with the minimum character width. Now that we know that largest number of segmentations, we try to segment the image into varying number of characters up to $MaxNumChar$, and pick one with the minimum total cost.

The structure of the problem lends itself nicely to a dynamic programming solution, where the subproblems entail finding the best way to segment the first few columns of Img into a certain number of characters. We are going to capture the result of this subproblem with the notation $OPT(c,i)$, which represents the optimal cost of segmenting the first i columns of Img into c characters.

Consider how $OPT(c,i)$ can be computed from smaller subproblems. Suppose that we know the best choice for the very last column cut to make is at column j , that is, column j is where the $c-1^{th}$ character should end. We note that the optimal cost for the subproblem, which is the lowest cost for segmenting the first j columns of Img into $c-1$ characters, is $OPT(c-1,j)$. With this information, we can solve the problem optimally for those $c-1$ characters, and combine that subproblem solution with the c^{th} character to get the optimal solution for the entire c characters. The extra cost incurred for appending the c^{th} character, which spans from column $j+1$ to i , is $bestMatch(j+1,i)$. The total penalty for the entire solution is given below.

$$OPT(c-1,j) + bestMatch(j+1,i)$$

However, we made the assumption that we know the value of j . Fortunately, even though we do not know which column j will minimize this value, we know that $j \leq i$. The optimal choice for j will then be among these possibilities, and so we simply go through all the possible values for j , and pick the best one. The recurrence relation is given by the equation below.

$$OPT(c,i) = \min_{1 \leq j \leq i} \{OPT(c-1,j) + bestMatch(j+1,i)\}$$

We consider a concrete example, where we try to compute $OPT(3,100)$ of the provided image, that is, we want to compute the lowest cost of segment the first 100 columns of the image into 3 characters. According the recurrence relation, we have to go through all the possible candidates for j , which is the column where the second character ends, and pick one that minimizes the overall cost. We present 2 of such scenarios to illustrate the intuition behind the algorithm.

In Figure 4, we consider the case where $j = 20$, that is, we consider the case where the first 20 columns will be segmented into 2 characters, and the third character spans from column 21 to 100.

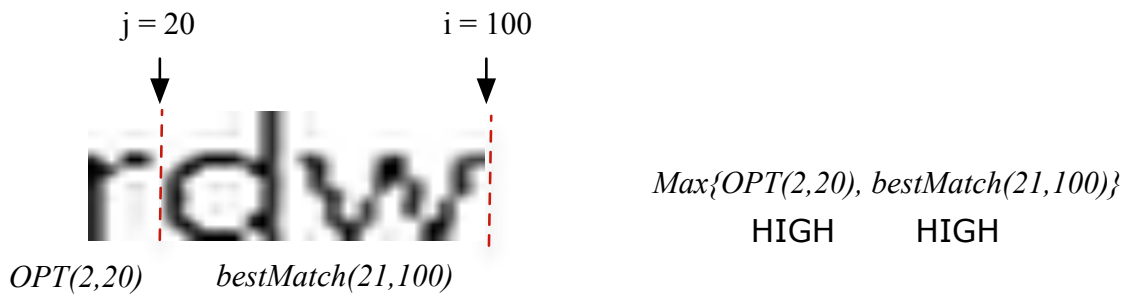


Figure 4. Simulation with $j = 20$

We observe that this choice for j is likely to incur a high cost because $OPT(2,20)$, which stores the lowest cost for segmenting the first 20 columns into 2 characters, is likely to be high. On the left hand side, we can see an example of oversegmentation,

where there is an attempt to segment an image of the letter ‘r’ into two characters, and there is no good way to divide ‘r’ into regions of pixels where each region is similar to samples in the example set. On the other hand, undersegmentation causes $bestMatch(21,100)$ to be high, since there should not be any character sample in the example set that can correspond to the pixels representing the two letters “dw”. We note from this example that for each choice of j , we take into account not only the previous cost of segmenting the first $c-1$ characters, as represented by $OPT(2,20)$, but also the additional cost of appending the leftover c^{th} character, as captured by $bestMatch(21,100)$. By minimizing the maximum over these two values, we can intuitively see that we are also minimizing the maximum cost over each segmentation we make.

