Historical Handwritten Word Classification using Trie Search on Autoencoded Character Segments

B.J. Wolf

University of Groningen, Department of Artificial Intelligence and Cognitive Engineering

Abstract: In this paper a pipeline is described which is used to classify historical handwritten texts. Using various preprocessing techniques, words are segmented, segments are stitched and classified as characters using a stacked denoising autoencoder and several classifiers. Finally a graph search incorporating language-specific character bigram models is used to find the best hypothesis among the character candidates to finally form a word hypothesis. Various systems are tested and initial results show that 57.10 % of 6035 words are classified correctly with a mean edit distance of 1.10 using a 3rd degree polynomial SVM.

Introduction

Handwriting recognition is a popular field of research and can be discerned in various classes of problems depending on the style, script, domain, whether online or offline data is acquired and most importantly the unit of interest; ranging from stroke to character, to word and beyond (Arica, 2001). Variations in style include cursive, connected, and handwriting 'font'. The most common researched script is Roman script, but more challenging scripts include Chinese, Modi and Arabic, where characters are share more morphological features.

The domain can be important for optimization, for instance when only digits need to be recognized, the confusion between a roman character 'O' and digit zero '0' can be disregarded. By reducing the domain, performance in recognition can be boosted, so handwriting recognition systems have the tendency to perform well in domain specific tasks. Fortunately general methods from either class of handwriting recognition can be used and tested in other classes of problems, a merit for the research field.

The proposed pipeline is designed to classify words based on character classification results of submodules. In the Methods section, the dataset is described, along with the modules in the processing pipeline and the experiments on optimizing the character feature extraction method, character classification and word classification. The results are displayed in the Results section and discussed in the Discussion section. Finally a short review on the group project is given.

Methods

In this section dataset and the word recognition processing pipeline will be discussed in detail, comprising of three parts, character recognition, word segmentation and word classification. Finally the setup of the performed experiments are presented to be discussed in the following sections.

Dataset

In order to test our handwriting recognition system, we have trained our recognition system on two distinct data sets, consisting of handwritten Roman words. These data sets are the Stanford data set and the KNMP data set. The Stanford data set is a set consisting of 46 pages of data, whereas the KNMP data set is a set consisting of 74 pages of data. The words and characters in these data sets have been manually cut and labeled, to ensure that the training labels are correct.

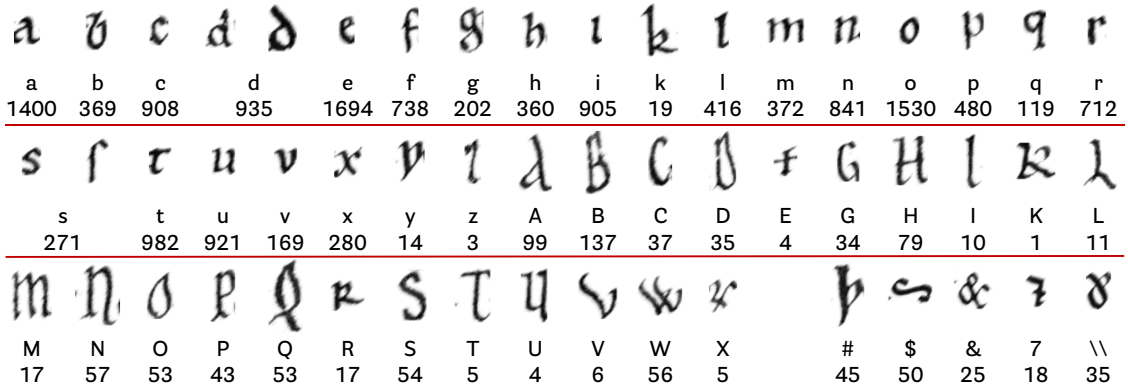

Table 1. Preprocessed sample images of the dataset, with their label and number of samples. Note that two classes have two types (d and s). Morphologically there's some unwelcome overlap between p,y,#; a,A,d; i,l,I; f,s_2; and d_2,\\.

The annotations yield 5414 words and 16223 selected characters with 50 separate labels. The character counts are not evenly distributed, especially the capital letters are low in numbers, which poses a problem when looking to optimize an algorithm with kfold validation.

