UPPSALA UNIVERSITY
Department of Information Technology

December 31, 2017

Handwriting Recognition

Artificial Intelligence Using Statistical Methods

Fabian Alenius, Kjell Winblad and Chongyang Sun

Abstract

Recognizing handwritten text means transforming a graphical representation of text into its symbolic representation. Handwriting recognition is used in a wide variety of applications. This paper describes and empirically evaluates a handwritten text recognition system based on Hidden Markov Models. The results show that Hidden Markov Models can successfully be used for handwriting recognition.

Contents

1 Introduction

Recognition of handwritten text has been a popular research area for decades because it can be used in many different applications [6]. There are two different approaches to handwriting recognition, *online* and *offline*. In the online approach we know the order in which the strokes and individual points were drawn. This information can easily be captured if the text is recorded by a digital pen or on a touchscreen. In the offline approach we are only given the final image. Online recognition is primarily used for signature verification, author authentication and digital pens. Application areas for offline recognition include postal automation, bank cheque processing and automatic data entry [6]. Formally, handwriting recognition is the task of transforming a language represented in graphical form into its symbolic representation [7].

The ultimate goal in handwritten recognition is to recognize words. However, one way to potentially decompose or simplify the problem is to segment words into its individual characters [3]. Segmentation can either be done *explicitly* or *implicitly*. Explicit segmentation tries to separate the word at character boundaries while implicit segmentation separates the word into equal sized frames. The implicit frames, each represented by a feature vector, are then mapped into characters.

This paper is focused on offline handwritten recognition. We attempt to tackle both character and word recognition. To simplify the word recognition problem, we assume that the images containing the words have already been explicitly segmented into new images containing the separated characters. For word recognition, we also assume that the words come from a finite lexicon. Finally, we assume that the strokes that together compose a character has a width of 1 pixel. These simplifying assumptions were made due to the limited time available for this project. In the following sections we describe how a handwritten text recognition system was developed based on Hidden Markov Models and evaluate its performance.

2 Previous Work

Because handwriting recognition is such a well-researched area there is a wealth of literature available.We mention only a few references that we found helpful. Cheriet et al. [1] gives a good review of the development of handwriting recognition. They also go on to give a broad overview of feature extraction and classification using a plethora of different techniques. Rabiner, L. R. [8] gives an excellent review of HMMs and the Baum-Welch training algorithm, as well as how to apply them in speech recognition. El-Yacoubi et al. [3] introduce an approach to recognize text using Hidden Markov Models with explicit word segmentation. Laan et al. [4] evaluate three different initial model selection techniques for the Baum-Welch algorithm, randomized, uniform and count-based initialization. Despite impressive progress over the last couple of decades, performance is still far away from human performance.

3 Overview of Hidden Markov Models

For pattern recognition problems, like handwritten character recognition, there will always be some randomness and uncertainty from the source. Stochastic modeling deals with these problem efficiently by using probabilistic models [2]. Among such stochastic approaches, Hidden Markov Models have been widely used to model dynamic signals. The Hidden Markov Model treats data as a sequence of observations, while using hidden states that are connected to each other by transition probabilities.

An HMM is characterized by the following [8]:

1. N, the number of states in the model.
2. M, the number of distinct observation symbols.
3. A, the transition probability distribution.
4. B, the observation symbol probability distribution for each state.
5. π, the initial state state distribution.

In contrast to a knowledge-based approach, HMMs use statistical algorithms that can automatically extract knowledge from samples. Also, HMMs model patterns implicitly with different paths in the stochastic work. The performance of the model can be enhanced by adding more samples [2].

4 Method

This section describes the handwritten text recognition system we developed. The section starts by giving a top level overview of the system and then it describes the details such as the HMM implementation, feature extraction and the datasets used for training.

4.1 Overview of Classifiers

The handwriting recognition system has two levels, each containing a classifier. The first classifier is a function that takes an image as input and outputs a character. The second classifier is a function that takes a string of characters as input and outputs a word.

We have implemented two kinds of word classifiers. The first one is called *Forward-classifier* and it has exactly the same architecture as the character classifier and is explained in this section. The second one is called *Viterbi-classifier* and is explained in Section 4.2.2. A flowchart that shows the classification process can be seen in Figure 1.

