Building a Naïve OCR System

Kaur Alasoo, Jaana Metsamaa
University of Tartu - Data Mining 2009

Problem Statement and Purpose

Imagine a situation when you are reading a book in a foreign language and you encounter a new word that you do not know the meaning of. It would be very helpful if you could use your mobile phone to take picture of that part of the page, mark the word in question and have to software to recognize that word and provide translation. The aim of this project was to implement the part of the software that deals with recognizing the word and converting it into text. Although there are many software tools that provide Optical Character Recognition (OCR), it is very interesting to implement one oneself and learn all the details and difficulties involved.

Getting Training Data

To simulate the real situation we took about 40 pictures with T-Mobile G1 phone from a book called Presentation Zen. We limited ourselves to only one book and one font to increase our chances of success. The critical feature of the phone is that it has a 3.2 Mp auto-focus camera which makes it possible to make sharp pictures from a very short distance. The choice of the book was mostly determined by the nice linearly separable font it had. As we will show in the section about pre-processing, some fonts are much easier to separate into letters than others.

After taking the pictures we manually cut out all words from them that were sharp and legible enough. Manual extraction was needed, because when pictures are taken with a mobile phone camera held so close to the text, only words in the center of the photo are in focus and other can be severely distorted and skewed [Figure 1]. In the end we had pictures of 242 words which are also available for download from the project website[8]. Each image of a word was named in the following format: "word-index.png" - eg. "says-1.png", "effect-2.png" and so on.

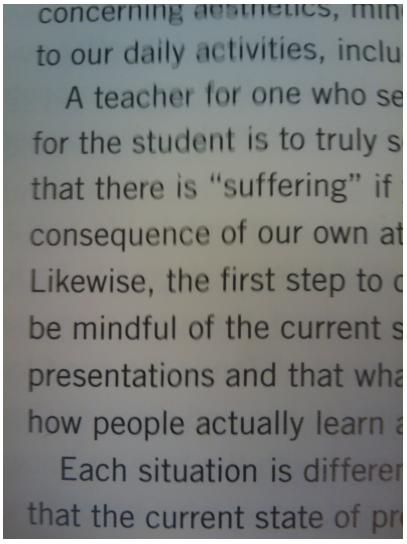


Figure 1: *Sample of input "snapshot".*

Pre-processing

We used OpenCV [3] open source computer vision library to process the images so that features for each letter could be extracted.

The pre-processing of the pictures consisted of the following steps:

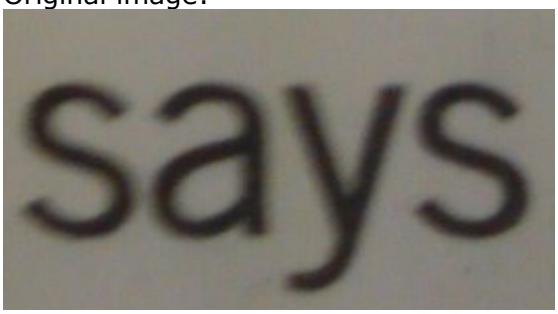
- Convert the picture to 8 bit gray scale image.
- Apply thresholding to get only black and white picture.
- Cut out the letters.

Converting to Gray-scale

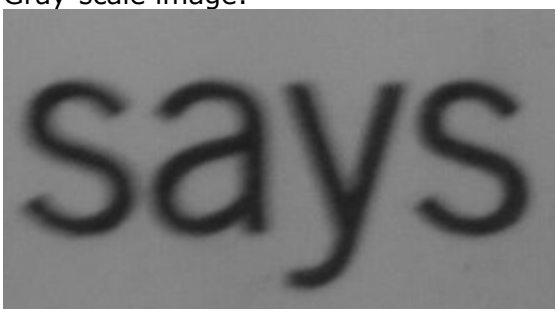
Conversion to gray-scale can be done in OpenCV with three lines of code and by using the `cvCvtColor()` function.

```
    IplImage *img = cvLoadImage("words/says-2.png"); //Import original
picture.
    IplImage* out = cvCreateImage(cvGetSize(img), IPL_DEPTH_8U, 1); //Create
new image with original image dimensions
    cvCvtColor (img, out, CV_BGR2GRAY); // Convert original image to gray-
scale and save it to new image variable.
```

Original image:



Gray-scale image:



The reason we had to convert our image to gray-scale was because thresholding could be applied to monochrome pictures only.