In order to create more data, morphed variants are created using a sinewave distortion. In particular a mesh with resolution r = 10 is distorted using a sinewave function which uses random uniform parameters. From the displacements, the new image is rendered using Shepards transform (1968). For each gridpoint P_{ij} the displacement d_i, d_j is given by formula 1.

$$d_i, d_j = \sin\left(\left(o + p \right) \times a \right) \tag{1}$$

Where the amplitude a is uniform randomly drawn from [1.5 4.5], the period p is drawn from [0.04, 0.1] and the offset o is drawn from [0 $2\pi/p$], which creates distortion vectors for each gridpoint P_{ij}. A selection of rendered images from an example image is shown in Figure 1.

Fig. 1. The original image of character 'K' (left) and four morphed character using a sinewave mesh distortion and Shepards rendering.

Processing Pipeline

The whole pipeline consists of roughly three parts, working together to eventually form a word classification result. These three parts are referred to as the *character* pipeline, the *word* pipeline, and the *path* pipeline. A flowchart of the whole process can be found as Figure 2.

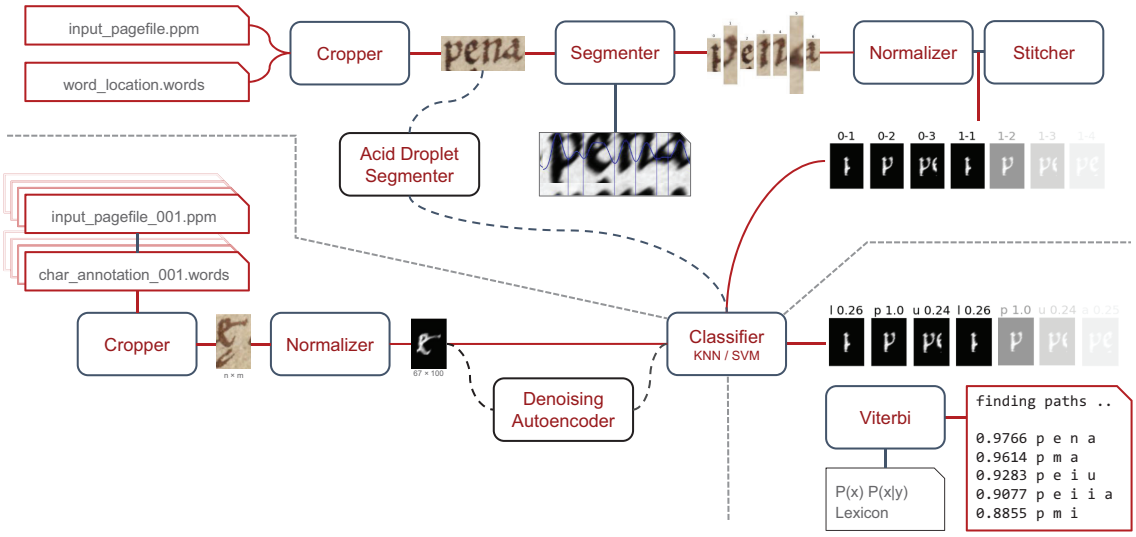

Fig. 2. The pipeline consists of a character pipeline section (bottom left) which fetches character images and trains a classifier. The words (top) are first segmented to be finally stitched to character candidates, undergoing the same normalization process. The path pipeline (bottom right) keeps multiple hypotheses for character candidates and constructs a graph which is searched by a Viterbi algorithm.

Character Pipeline

The character recognition process mainly consists of the following phases: preprocessing the image, using a stacked denoising autoencoder to extract the features from the image, and train a classifier on the preprocessed characters. The preprocessing consists of roughly five steps, see also Figure 3.

Preprocessing

To begin the pipeline, the annotated training set is first normalized. This ensures that all the known characters and character candidates are comparable. The first step is to convert the color images into grayscale images, by combining the rgb-channels into a single channel gray scaled image using a [.229 .587 .114] distribution, as is common practice in computer vision (Timar, 2003). The intensity values range between 0 and 255.

Since the to be described recognizer should (only) be able to recognize handwritten characters, a defining characteristic from these characters is used in our advantage. One goal of the preprocessing step is to eliminate less-information holding speckles in the image and increasing the quality of the details.

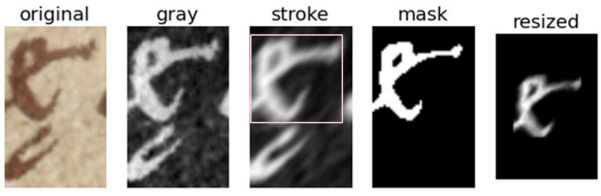

Fig. 3. The character preprocessing pipeline consists consists of the steps gray, stroke kernel convolution and refitting a cutout character on a fixed size canvas (67 × 100) modeled after the largest character in our dataset.