The classifiers contain HMMs for all elements in the set of possible outputs. So if the character classifier is trained to recognize the 26 Latin characters, it will contain 26 HMMs. When the classifiers are trained they are given input examples for all possible outputs. If the input I is given to one of the classifiers the following steps are performed to calculate the output:

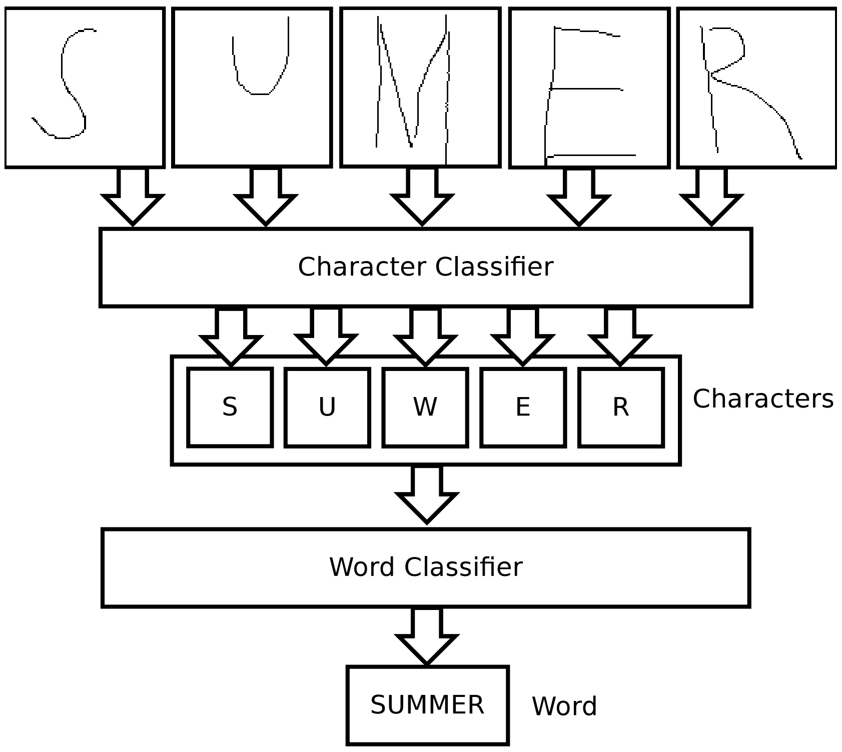


Figure 1: Flowchart of classification process.

1. The probability of I is calculated for all HMMs contained in the classifier:
   (a) I is translated into a sequence of observation symbols $\mathbf{O} = O_1, O_2, ..., O_n$. If I is a string of characters and the output of the classifier is a word, the translation is straightforward. Every character in the string is simply translated to the corresponding observation symbol. There are also special observations for the start and end states. This is explained in more detail in the following sections. If I is an image, the image is first segmented to a sequence of segments. An observation symbol is then obtained from all segments. See section 4.5 for more information about the image feature extraction.
   (b) The Forward algorithm [8] is then used to calculate the probability of $\mathbf{O}$ given the HMM.
2. The output symbol with the highest probability in the previous step is returned as output.

The following parameters must be supplied when a classifier is created[1]:

- The set of possible output symbols and corresponding training examples.
- The initialization method that should be used by the HMMs.
- A binary variable, specifying if the training examples should be used to train the model with the Baum-Welch [8] training algorithm.

[1]A few more parameters can be given but are not listed here because of lack of importance. See the source code of the system for information about other parameters. How the source code can be obtained is explained in appendix A.