We now consider the scenario where we set j to 60. This time, we can see that our choice is likely to be the best candidate to minimize $OPT(3,100)$. As can be seen in Figure 5, $bestMatch(61,100)$ is likely going to have a low cost because there will be a match to a dictionary entry corresponding to the letter ‘w’. $Opt(2,60)$ is also likely to be a low number because the first 40 columns can be segmented nicely into two characters, namely ‘r’ and ‘d’. The maximum across these two low values is thus likely to be low as well.

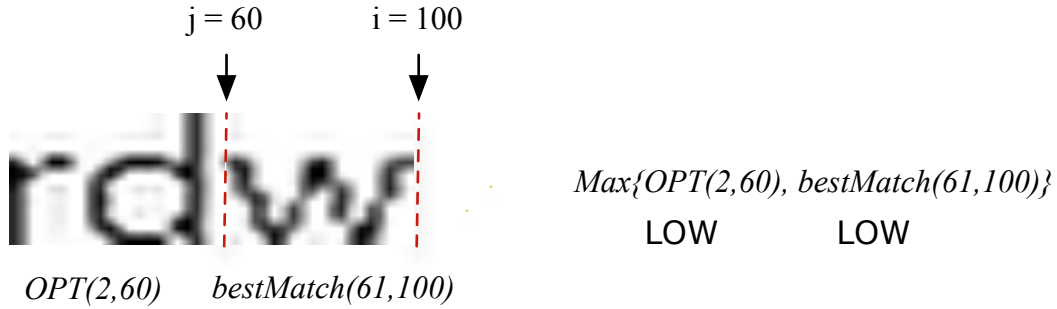


Figure 5. Simulation with $j = 60$

3.3 Optimization

Most of the computing time of this algorithm lies in the calculation of $bestMatch(i,j)$, which is the distance of the closest sample in the example set to the segment $Img[i,j]$. A possible approach is to simply compute the distances between all samples in the example set to the segment of interest. However, we noticed that there are some common properties of screen rendered text that can be leveraged. This observation coupled with the availability of the size and location of the segment in question allowed us to reduce the size of our search space quickly.

Below is a list of the heuristics used:

- (a) A segment whose height is $< \frac{1}{4}$ of $Img.height$ can't possibly be an alphabet or a digit.
- (b) If the segment contains an offset from bottom which is $> \frac{1}{4}$ of bitmap height, it can't possibly correspond to alphabets with dangling bottom tails like 'g','j','p','q','y'.

(c) If the segment contains an offset from top which is $> 1/4$ of bitmap height, it can't correspond to capital letters, numbers, and lowercase letters that have a head that reach the ceiling of images such as 'b', 'd', 'f', 'h', 'k', or 'l'

(d) If segment's aspect ratio is greater than the hardcoded maximum aspect ratio for a character, or is less than the minimum aspect ratio for a character, it can't correspond to the character. These min and max aspect ratio values are manually hardcoded by eyeballing a number of the text images.

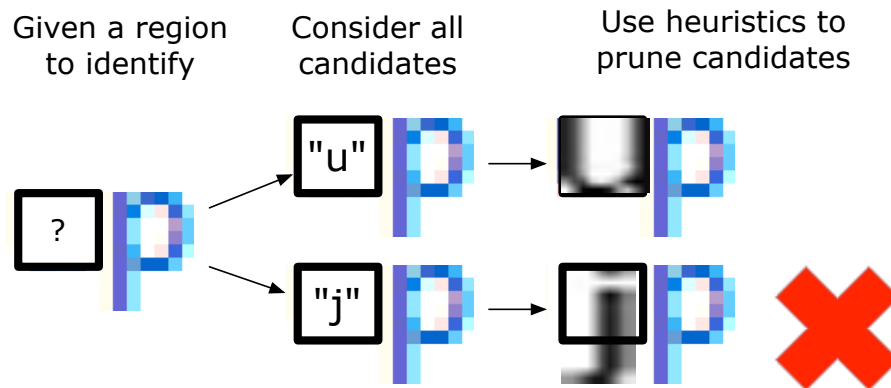


Figure 6. Pruning Heuristic (b) in action.

The segment in question can't possibly correspond to a character with a dangling tail like "j" since it has a noticeable offset from the bottom.

4. Extraction Step

The goal of the extraction step is to accumulate a corpus of image-character label pairs, which will help guide the recognition step. Since the recognition step uses the example set to calculate its segmentation costs, this corpus of examples defines the accuracy of the system.