For this purpose, a stroke kernel is created, which closely resembles the slanted pen stroke used to create the characters. The concept is that ink spots which cannot be created by such an stroke will be blurred out. Since the bulk of the mass of the character consists of vertical bars, most of the information of said character is stored in the 'terminals', where these vertical bars connect. The

difference between for instance ‘in’ and ‘m’ comes down to whether the connection between the first and second bar can be properly detected. Convolving an image with this stroke kernel enhances these slanted connections while reducing the width of the vertical bars, which is useful because the vertical bar structures do not contain as much information.

Otsu's Method (Moghaddam, 2012) is an algorithm for binarizing images which can be helpful as a first step towards doing recognition. The problem with Otsu's Method is that a lot of details are lost in this process. The next step is convoluting the grayscale image with the stroked Otsu mask. This leaves us with a single channel image with intensity values ranging between 0 and 1, where 1 denotes all ink and 0 denotes the background. This is useful for further processing, since classification algorithms favor inputs in this scale. Finally we create a bounding box based on the binarization and center the remaining image on a canvas (67×100 pixels) using the centroid of weight.

Stacked denoising Autoencoder

The preprocessed characters – now consisting of 7600 pixels, and thus features – are reduced to a feature vector using an unsupervised dimensionality reduction algorithm, in this case a Stacked denoising Autoencoder (SdA), implemented with Theano[1] to be trained using GPU’s (Vincent, 2008).

The SdA is essentially a Multi Layered Perceptron with some added properties that make it useful for dimensionality reduction. It is comprised of building blocks, denoising Autoencoders or dA, which are an expansion of the Autoencoder. An Autoencoder has an input x and weight matrix W along with a bias vector b, yielding an output vector y (2). Rather being trained through backpropagation, the Autoencoder is trained on the so called reconstruction error. The reconstruction z is defined as the output vector y multiplied with the transposed weight vector, using tied weights allows for reusing the bias vector (3). The reconstruction error, or cost, L is similar to the negative log likelihood, in essence the Autoencoder is optimized for storing as much variance as possible (4).

$$y = \sigma (W x + b) \tag{2}$$

$$z = \sigma (W^T y + b') \tag{3}$$

$$L = - \Sigma (x \log z + (1 - x) \log (1 - z)) \tag{4}$$

$$W' = W - \lambda W^T L \tag{5}$$

Now having the reconstruction error L the weights W of the Autoencoder are updated through calculating the gradient of the error with respect to the weights contribution and a learning rate λ (5).

The addition of the denoising Autoencode is to distort the input vector x with uniform noise [0 1], usually around 0.1 or 10%. The idea of the SdA is then to train one dA layer and treat its output as the input for the following layer. So each dA layer is pre-trained to reconstruct the output of the previous layer, eventually chaining a whole encoding to a smaller dimension and reconstruction from that smaller dimension.

[1] http://deeplearning.net/software/theano/

The weights of the SdA are initialized randomly when the phase called pre-training starts. In this phase, an input image is fed through the first layer of the network and reconstructed using the tied weights. Each layer is trained until convergence.

After the pre-training phase, the whole SdA is used to calculate a reconstruction error on the input image. So an image is fed through each layer and encoded. Then the reconstruction (or decoding) process commences in which the same weights are used to get back to a vector of identical length of the input. Backpropagating the reconstruction error finetunes the SdA to optimally encode the shown examples. Several layer sizes and configurations have been tested, as described in the experimental setup.

Classifier

Now we have a smaller length feature vector, we train an 3rd degree polynomial SVM provided through Python Scikit-learn as an implementation of libSVM. Normally with an SVM one can only get back the best classification result, without any (meaningful) level of confidence, a distance measure is lacking.

This problem is solved by making a statistical version of the SVM, which does elongate the training and classification process, but provides a fundamental basis for the *path* pipeline described in the next section. In order to get a probability from these support vectors, the algorithm repeatedly classifies the same input using gradually fewer support vectors. E.g. when an image is classified as 'g' with 1000, 750, 500, 250 support vectors, but is classified as 'o' when left with 50 vectors, we can estimate a probability of, say, at least 75%. Details of this process can be found in (Pedregosa, 2011).