4.2 Topology of Hidden Markov Models

Because the Baum-Welch algorithm assumes that the topology of the model is correct, it is important to devise a suitable topology before training starts. The topology of the model is usually constructed by using prior knowledge of the data. For handwritten signals, a left-to-right HMM is often used where no back transitions from right to left are allowed. [9]

4.2.1 Hidden Markov Model Topology for the Character Classifier

A model is created for each character in the training phase, as described in Section 4.1. Because the Latin alphabet only contains 26 characters, it is not too computationally costly to train a separate model for each character. The topology is very similar to the topology suggested by Laan et al [5]. We used special beginning and end states denoted by *start* and *end* respectively. If the image feature extraction step produces n segments there will be $n+2$ states in the HMMs. Special beginning and end states are included because multiple training observation sequences are concatenated to form one observation sequence. See Figure 2. The sequence is then used as input to the Baum-Welch algorithm. The initialization of the transition and emission matrices are done in such a way that the following properties are fulfilled:

- The beginning state *start* will always emit the special symbol @ and the end state will always emit the special symbol $.
- The beginning state *start* will always transition to the first normal state.
- The ending state *end* always transitions back to the beginning state *start*.
- All other states always transition forward to a state that has not been visited since the last visit to *start*.

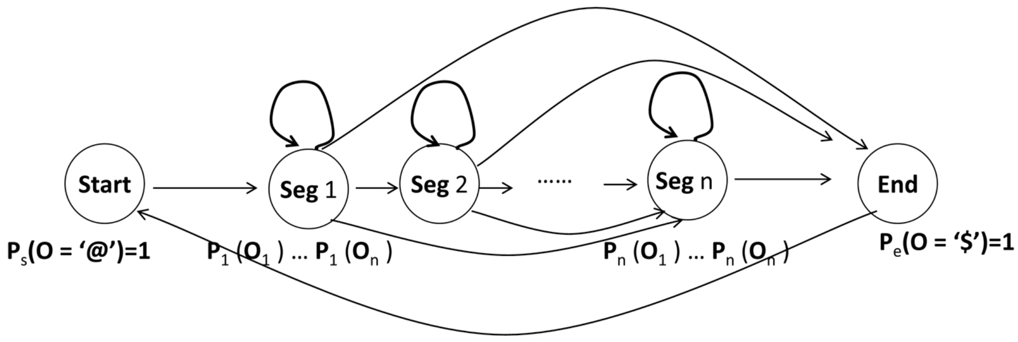


Figure 2: Character topology.

4.2.2 Hidden Markov Model Topology for the Word Classifiers

We have implemented two classifiers for classifying words. The first called *Forward-classifier*, has exactly the same topology as the character classifier and uses one HMM for every character as described in Section 4.1. If the word has n letters, there will be $n + 2$ states in the corresponding HMM.

It is natural to implement one HMM for each of the words when the vocabulary is small. When the vocabulary is larger, this approach may have performance problems. With a large vocabulary using a single HMM can be beneficial. A single model has the additional benefit that it can learn common patterns in words such as ”ing”.

The second word classifier that we implemented, called (*Viterbi-classifier*), uses a single HMM for the whole vocabulary. The HMM used in the *Viterbi-classifier* is called *Word HMM* in the following text. See Figure 3.

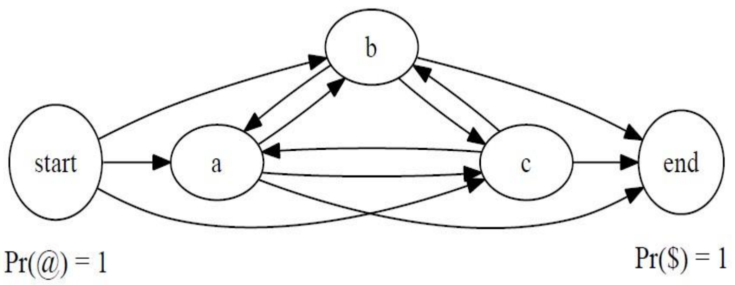


Figure 3: Word HMM topology for a three-letter alphabet.

Unlike the HMMs used in the character classifier, *Word HMM* have states that have transitions to all other states. It has 28 hidden states (26 for the 26 Latin letters and 2 with special emissions @ and $). The transition matrix is a 28 * 28 matrix, which is estimated from the lexicon analysis, using the method explain below. For example, if there are only three words in our vocabulary: DOG, CAT and CAP. Then the probability of going from A to T is 0.5, A to P is 0.5 and A to other letters is 0. According to the parameter estimation method we just explained, the transition probability matrix only needs information from every two successive letters in the words.