Thresholding

In an 8 bit image each pixel is represented by one number from 0 to 255 where 0 is black and 255 is white. The simplest way to convert the image to black and white pixels would be to select one value, lets say 128 and consider all pixels that have higher value to be white and the others black. The biggest problem with this approach is that the brightness can vary from picture to picture and as a result some images might become totally black while others are entirely white.

The solution to this is something called adaptive thresholding. In OpenCV it is implemented in a function `cvAdaptiveThreshold()`. The idea is very simple. For each pixel in the picture you look at *n* neighbours of it and calculate their mean brightness. If the value of the current pixel is higher than mean of the neighbours - *k*, then we say that the pixel is white. Otherwise it is black. *n* and *k* are parameters chosen by the user and in our case we observed that the suitable values were $n = 101$ and $k = 5$ (default).

The code that we used to get the result on Figure 2:

```
cvAdaptiveThreshold(out,out,255,CV_ADAPTIVE_THRESH_GAUSSIAN_C,CV_THRESH_BINARY,101,5);
```

Figure 2: *Result after adaptive thresholding*

Cutting Out the Letters

We used a very naïve approach to separate individual letters from the words. First we scanned the vertical columns of pixels from left to right and stored the numbers of the columns in which all pixels where white. In the source code that is available from the project website[8] it is accomplished by the method `findY()`. Next we looked at the vector

of column numbers and determined where they are interrupted (`findCutoffs()`). These points became the cutting points. An example is shown on Figure 3.

Figure 3: *Vertical cutting points.*

After doing the vertical cuts we apply the same algorithm to pixel rows to remove the white areas from above and below of each letter separately.

We admit that this approach is not perfect and in fact works only on fonts that have linearly separable letters. For example when we take a picture from another book (Figure 4), then our algorithm breaks down, because f, f and e cannot be separated anymore by a straight line. One possible solution would be to look at areas of connected pixels and try to do the cut based on that information. Another is proposed and implemented in Tesseract OCR, that we also cover later in this report. Totally different approach to OCR is proposed by Lu *et al*. in [1] that does not require separating the letters at all.

Figure 4: *f, f and e cannot be separated by a straight line.*

Feature Extraction

Now that we have each letter cut out we need to convert them to feature vectors that can be used as an input to classification algorithm. First thing that we noticed was that it is very easy to find the aspect ratio for each character and we decided to use it as the first feature.

Another obvious thing to look at are the pixels. The simplest approach would be to use the values of all pixels as the features. However, the problem with this is that different letters have different number of pixels in them. In addition, the number of pixels per character also varies form photo to photo depending on the distance it was made from.

The solution that we came up with was to resize the picture corresponding to each character to 16 x 8 pixels. These numbers are actually quite arbitrary and we chose to use them

mainly because the same dimensions are used in a publicly available OCR data set obtainable from [9]. Using 18 and 9 or anything reasonably similar would probably not change the results too much. The resized picture for one letter is shown on [Figure 5]. In OpenCV the resizing is done using `cvResize()`, which in its default behaviour uses bi-linear interpolation. It would also be possible to use nearest neighbour interpolation, which would result in only black or white pixels.