Apart from the polynomial SVM, other classifiers are tested including QDA, LDA and SVM_{rbf}

Word pipeline

This process is performed for all of the pages of the data: the preprocessing algorithm segments the characters into segments, which are stitched back together in (overlapping) stitchings. Some stitchings are considered character candidates. These are called candidates simple because the design of this stitching process allows for conjoint or parts of characters to be classified. The hypothesis is that true characters (which are complete and don't have other character pixels) will be classified with a higher confidence level, which should increase the probability of correctly stitched character segments being used for the final word classification.

Segmentation

For our character segmentation step we use the concept of over segmentation. The idea is to split the word image into segments, making sure that all characters are separated from each other. The downside is that characters themselves might be split in the process (causing more segments than characters to be formed). This step is important for the character classification step, as the classifier expects single characters only.

A horizontal density histogram is made from a grayscale word image, where full ink has value 0 and paper has value 1, so inversed from the characters (for now). Then a 12px hanning window smooth function is used to transform the histogram into a curve. The smoothing ensures that we don't end up with a very large number of segments, while still cutting on at least character. Vertical cut lines are obtained from selecting the minima from this curve. Finally the original word image is cut into segments with the obtained cut lines.

After the segmenting process, the resulting segments should be stitched back together with other (subsequent) segments to form stitches. At this point, the algorithm does not know which segments need to be merged with each other to form the actual character candidates. It is known that given all the different possible combination of merged segments, the required combinations (the individual characters) are present. The following heuristic is used for merging the characters:

- Only subsequent segments are merged together.
- Segments smaller than the width of the smallest known character, 12 px, are required to be merged with other segments, as they are unlikely to be a single character on their own.
- Merged segments may not exceed a certain width that corresponds to the maximum width, 76 px, of the known characters.

This results in merged segments which are called stitchings (Fig. 3). The stitchings are then classified in the character classification step – after normalization – to form character candidates.

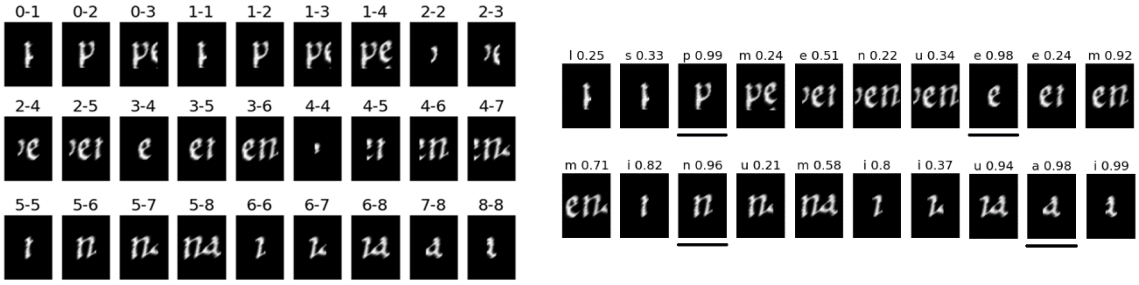

Fig. 3. The stitchings (left) show which segments are used. In the case of the word 'pena', we have 9 segments which can be stitched into 27 stichings. After classification some stitchings are promoted to character candidates (right). A threshold on a confidence level above 5 times chance is imposed to prune unlikely hypotheses in the set of character candidates. The underlined images are the annotated characters.

To deal with the different dimensions of the character and character candidates, all white space (i.e. zero-values) around the character is eliminated. At this stage, some noise may have been included in the image. Especially the roman 'r' (s) character has some unwelcome overlap with the next character.

To eliminate these cut segments of other letters, the vertical bounds of the centermost connected component are used to further crop the image. The horizontal axis is not cropped to make sure that wrongly filtered 'm' characters do not end up as 'i' and 'n' both with an 'm' annotation. This could distort recognition efforts later on, presenting the image of 'in', but the tree-search algorithm for the possible recognitions copes with this. The final cutout is then centered on a 76 × 100 pixel canvas, matching the specifications of the characters that have been used for training the classifier.

Path pipeline

After a word has been segmented into segments, the segment stitched into stitches, and the stitched classified as character candidates; the path pipeline sets to efficiently traverse the resulting search space to find the sequence of character candidates that have the highest likelihood and form the word classification hypothesis.