The observations are the letters we have observed from the character classifier. The observation probability matrix is created from observations when testing the character classifier. For example, if we have 10 test example images for A, and 5 of them are classified to be A, 3 to be B and 2 to be C, then $P(observation = A) = 0.5$, $P(observation = B) = 0.3$ and $P(observation = C) = 0.2$. For this example, the row for A in the probability matrix will be set to $[0.5, 0.3, 0.2, 0, 0, ..., 0]$.

So given the *Word HMM* described above the classification of a string of characters can be done in the following way:

1. Use the string as an observation sequence and apply the Viterbi-algorithm to get the most probable sequence of states.
2. Calculate the similarity of the resulting string to all possible output words. As similarity measure we use the hamming distance.
3. Return the most similar word.

Based on our topology, useful knowledge from the character classification step is preserved. To demonstrate the performance of the *Viterbi-classifier*, we tested our topology using vocabulary that included eight animal names. See Section 5.3. Among the results from classifying "pig", after using the character classifier, we got "dxq" as observation. *Viterbi-classifier* classifies "dxg" to "pig" instead of "dog". This is attributed to how we estimate the parameters. After observed "d", the path "dog" is more likely to be chosen instead of "pig". However, there is less chance that "o" is observed as "x" (10^{-10}) than "i" is observed as "x" (0.15), so the path is successfully fixed.

4.3 Initial Parameter Selection

Hidden Markov Models can be efficiently trained by the Baum-Welch (BW) algorithm, which is an iterative process for estimating parameters for HMMs. As an iterative algorithm, BW starts from an initial model and estimates transition and emission probability parameters by computing expectations via the Forward-Backward algorithm. The algorithm sums over all paths containing a given event until convergence is reached.

Since the Baum-Welch algorithm is a local iterative method, the resulting HMM and the number of required iterations depend heavily on the initial model. There are many ways to generate an initial model, some techniques consider the training data while others do not [5]. Similar to Laan et al. [5], we tried two initialization strategies, count-based and random. The random initialization strategy assigns the values in the transition and emission matrices to random values. For the count-based initialization the emission matrix is assigned based on information from the training examples. The transition matrix is then assigned so that all states get the same probability of transferring to all reachable states.

4.4 Implementation Issues

During the implementation of the HMM and related algorithms, we ran into some problems. These problems are described in the following sections.

4.4.1 Floating Point Precision

When the number of states $\mathbf{t}$ grows sufficiently large, the forward variable $\mathbf{a}_t(i)$ and backward variable $\mathbf{b}_t(i)$ approaches zero and exceed the precision range of floating point numbers. One way to solve this problem is by incorporating a scaling procedure. For each $\mathbf{t}$, we first compute $\mathbf{a}_t(i)$ according to the induction equation (20) in [8], and then multiply it by a scaling factor $\mathbf{c}_t$, calculated by Equation 1, where N is the number of states.

$$\mathbf{c}_t = \frac{1}{\sum\limits_{i=1}^{N} \mathbf{a}_t(i)} \tag{1}$$

To avoid the underflow problem in the Viterbi algorithm, which calculates the maximum likelihood state sequence, we add log probabilities instead of multiplying probabilities. With this change, no scaling is required.

4.4.2 Zero Probability Transitions

Another problem that may occur when training HMM parameters when little training data is available, is that the probability matrices may contain zero values for valid transitions and emissions. This could lead to bad results if the test set contains a lot of these "impossible" transitions. In the *Word HMM* we initialize all zero probabilities to the value 10^{-10}. For the other HMMs this is left as future work.

4.5 Image Preprocessing and Feature Extraction

The character classifier classifies images that contain handwritten characters, as described in Section 4.1. A sequence of observation symbols is extracted from the supplied training data and used when an image is to be classified. This process is called feature extraction.