Accumulating these character images and their labels is a non-trivial task. Automatically generating images of characters using library method calls does not allow the example set to learn new character sets without having detailed information on the types of font that have to be covered. Fortunately, common UI components provide a rich set of text images that we can use. The problem is that the locations of these images need to be isolated from a given screenshot. Once the relevant regions have been identified, they will need to be labeled. Furthermore, since many of these images correspond to words, and we have to recover the mapping between each letter of the word label to the relevant region of pixels to finally produce image-character label pairs ourselves. Finally, we have to deal with noisy samples to maintain the quality of the samples in the example set.

The extraction pipeline can then be classified into two majors steps, as shown in Figure 7. The first is the gathering of weakly-labeled samples, which is a collection of image-word labels. The second step is to produce actual image-character pairs that will be added to the example set.

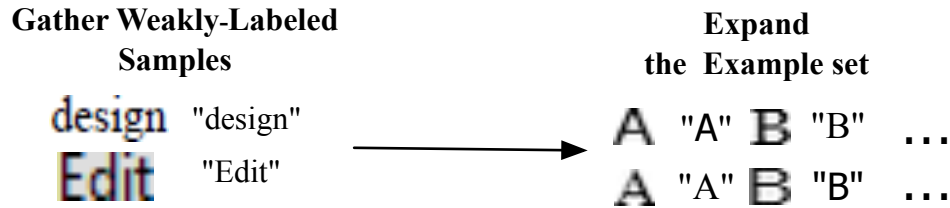


Figure 7. Extraction Step Overview

4.1 Obtaining Weakly-Labeled Samples

4.1.1 Gathering Images of Text

We obtained images of these texts by using Prefab on popular user interfaces and websites such as Windows Explorer and Wikipedia[2]. Each session of gathering these images involves having a person navigate across different user interfaces, and having the screen activity recorded as a video footage. Prefab will then be applied on the video recording to extract cropped images of text. At the end of this step, we have a set of unlabeled images of text. We assume that every image has been tightly cropped to the relevant region containing text by Prefab, and this assumption removes the need to perform further word boundary detection.

4.1.2 Labeling Images

We were able to provide word labels to the obtained images with the help of Amazon Mechanical Turk. Workers were given the task of labeling a set of images with their respective text content. Ideally, we would like to have the labels to indicate the mapping between single characters to regions of pixels in the image. However, we decided that this task is too much to ask for turk workers to do, not only because it is time-consuming, but also because it requires each region to be accurately cropped. We settled with the labels being the word corresponding to the entire image, and in addition, the workers were required to indicate extra attributes of the text. These extra attributes help us filter out images that either do not contain text, or are hardly recognizable.

Besides the problem of having noisy unlabeled images, we also encounter noise in the labels provided by the turk workers. The sources of noise are mostly attributable to human errors, such as typographical errors. Interestingly, some annotations were suspected to be generated by off-the-shelf OCR technologies. We tried to remedy this by providing a tutorial page with sample tasks which require more involved human interaction to be solved. We also had multiple works review the same image, so we could pick the label which the majority of the workers agreed upon.

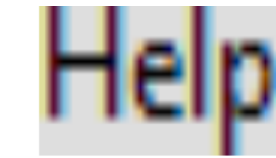


Figure 8. Example of a Single Annotation Task

Help

☐ Italicized ☐ Dark background
☐ Poorly cropped ☐ Icon ☐ Difficult to read

4.2 Expanding the Example Set from Weakly-Labeled Samples

The obtained set of weakly-labeled samples will have to pass a sequence of criteria before being used to expand the example set. These samples are not directly usable because they are image-word label pairs, whereas the example set contains only image-character label pairs. Thus, the weakly-labeled samples have to go through an intermediate step before being absorbed into the example set.

The example set initially contains seed data. We cannot make use of the weakly-labeled samples with an empty example set because the conversion from the image-word label pairs to image-character label pairs require some images to compare against. The seed data are automatically generated by library method calls. There are 5 different entries for each alphabet. Each of the 5 entries corresponds to one of the 5 predetermined fonts, which are chosen so that they cover the different features of characters of different fonts. For instance, some of the chosen fonts are serif, while the others are sans-serif. By making the seed data as diverse as possible, we are more likely to capture a wide variety of new samples from the training set.

We perform several iterations of randomly sampling from the weakly-labeled training samples, filtering training candidates, and expanding the example set with the interpreted training samples. A training log is also kept to establish the mapping between weakly-labeled samples, as well as the elements in the expanded example set that are produced by them. This iteration is repeated until we are confident that the example set allows recognition to be performed accurately.