Trie construction

The character candidates contain spatial information, i.e. which segments are used for the particular candidates. This prevents the search space to have paths that use segments twice (Crochemore, 2007). There are two kinds of variation that can be found in this search space; all sequences of character

candidates that are valid with respect to the spatial information and the multiple hypotheses per stitching.

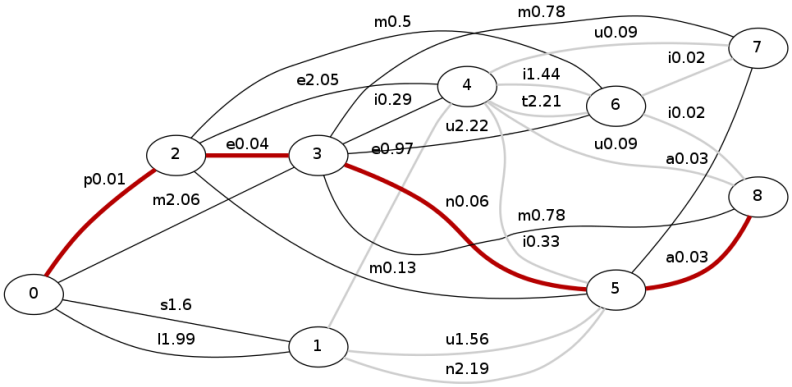

Fig. 4. A visual representation of the trie structure for the word 'pena'. Node 0 is the starting node and node 8 is the terminal. The edges represent the hypotheses of character candidates along with their negative $\log_2$ probability. The bold red edges form the optimal path (lowest cost) through the search space. Black edges have been visited and grey edges have not. Hidden from this representation are the transition probabilities based on a bigram lexicon model.

Trie search

Consider as an example Figure 4. There are two edges from 0 to 1 using the same stitching but form two character candidates. Both the 'ɾ' (s) and 'l' are considered possible labels. This is an increase in search space due to multiple hypotheses. A frequently occurring variation in the paths of sequential character candidates can be seen in the structure of nodes 3, 4, and 6. These three nodes represent that two subsequent 'i' stitches can also be stitched together to form a probable 'u'. These kind of morphological variations occur often and are corrected through lexical information. i.e. three sequential 'i' character candidates are more likely to form 'in', 'ni' or even 'm'.

The search algorithm is set to find the path through the trie with the lowest cost. First neglecting the transition probabilities, the cost for a path is defined as the mean cost of its edges, correcting for length. This design accomplishes three things. First the cost is scaled in such a way that the lowest character probabilities have the highest cost, enabling searching for the path with the lowest cost instead of the highest probabilities. Secondly by using negative log probabilities, instead of just negative probabilities, the final measure is not a very small probability, but rather a positive number which carries value in the first few decimals, rather than the last few. This prevents overflowing or loss of float accuracy to occur. Finally, by taking the mean rather than the sum of the cost, a correction for word length is imposed. For example the sequence 'in' might have the alternative hypothesis 'm'. The correct reading can only be determined through context, so one reading should not be implicitly favored over the other. Suppose all characters have the same hypothesis confidence, namely 0.8. Then without taking the mean the cost would be 0.32 for 'm' and 0.64 for 'in'. With taking the mean, both readings are considered equally plausible until the transition probabilities are taken into account.

To speed up search and end up not having to calculate all paths to find the optimal path, an ordered list implemented as a priority queue was used to ensure that the path with the current lowest mean cost was explored.

Lexical information

In order to determine which alternative readings should be favored given context, the cost C of each added edge is augmented to be the cost of the transition T from the previous character hypothesis

(with a lexicon weight factor λ) added to the previously mentioned cost of the hypothesis confidence *P* itself:

$$\mathrm{C}(h_n|\, h_{n-1}) = -\,\log_2 P + \,\lambda T(h_{n-1}, h_n)$$

$$T(h_{n-1}, h_n) = P(h_n \mid h_{n-1})$$

The Bayesian transition probability is estimated using a language specific dictionary[2], in this case a Latin dictionary. In comparison bigram model based on the labeled data is also constructed and compared.

Experimental setup

In order to get to an optimal working word classifier, some experiments have been set up to do parameter sweeping and method comparison. The naturally first module to optimize is the character classification. All comparisons are done using 10-fold cross validation methods on the classifier.