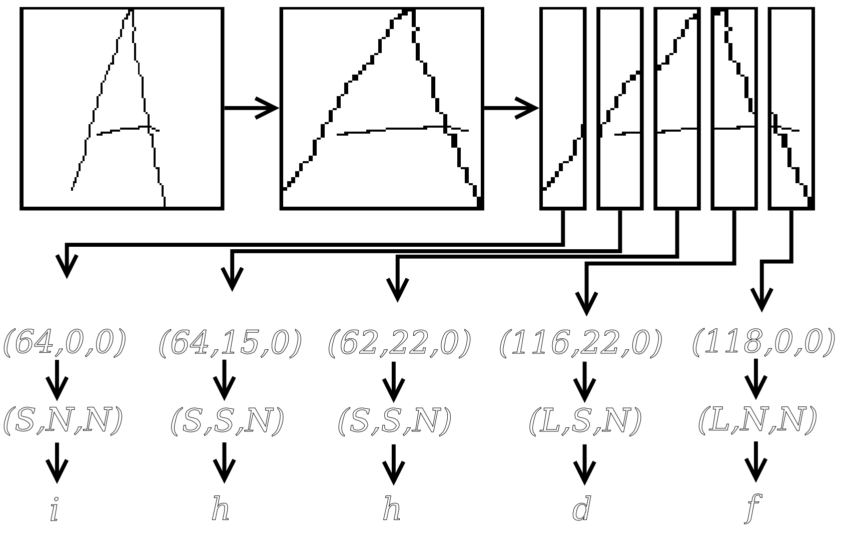


Figure 4: An illustration of the feature extraction process.

As mentioned in Section 1, our system assumes that there is one image per character, that the lines in the characters consists of a single color and that the lines are one pixel wide. The feature extraction process can be divided into the following steps:

1. The scaling step makes sure that the character fills the whole image. Because of this step, it does not matter where in the given image the character is painted. The scaling may cause a problem because the lines may get thicker than one pixel after scaling, if the original painted character just fills a small part of the image[2]. This problem will be avoided if the training examples contain images where the character fills a small part of the image. This is because the model will learn to recognize images with thicker lines. The following algorithm is used to do the scaling:

   (a) The minimum rectangle R in the image that contains the whole character is found.

[2]The standard Java image scaling algorithm is used for the scaling.

   (b) The rectangle R is scaled to fill the size of the original image. The scaled version of R is returned as the scaled image.

2. The new scaled image is sliced into N vertical segments of the same size.

3. An observation symbol is extracted from every segment in the following way:

   (a) The number of pixels in the three largest components in the segment are found. The component sizes are then put into a triple (s_1, s_2, s_3) that is sorted so that the largest number is first. A component is defined as a set of colored pixels that are connected and that contains all pixels that are connected to one of the pixels in the set. Two colored pixels are connected if they are neighbors or if it is possible to create a path of colored connected pixels between them. All pixels except the border pixels have 8 neighbors. If the segment contains less than three components, the triple is filled with zeros.

   So for example if a segment contains two lines. One line containing 10 pixels and another line containing 5 pixels. Then the resulting triple will be $(10, 5, 0)$.

   (b) The elements in the triple is classified as *Large*, *Small* or *None*. The classification function c is defined as in Equation 2. The constant d in the equation is given as a parameter to the feature extraction.

   $$c(s) = None \text{ if } s = 0, Large \text{ if } s > d \text{ and } Small \text{ otherwise.} \tag{2}$$

   The triple (s_1, s_2, s_3) is translated to a triple of classes (c_1, c_2, c_3) by applying the function c to all elements in the triple.

   (c) The triple of classes is mapped to an observation symbol. In total there are 10 different triples and there is one observation symbol per triple. So in total there are 10 different observation symbols.

The classification constant d and the number of segments N are parameters to the feature extractor. An example of feature extraction for an image can be seen in Figure 4.

4.6 Dataset

An attempt was made to find a dataset with handwritten text, but no dataset that fulfilled our requirements was found. The datasets that were found would require a lot of preprocessing. Figure 5 shows a sample from one of the datasets we found. To get good results from that kind of dataset it would be necessary to implement baseline slant normalization, skew correction, skeleton calculation and so on.

Therefore, instead of spending a lot of time preprocessing word images, we implemented a Graphical User Interface to create our own dataset. The largest advantages of this solution is that our solution records one pixel wide lines and the characters are already separated. A large part of the work, image processing, was thus reduced significantly. Our dataset contains 100 examples for every capital letter in the Latin alphabet[3]. An example image from our character image dataset can be found in Figure 4 that shows the feature extraction process.