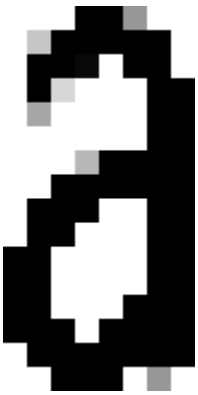


Figure 5: *Letter a after resizing.*

To get the feature vectors we used the `cvSave()` function to save the picture as an XML file. An example is shown on Figure 6. After that we just had to write a small Python script that takes all XML files and converts them into one comma-separated (CSV) matrix. The final data matrix is available for download from the project homepage [8].

```
<?xml version="1.0"?>
<opencv_storage>
<!--
Aspect ratio: 0.774194
Letter: a
-->
<letter type_id="opencv-image">
  <width>8</width>
  <height>16</height>
  <origin>top-left</origin>
  <layout>interleaved</layout>
  <dt>u</dt>
  <data>
    255 255 255 0 0 151 255 255 255 199 0 0 0 0 0 255 255 0 0 8 255 0 0
    255 255 0 215 255 255 255 0 0 255 167 255 255 255 255 0 0 255 255
    255 255 255 255 0 0 255 255 255 183 0 0 0 0 255 255 0 0 0 0 0 0 255
    0 0 0 255 255 0 0 255 0 0 255 255 255 0 0 0 0 255 255 255 255 0 0 0
    0 255 255 255 255 0 0 0 0 255 255 255 0 0 0 0 0 0 255 0 0 0 0 255 0
    0 0 0 0 0 0 255 255 0 0 0 255 151 255</data></letter>
</opencv_storage>
```

Figure 6: Output XML file with a feature vector

The obtained data is best characterized by the following plot [Figure 7]. The frequencies of letters in our data are depicted by light green, dark green shows the general character frequencies in English. As we can see, collecting training data by randomly taking snapshots from pages in books does not do any justice to infrequent letters like j, x, q and z. That is why in Tessarect [6] they actually use artificially generated training data that has all characters and symbols in many fonts.

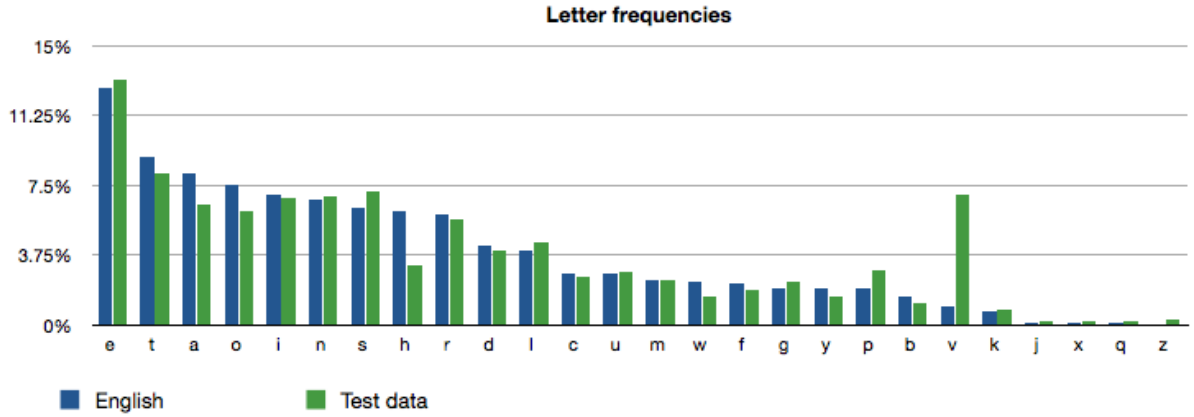


Figure 7: *Word frequencies in the English language compared to our test data.*

Training the Classifier

We decided to use Support Vector Machine (SVM) classifier with Radial Basis Function (RBF) kernel, because it was shown by Lin and Fujisawa [2] to outperform other classification methods. In addition, `e1071` package for R [7] made it very easy to implement. After performing grid search and trying several values for sigma and cost parameters we obtained the best results with `sigma = 0.001` and `cost = 6`. In 10-fold cross-validation the average character recognition accuracy was 98.86%.

Tesseract OCR

Tesseract [6] is a free optical character recognition engine, originally developed as proprietary software by Hewlett-Packard between 1985 and 1995. After ten years without any development taking place, Hewlett Packard and UNLV released it as open source in 2005. Tesseract is currently developed by Google and released under the Apache License, Version 2.0. Tesseract is considered one of the most accurate free software OCR engines currently available. We wanted to know how our OCR system compared to it and tested it on the same words as our own.

Although Tesseract is a full-blown OCR system for recognizing big chunks of text from books or scans, there are some interesting ideas and methodologies that could also improve our system that concentrates on individual words.

