Mateusz Kulinski (6424196) – mk00756

COM2028 Report

Table of Contents

1: Introduction

The objective of this report is to describe the findings of the investigation on optical character recognition (OCR), particularly for recognising printed pages, handwritten pages, and signatures. OCR has long been an issue, with multiple different ways of approaching the problem. There are various different methods which have been tried in an attempt to convert written characters into digital ones. In this report I will be covering multiple machine learning methods which can be used to solve this problem.

2: Analysis

One of the first key elements to consider when attempting to build an OCR system is the data to be extracted. The data needs to be altered in such a way that it preserves its integrity and meaning as much as possible, while allowing for a computer to read and manipulate the data. This invloves a number of steps, the first of which is digitisation.

Precautions should be taken when digitizing a written or printed page with a scanner, as multiple issues may arise. The scanner may pick up any foreign entities within the input (such as stains on the page, crumbs attached to the page, ink which is not part of any characters, including various other possibilities) and convert this to unwanted noise within the digitised data. Some post-processing can be done after this step to remove noise, such as manually inspecting and altering the file generated by the scanner such that the noise is no longer present, or invoking a gaussian kernel upon the image to reduce the noise such that it is not perceptible. It is noteworthy that the latter approach will also alter the data itself, albeit only slightly. I personally made sure to watch out for anything that could produce noise once scanned when scanning the data I would later use for training/testing.

Once the data has been digitised, the data needs to be segmented. “The Segmentation phase is the most important process”[1]. As described in [1], there are many issues that arise at this stage, especially with handwritten characters. I shall describe the measures taken to avoid certain issues. To make the extracting of letters as simple as possible (for both handwritten and printed data), I

used non-joined characters. To easily automate the extractions of these characters such that they were easily sorted, I made sure that the characters appeared in a defined sequence.

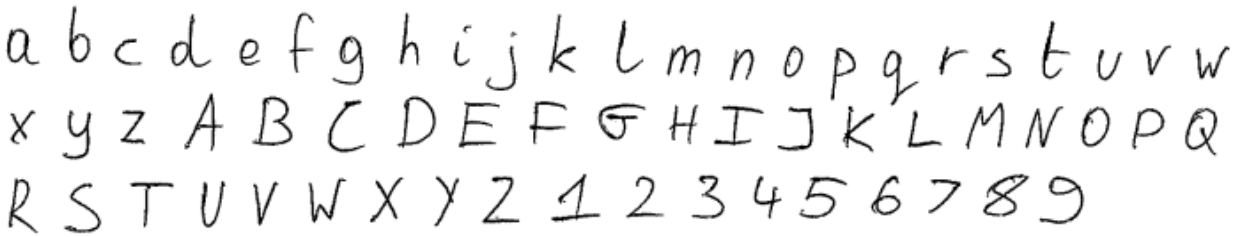

Figure 1 – handwritten sequence of seperated characters

I had an issue with entering handwritten data in particular, as I found out that the skew of my writing had caused issues within the code used to extract the characters and sort them. To overcome this I decided to write only one sequence of characters per page, rather than a couple of these sequences, to make sure I minimise the skew in my writing. Despite this, some of these pages still caused issues with the extraction system. These pages were abandoned when it came to testing and training the recognition systems.

I used a copy of the code used to generate the training data to produce testing data. The testing data did not need to be fully organised, therefore I just let the algorithm output to a single directory rather than producing a seperate directory for every single character, as I have done for the training data. The testing data still followed the sequence shown in Figure 1.

After the segmentation step, both the training and testing data were ready to be used by the machine learning algorithms in order to produce an OCR system. I had decided to experiment with three such algorithms. The algorithms I tried were the ‘K nearest neighbours’ algorithm (KNN), an ‘artificial neural network’ algorithm (following the multi-layer perceptron model) (NN), and the ‘support vector machine’ algorithm (SVM). I used all of these algorithms on both printed and handwritten data, the code remaining largely unchanged to do this. The next section will be describing the steps required to use the algorithms for the OCR task for both printed and handwritten characters. Signature recognition will be described later.

The KNN algorithm required three inputs in order to train. Firstly, the number of neighbours (as indicated by the ‘K’ in KNN). I had experimented with multiple different values for this parameter, and I found that using the three nearest neighbours seemed to produce the best results for the data I had. A list of the character images had to be provided, along with the target values, as the KNN algorithm is a supervised learning system. The indexes of the lists had to match up with each other, meaning that if the first index in the data list was the character ‘a’, the first index of the target list needed to also be an ‘a’ (or something that can later be interpreted as ‘a’). An elegant solution to ensuring that targets always match up with inputs was provided in the segmentation step. Every character that was extracted had a unique filename, the filename followed the pattern of ‘character’ followed by the number of previous occurances of this character. This meant that when I was reading in ‘a’ characters, every single one belonged to the ‘a’ directory and also had ‘a’ in their filename.