[3]The dataset is available together with the source code for the system. See appendix A.

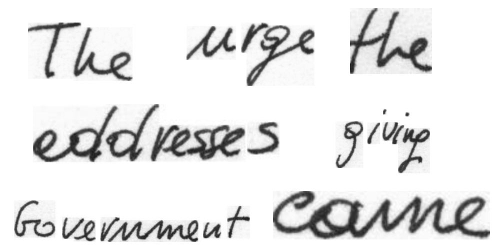


Figure 5: Word image examples.

To get a dataset for training the word classifier a generator was created[4]. The generator creates random errors in the words given as input. To generate the dataset is obviously not optimal for practical applications, but it is good enough to test the implementation.

5 Result

In this section we present the results from evaluating the system described in Section 4. We discuss the results in more detail in Section 6. Finally, in Section 7 we present some future changes to the system that we believe will improve the results.

5.1 Character Classification with Different Parameters

In this section we describe results obtained with the *character classifier*. As described in Section 4.5 the image feature extraction step in the character classifier takes two parameters. The first parameter is the *number of segments* that should be created. The second parameter is the *size classification factor*, which is used in Equation 2. For the experiment we only had 100 examples for each of the 26 characters. How the examples are produced is described in Section 4.6. An initial experiment was performed to test count-based initialization and random initialization before and after training with the Baum-Welch algorithm. The initial experiment shows that there is probably not enough training data for the training to have any positive effect when using count-based initialization. This could possible be fixed to some extent with some kind of smoothening of the model parameters. 10 test examples and 90 training examples for every character were selected randomly from the example set for the experiments. The results from the initial experiment can be found in Table 1. In the initial experiment 1.3 was used as the size classification factor and the number of segments was set to 7.

[4]Please, see HandRecosrcapiword_examples_generator.py in the source code for documentation of the word example generator. See appendix A.

NOE	RIBF	CBIBT	RIAT	CBIAT
90	4%	53%	16%	16%

Table 1: Test of the character classifier with different initialization methods and before and after training. NOE="number of training examples for every word", RIBF="random initialization score before training", CBIBT="count-based initialization score before training", RIAT="random initialization score after training", CBIAT="count-based initialization score after training"

Only count-based initialization is considered in the experiment of different parameters, because the initial experiment showed that the best result seems to be produced when only using count-based initialization and no training. When testing the parameters, 5 models were created in the same way as in the initial experiment. The average accuracy for these 5 models when testing them with their own test example set was recorded as the accuracy for the configuration. For all 5 models that were created, different training example sets and test example sets were randomly selected. We used 90 training examples and 10 test examples for every character, as in the initial experiment. The results of the experiment can be seen in Figure 6.