First the autoencoder is trained on a predetermined amount of characters per class label. If not enough examples are available, additional training data is created randomly using Shepards sinewave distortion. This parameter is called n_per_class. The final reconstruction error on the dataset is reported as a means for comparison. The error E is calculated as the mean error per pixel, i.e.

$$\mathrm{E} = (\,|\,\text{output} - \text{input}\,|\,) \,/\, 7600$$

Initially a sweep is done for a three layered SdA for layer 1 500:500:5000, layer 2 100:100:2000 and layer 3 25:25:1000 using only 100 characters per class. Then we determine an appropriate amount per class to ensure a proper trained Autoencoder.

The classifiers are trained on the original images using 10 fold cross validation. While it is important for the Autoencoder to have about the same sample size per class, we don't mind that label frequency is incorporated into the classifier models.

Then all labeled word images are fed through the pipeline and classified using a fixed lexicon factor $\lambda = 0.1$. Lexicon optimization is treated in the paper of person 2.

Results

Autoencoders

First one sweep over layer sizes yielded optima for two configurations, 1000-500-250 and 500-250-250 with a reconstruction error around 1%.

case	*N_per_class*	*Size Layer 1*	*Size layer 2*	*Size layer 3*	*E*
1	500	1000	500	250	0.0147
2	500	500	250	250	0.0172
3	1000	1000	500	250	0.0162
4	1000	500	250	250	0.0184
5	2000	1000	500	250	0.0220
6	2000	500	250	250	0.0557

[2] http://comp.uark.edu/~mreynold/recint1.htm

Character Classifiers and Word classification

Parametes for the rbf – SVM have been optimized using grid search on the parameters c and gamma. Several combinations have not been tested, for they were shown inferior.

	Character Classification 10-fold				*Word classification*			
case	SVM_{poly}	SVM_{rbf}	QDA	LDA	SVM_{poly}	SVM_{rbf}	QDA	LDA
1	0.9635	0.9877	0.8325	0.8923	0.3208	0.2991	x	x
2	0.8582	0.8813	x	x	0.2650	0.2435		
3	0.9743	0.9822	0.9390	0.9621	**0.3255**	0.2889	x	x
4	0.8991	0.8762	x	x	0.2759	0.2647		
5	0.9493	0.9887	0.9035	0.9302	0.2894	0.2825	0.0536	0.201
6	x	x	x	x				

The confusion matrix for the best word classifier, in our case a SVM with a polynomial kernel and an SdA with a 1000 examples per label, and a [1000 500 250] structure is shown as figure 5. Note how clear the distinction is between lowercase characters, uppercase characters and symbols.

Pure SVM method

A version with only a 3rd degree polynomial had a character classification of 0.9313 but a word classification rate of 0.6090, a big difference to the autoencoder (figure 6).

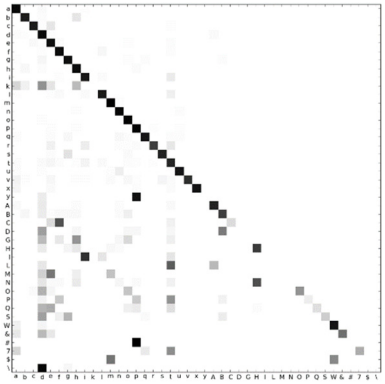

Fig 5: confusion matrix of tenfold cross validation with autoencoder + svm

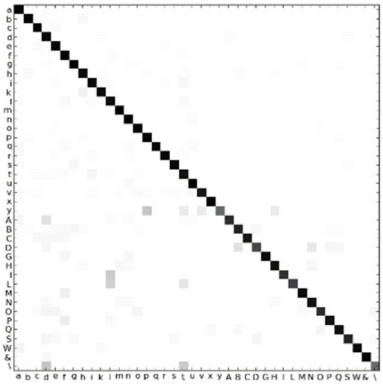

Fig 6: confusion matrix of tenfold cross validation with pure svm

Discussion

A lot of intermediate experiments and parameter sweeps haven't made it into this paper, for ease of the reader, but have been discussed thoroughly throughout the presentation sessions. A few conclusion that we can draw from our methods start with the observation that character classification is relatively easy.

Even though confusion matrices looked as bad as figure 5, compared to figure 6. Character classification results were high due to the not evenly spread data in the testsets. When a classifier gets all the vowels right, the majority of all characters are classified correctly. It was no surprise that nearly all character classification methods performed adequate with an autoencoder character, but the autoencoder proved destructive for word classification.