Finally, not all of the training samples will directly contribute to the expansion of the example set. We introduced several filtering steps which training samples had to go through in order to be included in the expanded example set. The need for filtering training samples arises out of the concern for efficiency and accuracy. The larger the example set is, the slower the recognition step will be, whereas a bad or noisy training sample might mislead the recognition step and reduce accuracy.

With this motivation, we are going to go through the pipeline which converts a randomly sampled, weakly-labeled example to a collection of usable image-character label pairs.

4.2.1 Attempt at Character Recognition

The first check was to attempt OCR the training image itself using the current state of the example set, and to see if the predict result matches the true label. If the result matches, we reason that the current example set is comprehensive enough to identify the image, and that including the training sample in the example set will just be redundant. In this case, we just randomly sample another one.

On the case where the OCR result does not match the annotation, we assume that the annotation is right, and reason that the example set will benefit from including this training sample. Since we are doing random sampling with replacement, there is a possibility that the same training sample might have been included in the example set previously. We reason that there is no point in having multiple entries generated from the same sample. In fact, it might be possible that the previous entries were inaccurate due to bad segmentation. Thus, we go through the training log, and delete all the previous character entries that were produced from the same sample, and allow the training sample to continue through the filtering process.

4.2.2 Guided Segmentation

The fact that our training samples are weakly-labeled introduces the need to detect the mapping between each character in the label to certain segments in the image. The input to this algorithm is then an image and the text corresponding to the image. The output is a list of rectangular regions that, where each region shows a part of the image that corresponds to a single character in the word label.

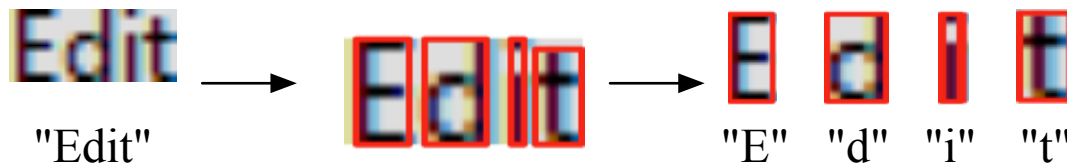


Figure 9. Guided Segmentation Algorithm

For the weakly-labeled samples to be useful, we have to find the mapping between each character in the label and regions of pixels in the original image.

At the core of the algorithm is dynamic programming, and is very similar to the algorithm used in the recognition step. The problem statement is as such:

Input : 1. Word label $w = w[1], w[2], \dots, w[n]$

2. Sample Image Img

Output: 1. $\{region[1], region[2], \dots, region[n]\}$

i.e. the segmentation of Img into n regions such that

$Max\{Cost(region[1]), Cost(region[2]), \dots, Cost(region[n])\}$ is minimized.

The details as to how the cost is calculated, as well as the solution to the problem, are discussed in detail in section 3. The vital difference between this algorithm and the

one used in the recognition step is that this algorithm is provided with the true label itself, and attempts to find the best way to segment the image into rectangular regions such that each region best depicts a particular character in the word label. Recognition step, on the other hand, is only provided with the unlabeled image, and its goal is to find the best segmentation and word label that describes the image.

In conclusion, the result of the guided segmentation algorithm is a compilation of image-character label pairs, each of which is a possible candidate for expanding the example set.

4.2.3 Candidate Filter

The filters so far only act upon the training sample as a single image-word label pair. Now that we have obtained these candidates, these image-character label pairs have to individually go through yet another filtering process before finally being accepted into the example set.

4.2.3.1 Negative Samples Filter

The candidates produced by guided segmentation algorithm are not necessarily right, even if the provided annotation is correct. After all, the segmentation process itself is just an algorithm, and is based on the current state of the example set. Thus, there are many occurrences where we encounter poorly segmented images. With the presence of such poor labels, some of which are slight, but repeated, alterations of similar-looking glyphs with the same character label, the example set might eventually contain a mapping between a nonsensical glyph to a particular character. The proposed counter to this problem is to have a *negative example set*, which contains the supposedly bad samples.