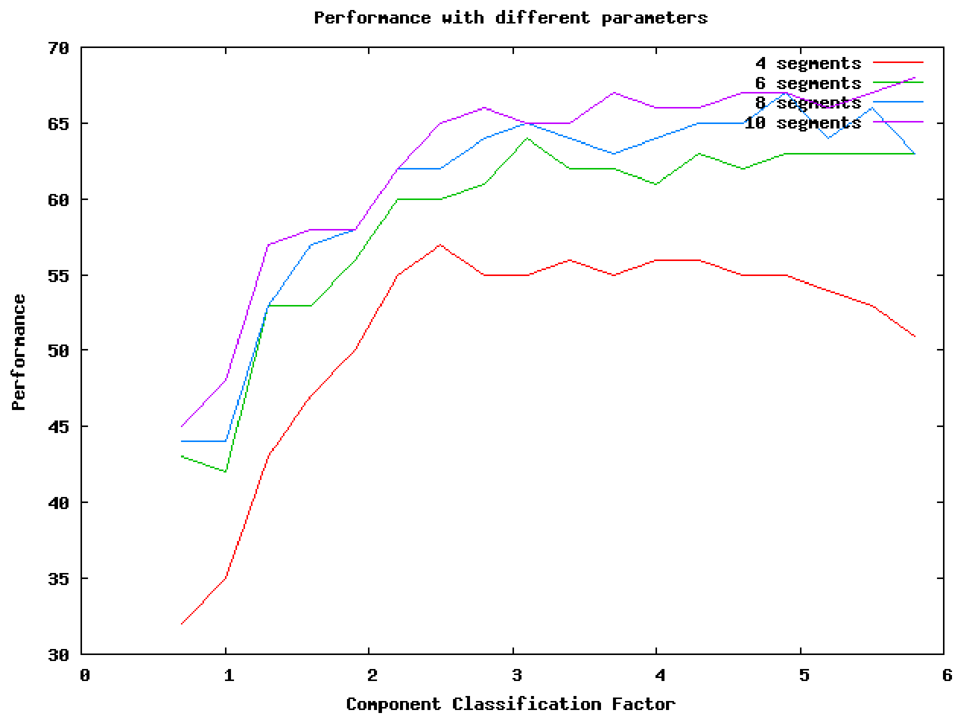


Figure 6: Results for character classification test with different parameters. The performance is the percentage of correctly classified characters.

5.2 Forward-classifier with Different Initialization Methods

In this section we will describe results obtained for the *Forward-classifier*, used to classify words, introduced in Section 4.1. The classification results for running with the words in Table 2 will be presented. The training and test example words were randomly generated with the generator having the properties in Table 3. See Section 4.6, for more information about the word example generator. We used a word example generator because it meant that we could generate as much training data as we needed.

We used a total of 100 test examples to test the accuracy of the created classifiers. Five test examples each for the 20 words. The test examples were generated using the same properties as the training examples. Two initialization methods, count-based initialization and random initialization, were tested with 100, 200, 400, 800 and 1600 training examples. The results of the test is presented in Figure 7. It contains the test scores for the two initialization methods, before and after training with the Baum-Welch algorithm. The test score is defined as the percentage of correctly classified test examples.

dog	cat	pig	love	hate
scala	python	summer	winter	night
daydream	nightmare	animal	happiness	sadness
tennis	feminism	fascism	socialism	capitalism

Table 2: Words supported by the resulting classifier.

Probability of extra letter at position	0.03
Probability of extra letter equal neighbor	0.7
Probability of wrong letter at position	0.1
Probability of letter missing at position	0.03

Table 3: Properties enforced by the word training example generator.

5.3 Two Level Classification with the Viterbi-classifier

In this section we will describe results obtained for the *Viterbi-classifier* together with the character classifier. The classifiers are introduced in Section 4.1. Thus unlike in Section 5.2, we will illustrate the results for a fully working two stage offline-handwritten recognition system.

A test string for a word w used as input to the classifier in the test was generated in the following way:

1. A test example set with examples of character images containing 20 images for every character. For every letter in the string a corresponding character image was chosen randomly.
2. The selected character images were classified with a character classifier which used 100 training examples, 11 segments and a *classification factor* of 4.6.
3. The resulting string from step 2 is the final test example for the word w.

Two experiments were ran with two different sets of possible output words. These sets can be seen in appendix D. *Example set 1* contains 8 words with 3 letters in each. *Example set 2* contains 151 unconstrained English words