The biggest flaw in our setup is that the trained classifier gets to see character candidates to classify with a certain confidence score, or to be precise array with a score for each class. When we're dealing with a pure svm, the incorrect stitched character candidates receive low scores in general, with exepction of some 'in' 'm' mixups. But when using the autoencoder in this way, we allowed for

something bad to happen. Because the autoencoder is trying to reconstruct the original character, it is now sensitive to all inputs. In essence, area's in the canvas that don't hold information for characters, such as the corners, are disregarded by the autoencoder, while they are of utmost importance of determining whether it is a clean cut character or some poorly segmented word fragment.

Improvements lie in using multiple voters or mayhaps the incorporation of a classifier that discerns poorly cut characters from true characters. Other than that the algorithm works relatively fast (according to the experimenter) and graph search works relatively well (see person 2). I'm curious whether the method described by person 1 in terms of the acid drop would boost the performance of the word classification, since it provides cleaner cuts.

Team Contribution

The group of four people working together on this project have not all contributed evenly to the final product or during the course. I can honestly say that I've done at least 60% of the work. As a result *this paper is written individually*. Some results are shared with other members, not all.

In summary, I have single handedly written the whole initial pipeline from preprocessing to post processing, with initial word recognition rate around 69% by only using an 3rd degree polynomial kernel svm for classification. Furthermore I was managing the source code and making, preparing and doing the presentations. During the course I implemented the SVM, QDA, LDA and RBF classifiers, the stroke kernel convolution, the character warp module, the final recognizer script, the autoencoder, the trie search algorithm, the trie visualization and numerous test scripts. These included the 10-fold validation scripts for the character classifiers but also scripts to produce confusion matrices and grid-parameter search.

<person 1> has focused on improving the segmenter by implementing an acid droplet technique from scratch, which showed promising results, but was outperformed by the pure svm method during the time of the deadline for the final classifier. In my opinion he did a fair 25% of the work.

<person 2> has worked on a tool to draw characters that could be classified by our system and was mainly responsible for optimizing post processing, including building the bigram models and sweeping the lexicon factor λ parameter. 10% contribution is all that I can give.

<person 3> had initially focused on implementing an Autoencoder from scratch, but failed to deliver for four weeks, on which I took over. He was responsible for the final parameter sweeps, but after three weeks he did not do them, I had to take over again. In the end he didn't contribute in code, but proved helpful during the weekly discussions. No more than 5% contribution.

In conclusion

I had fun during the course, spending hours to finding new methods and implementing them in new ways to try and get the word recognition rate up. I would have liked to produce a more report like final product in which all the acquired data could be presented and discussed. Perhaps once every two weeks. The groups were too big, we could've done (and did) the same with three persons. Nevertheless this was a good course, dealing with AI from start to finish and harnessing all learned matter and put that to good use.

References

Arica, N., & Yarman-Vural, F. T. (2001). An overview of character recognition focused on off-line handwriting. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions On*, *31*(2), 216–233.

Bulacu, M., Van Koert, R., Schomaker, L., Van Der Zant, T., & others. (2007). Layout Analysis of Handwritten Historical Documents for Searching the Archive of the Cabinet of the Dutch Queen. In *ICDAR* (Vol. 7, pp. 357–361).

Crochemore, M., & Vérin, R. (1997). Direct construction of compact directed acyclic word graphs. In *Combinatorial Pattern Matching* (pp. 116–129). Springer.

De Campos, T. E., Babu, B. R., & Varma, M. (2009). Character Recognition in Natural Images. In *VISAPP (2)* (pp. 273–280).

Farrahi Moghaddam, R., & Cheriet, M. (2012). AdOtsu: An adaptive and parameterless generalization of Otsu's method for document image binarization. *Pattern Recognition*, *45*(6), 2419–2431.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., … others. (2011). Scikit-learn: Machine learning in Python. *The Journal of Machine Learning Research*, *12*, 2825–2830.

Shepard, D. (1968). A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM national conference* (pp. 517–524). ACM.

Timár, G., Karacs, K., & Rekeczky, C. (2003). Analogic preprocessing and segmentation algorithms for offline handwriting recognition. *Journal of Circuits, Systems, and Computers*, *12*(06), 783–804.

Vincent, P., Larochelle, H., Bengio, Y., & Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning* (pp. 1096–1103). ACM.