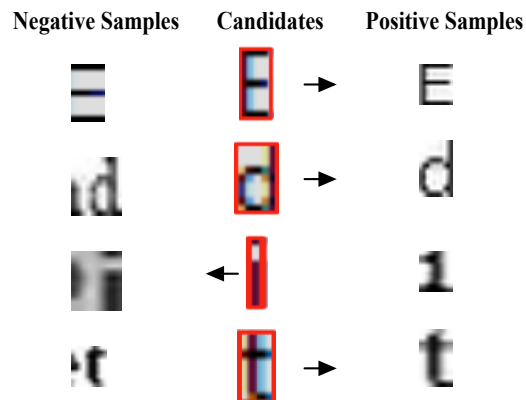


Figure 10. Negative Samples Filter
In this case, the candidate 'i' is closer to a negative sample than a positive sample, and is not added to the example set.

Each segmented region obtained from the guided segmentation step will only be added to the positive example set only if the region's distance (please see section 4.3) to its closest positive sample is less than its distance to its closest negative sample. Collecting the entries for the negative dictionary is for the most part a manual process, since we want to prevent the possibility of a negative example set ever containing a positive example. If this were to happen, all future possible positive entries will be rejected. The process of collecting these negative samples is aided by having a driver that automatically goes through each entry in the positive example set, and checks if its distance to the other positive entries is less than the distance to the negative entries. A list of positive entries that fail this condition will be compiled, and the list will be manually

gone through to see if these suspicious entries should indeed be moved to the negative example set. By expanding both positive and negative example set, we hope to achieve a higher accuracy by detecting bad training samples, and accepting good training samples that are similar to the content of the positive example set.

4.2.3.2 Evaluation Set Filter

The last check on the segmented regions is to see if retaining these regions will improve the accuracy of the OCR over all the possible inputs. Since this is impossible to check, a validation set containing 100 image-label pairs is set aside from the training set. The validation set stays the same throughout the extraction step, and we perform some preprocessing beforehand so that the recognitions step can jump right to the calculations without doing any image-modifications. A record of the best performance on the evaluation set is maintained throughout the training iterations. The remaining candidates in the filtering pipeline will finally be accepted into the example set only if they have not decreased the recorded best performance over the validation set.

5. Experimental Results

5.1 Setup

Measuring the performance of this custom approach is a challenging task due to the specificity of the problem we are trying to solve, as well as the diversity in screen-rendered text. We decided not to compare this custom approach to off-the-shelf OCR technologies such as Tesseract since these solutions are tailored for handwritten text, and do not perform as well on low-resolution text. Since the main purpose of this tool is to identify screen-rendered text embedded in common user interfaces, we limited our testing to images of text which are not italicized, rotated, or distorted in any way. We obtained images of these texts by using Prefab on popular user interfaces and websites such as Windows Explorer and Wikipedia [2].

Out of about 10,000 image-word label pairs, we used 3000 for training, and the remaining 7000 as our testing set. Before experimentation, we have prepared an automatically generated example set of 260 image-character label pairs. We used the training set to expand the example set to contain a total of 1000 image-character label pairs. This process involved some human supervision to ensure that the initial example set contains reliable seed data, and that most of the common segmentation errors were recorded in the negative example set. The expanded example set obtained after training achieved around 95% accuracy over the training set. Although this number does not indicate anything about the performance on the test data, we note that by using only 1000 image-character label pairs, where 260 of which were the seed data which were generated by libraries, we were able to generalize over the entire 3000 words pretty well.

The main strength of this custom approach is its ability to expand its example set. As such, the goal of our experiment is to show how varying the number of training samples affects the accuracy and performance of the OCR. For each fixed number of samples to train on, we repeatedly picked a random subset from the testing set, trained on it *without* any human supervision, and measured accuracy of the expanded example set

on 1000 samples of the remaining untrained subset. Every time we increase the number of training examples, we start afresh with the initial example set of 1000 image-character label pairs.

5.2 Results

As expected, the accuracy increases with the number of training samples. Starting with the initial example set, our average accuracy was 61.76%. We noted that there was a sudden increase in accuracy when we just used 100 training samples, where this caused an increase in accuracy of over 12% to 74.22%. From that point onwards, the accuracy only slowly increases with the training sample. The highest observed accuracy of 86.44% was achieved when 3600 word image-label pairs were used, and increasing the number of training examples to 5000 does not seem to improve or detract the performance much.