Figure2 – The images contained within the ‘training_set/E’ directory

The NN algorithm required multiple variables in order to train. I have left the majority of variables set to the default setting of the ffnet package. I had set the ‘max_fun’ variable to ‘10000’ to ensure that the NN algorithm stops training after this many iterations. The idea behind doing this was so that the algorithm stops naturally when it reaches a certain degree of change (before it hits 10000 iterations), as defined in the ffnet package. Reference [2] describes vividly how a neural network works, I used it to see what parameters would work best for this task. Since we are using images of size 20x20, and are training the OCR system to recognise 61 different characters, I set the input parameter to 400, and the output parameter to 61. At first glance it would appear to make sense to add in a hidden layer to aid the system in interpreting the features of the characters so that it may recognise them better, however I decided not to include a hidden layer, as I seemed to get the best results without it. Like with the KNN algorithm, I needed to provide lists of data and respective targets, however the targets needed to be in vector form in order to be usable by the NN. After entering all of the above I was able to train a neural network and save it for further use.

The SVM algorithm was perhaps the simplest to use, requiring only the data and targets to train. However upon further inspection, the algorithm would not want to converge (even after 20k iterations) when given this data. To overcome this obstacle, I have used the histogram of oriented gradients (HOG) method to extract features out of the character images, and used the hog features as the data instead. I used just two pixels per cell for HOG, to ensure that the features are of high quality and precise, however this is computationally expensive for large datasets. Now the SVM would train completely fine, and also quickly.

After training all of the algorithms described above, I tested them on a testing set. The below tables show the results on handwritten and printed data.

Algorithm	Accuracy
KNN	86.1566484517%
NN	83.4244080146%
SVM	90.3460837887 %

Table1 – Recognition accuracy for printed data for three different algorithms

Algorithm	Accuracy
KNN	63.9344262295%

NN	36.0655737705%
SVM	86.8852459016%

Table2 - Recognition accuracy for handwritten data for three different algorithms

As we can see from these results, the SVM algorithm seems to provide the best recognition accuracy. Note however that SVM is the only algorithm using HOG features for data, and this may be the reason why it is performing so well, I would have liked to test this however I did not have the time. I also think that overfitting may be occurring when the NN is trying to recognise handwritten data, due to the rather low recognition rate when presented with new training data. The handwritten set was also very limited with the amount samples, as well as only including characters written by myself. Ideally I would be able to find a larger set of character, from different people.

Now came the task of recognising signatures. Unfortunately, I was not able to get the signatures of another person, therefore I decided to write two different signatures myself. This might even prove to be an added challenge for the signature recognition system as the two signatures will have handwriting similarities.

To complete this task I decided to use the SVM code used for the previous two tasks, however I modified the segmentation system to produce images of size 100x25 rather than 20x20, as these dimensions were suited better for a signature. The segmentation code could largely be re-used as signatures can be interpreted as lengthy characters (so long as they do not contain a space). I had not modified the code greatly, therefore instead of producing a testing and training set out of the images I entered, I instead produced only a training set and then manually removed parts of the training set to be used as the testing set. The SVM code also needed slight adjustments in terms of the targets, as there would now only be two targets, one for each of the different signatures.

After creating the code for recognising signatures, I tested it. The SVM system was 100% accurate in identifying the signatures. This is highly inconclusive as the data entered to the system was very limited and had very little variation. I simply wrote out two different signatures, thirty times each, and then took three signatures from the thirty I’ve written to be the test set. The remaining 27 were used as the training set (NOTE: I manually removed images produced by the segmentation system which did not resemble the signatures, as I did not have the time to fix the segmentation system so that no errors would occur). I would definitely like to revisit this system and try it out with a greater number of unique signatures.

Regarding the coursework challenge, I was not able to produce any results or find a partner to exchange handwritten data with. However, I will describe the workflow I have planned to solve this task.

I would gather several handwritten notes from multiple people, making sure I label which notes came from which person. Then, I would extract the words out of a page of notes, and generate an ‘Average word’ for that person’s page of notes. This would eventually result with me having multiple average words for each page of notes of every person. I would then use the HOG to extract

features out of each average word, and feed it into an SVM along with the target (the person who wrote the page from which the word was derived). Then to test the SVM I have just trained, I would take a new page of notes from each person and generate a new ‘average word’ to use for testing. If all went well, the system would correctly recognise the person who wrote the page of notes from which the average word was derived from.

3: Conclusion

The SVM algorithms seems to be by far the best algorithm for the OCR task, out of the algorithms I had experimented with. It produced the best results for both printed and handwritten data, and was also the only algorithm capable of producing results for the signature recognition task.

There are various ways to improve on the work done here. The segmentation code was quite crude and did not always produce the best images for the OCR task. I hypothesise that, with better quality testing and training images, as well as much larger sample sizes, the results would be quite different than the ones presented here.

4: References

[1] H.Modi, M.C.Parikh, “A Review on Optical Character Recognition Techniques”, [Online]. Available: https://pdfs.semanticscholar.org/41b3/f9b53993d97674102e76f80ebb7feea9761b.pdf (accessed 3 May 2018).

[2] N.Ventaka Rao, DR. A.S.C.S.Sastry, A.S.N.Chakravarthy, Kalyanchakravarthi P, “Optical character recognition technique algorithms”, [Online]. Available: https://pdfs.semanticscholar.org/18db/29a7ae4a6abb4c0bc7b76d26c627ca7d58e3.pdf (accessed 3 May 2018).

5: Appendix

I have provided a .zip file which contains all of the source code, as well as all of the testing/training images I used, and the unsegmented images from which sets were extracted.