Currently we do not do any baseline adjustment or recognition. This means that if a word is not correctly aligned horizontally, the accuracy drops. Tesseract does not only recognize the baseline of a diagonal text, but it is also able to correctly determine curved lines (that can occur very often during scanning). We think that de-skewing words before cutting would improve our accuracy with some of the words that we are not able to recognize at the moment and it would probably be simpler to implement than a line finding algorithm such as the one in Tesseract.

Our system was initially restricted to one font and to pictures taken with a phone. While trying to test the system on another font we realized that our bottleneck was the letter-cutting-algorithm that assumes linearly separable fonts and otherwise just puts many letters together. Tesseract does not do any permanent cuts and allows itself to go back to a

previous point if the cut does not improve recognition confidence. Possible cutting-positions are illustrated with arrow-heads on [Figure 8]. This makes Tessarect also able to recognize characters that are not separable by a straight line.

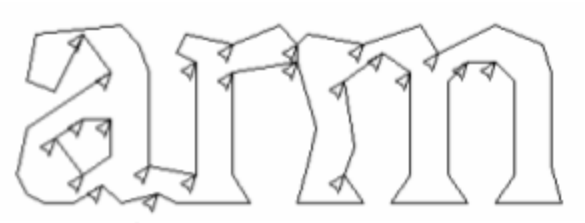


Figure 8: Finding good letter-cutting positions.

Another idea implemented in Tesseract is linguistic analysis. When unsure of its accuracy Tesseract makes the final decision choosing from the best available words from seven classes - Top frequent word, Top dictionary word, Top numeric word, Top UPPER case word, Top lower case word (with optional initial upper), Top classifier choice word.

Results and Comparison

To compare our system with Tesseract we had to install Tesseract to our system which was actually quite simple. But when we tried to use it on our pictures we run into the following error with many files:

```
check_legal_image_size:Error:Only 1,2,4,5,6,8 bpp are supported:32
```

We tried to solve it by converting all pictures to 8 bit TIFF images using OpenCV, but it did not help. The same files still did not work neither in our computer nor on the Free-OCR.com installation of Tesseract. We found that changing something in the source code and recompiling could help, but unfortunately we did not have time to test it.

Luckily 80 pictures out of the 242 still worked with Tesseract and based on them the character recognition accuracy was 91.47%. In comparison, 10-fold cross-validation accuracy on the all data with our method was 98.86%. Interestingly Tesseract tended to make more mistakes with shorter words which could be explained by the fact that its line recognition algorithm does not work very well in that case.

Obviously, if we had trained Tesseract on our data we would have got better results, but the weakest part of it, as also mentioned in [5], is probably its classification algorithm and newer methods such as SVMs tend to work much better.

Conclusion

Partly because of the stringent constraints we set in the beginning, we were able to build an OCR system that is able to recognize characters with a very high accuracy (98.86%) and in this specific task even outperforms general purpose OCR. If one needs an OCR system for a specific tasks such as recognizing letters in individual words or recognizing car number plates, then building a dedicated OCR system might be quite reasonable.
On the other hand, developing a program that is able to understand scanned documents in many different layouts and fonts is much more difficult and requires a lot of work. In this

case it might be reasonable to use some existing solutions (such as Tesseract) and modify them according to your needs.

References

1. Zhidong Lu et al. *A Robust, Language-Independent OCR System*, Proc. SPIE, 1999
2. C. Liu and H. Fujisawa, *Classification and Learning for Character Recognition: Comparison of Methods and Remaining Problems.*
3. OpenCV homepage (http://opencv.willowgarage.com/wiki/)
4. OpenCV Reference Manual (http://www.seas.upenn.edu/~bensapp/opencvdocs/ref/opencvref_cv.htm)
5. Ray Smith, *An Overview of the Tesseract OCR Engine*, Proc. of ICDAR, 2007
6. Tesseract homepage (http://code.google.com/p/tesseract-ocr/)
7. e1071 package for R (http://cran.r-project.org/web/packages/e1071/index.html)
8. Project homepage (http://kaur.pri.ee/projects/wordocr)
9. OCR dataset (http://ai.stanford.edu/~btaskar/ocr/)