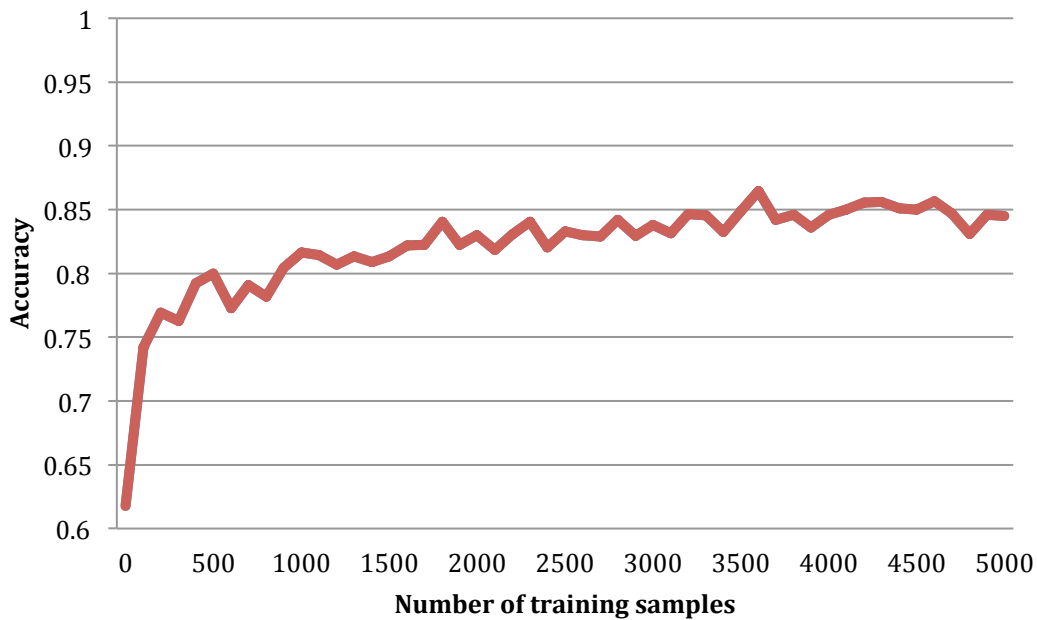


Figure 11. The accuracy of our system varied against the example set size.

5.3 Error Analysis

Despite incorrectly identifying about 15% of the samples, we observed that a majority of the erroneous OCR results can be argued to be understandable, or not that too far off from the true labels. About one-third of the errors can be fixed by the replacement of a single character. The main cause of such errors is the fact that some characters have similar, if not, same pixel representations. Most of these ambiguous pairs are comprised of pairs of lowercase and uppercase forms of the same character, and also pairs of distinct characters that are inherently ambiguous, as can be seen from Figure 12.



Figure 12. An example of an indistinguishable pair of characters

The second major cause of the overall error is due to a single over-segmentation of a character into two characters. Most of such cases involve the splitting of a particular character into that same character, plus an additional vertical-line-like character like ‘l’. The absence of penalty in splitting characters allows the OCR to over-segment text to achieve a lower overall objective cost. Thus, when the OCR is presented with a new representation of a certain character, it has a tendency to indirectly remove some blocks of pixels out of the new image by identifying them as characters like ‘l’. By doing this, it essentially has the ability to crop the new representation to make it look more similar to the representations contained in the example set. These kinds of errors are hard to fix even with large training samples, because the OCR can attain very low distance costs for each segment by performing its own cropping.

If we were to be more lenient in the calculation of our accuracy and permit cases fixable by single character deletion or substitution, we would attain over 10% increase in overall accuracy. With just 100 training samples, the OCR attains over 90% accuracy, and with 4100 training samples, the OCR attains its peak of 95.78% accuracy. This perhaps indicates that most of the OCR results, even the wrong ones, would still be understandable by humans, or easy to fix if the context of the text of were known.

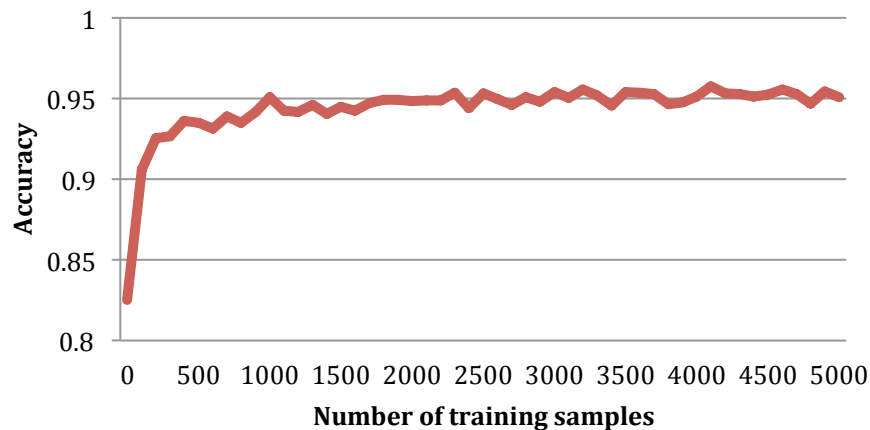


Figure 13. Accuracy graph showing the effect of varying the number of training samples, but discounting errors caused by single character over-segmentation, or single character substitution.

6. Future Work

We believe that handling the previously mentioned major sources of error, which are single-character over-segmentation and substitution, would dramatically improve the accuracy of the OCR.

There have been previous attempts at the recognition low-resolution text, which started off by identifying indivisible chunks of pixels, where no attempts of further segmentation will be made on these regions [4]. Our approach, on the other hand, treats every single column as the smallest indivisible region of pixels. We believe that having a better knowledge on which regions are indivisible will alleviate the problem of single-character over-segmentation, especially in cases where it is very obvious that two regions of pixels are strongly connected such those cases that result in an extra 'l'.

Having a language model may help in both reducing over-segmentation, as well as distinguishing between ambiguous character pairs. We can gather the probabilities of transitioning from a character to another by performing some statistical analysis on common user interface text, which can be more accurate if the domain of the user interfaces to OCR on is confined enough. Since the dynamic programming algorithm identifies a new character incrementally, it is possible to incorporate the probability of transitioning between different characters into its cost calculation, and discourage bad segmentations that result in unexpected words.

We have considered including aspect ratios in our cost calculation. Since our distance calculation automatically resizes all images to be 10x10 in dimension, we lose track of any differences in aspect ratios. Because of this, characters like 'i' and 'l', which have fairly different aspect ratios, are considered to be similar.

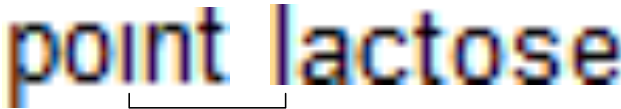


Figure 14. Ambiguity arising due to ignorance of aspect ratios

It is possible to allow the algorithm to take into account the difference in aspect ratios between two images while keeping the distance function Euclidean by incorporating the aspect ratio difference into the dynamic programming cost calculation.

7. Conclusion

We have presented a novel way of recognizing screen-rendered text, which is based on instance-based learning, and relies on dynamic programming for segmentation. With the analysis of the errors, and its accuracy, we believe that our custom approach has the potential for use in many practical applications.

8. Acknowledgement

I would like to thank Professor James Fogarty for allowing me to work on this interesting project. I also received useful advice from Professor Dan Weld, and I am grateful to have Morgan Dixon as my supervising grad student.

9. References

- [1] Morgan Dixon and James Fogarty. 2010. Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '10). ACM, New York, NY, USA, 1525-1534. DOI=10.1145/1753326.1753554 <http://doi.acm.org/10.1145/1753326.1753554>
- [2] Morgan Dixon, Daniel Leventhal, and James Fogarty. 2011. Content and hierarchy in pixel-based methods for reverse engineering interface structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '11). ACM, New York, NY, USA, 969-978. DOI=10.1145/1978942.1979086 <http://doi.acm.org/10.1145/1978942.1979086>
- [3] S. Wachenfeld, H. Klein, and Xiaoyi Jiang, "Recognition of screen-rendered text, " in *Pattern Recognition, 2006. ICPR 2006. 18th International Conference on*, 0-0 2006, vol. 2, pp. 1086-1089.
- [4] S. Wachenfeld, H. Klein, S. Fleischer, and X. Jiang, "Segmentation of very low resolution screen-rendered text," in *Proc. of Int. Conf. on Doc. Anal. and Rec.*, 2007.
- [5] Hourcade, J.P., Perry, K.B. and Sharma, A. PointAssist: Helping Four Year Olds Point with Ease. *IDC 2008*. 202-209.
- [6] Hwang, F., Keates, S., Langdon, P. and Clarkson, P.J. Multiple Haptic Targets for Motion-Impaired Computer Users. *CHI 2003*. 41-48.