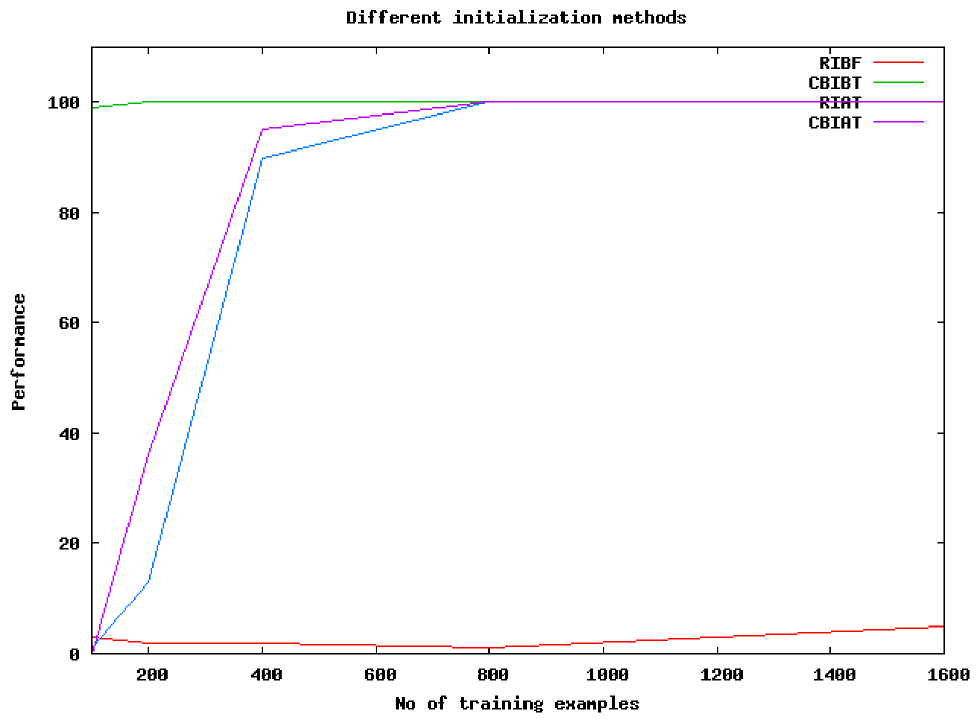


Figure 7: Test with different number of training examples and different initialization methods. NOE=”number of training examples for every word”, RIBF=”random initialization score before training”, CBIBT=”count-based initialization score before training”, RIAT=”random initialization score after training”, CBIAT=”count-based initialization score after training.

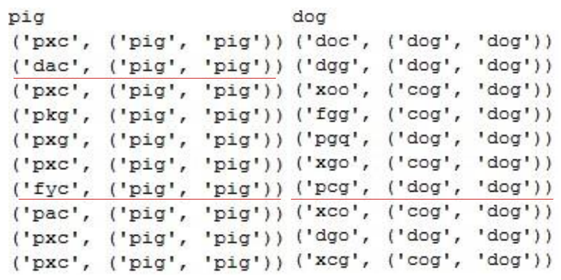
chosen from common prefix and root word examples in the dictionary. To test the difference between using the viterbi correction and simply the similarity measurement, we performed tests both with and without the Viterbi correction. For every word in the list 10 test examples were created. With the Viterbi correction the score was 96% for *example set 1* and 91% for *example set 2*. Without the Viterbi correction the score was 70% for *example set 1* and 91% for *example set 2*.

The results for *example set 2* are almost the same, with and without the Viterbi step respectively. Figure 8 shows how the string looks like before and after the Viterbi step, as well as after the similarity mapping, in this order From the figure it is possible to see that the Viterbi step actually improves the input string.

6 Discussion

6.1 Character Classifier

We believe that the amount of available training data is a liming factor for the character classifier. Because when we train the system, after it has been initialized with the count-based method, with the Baum-Welch algorithm the



```
pig                           dog
('pxc', ('pig', 'pig')) ('doc', ('dog', 'dog'))
('dac', ('pig', 'pig')) ('dgg', ('dog', 'dog'))
('pxc', ('pig', 'pig')) ('xoo', ('cog', 'dog'))
('pkg', ('pig', 'pig')) ('fgg', ('cog', 'dog'))
('pxg', ('pig', 'pig')) ('pgq', ('dog', 'dog'))
('pxc', ('pig', 'pig')) ('xgo', ('cog', 'dog'))
('fyc', ('pig', 'pig')) ('pcg', ('dog', 'dog'))
('pac', ('pig', 'pig')) ('xco', ('cog', 'dog'))
('pxc', ('pig', 'pig')) ('dgo', ('dog', 'dog'))
('pxc', ('pig', 'pig')) ('xcg', ('cog', 'dog'))
```

Figure 8: Example image from output of the Viterbi-classifier experiment.

performance becomes worse. If the classifier is to be accurate for a random person's handwriting, it would be beneficial to let more people paint training examples to get a more generalized classifier.

Our approach will always have problems with characters that look similar to other characters when turned upside down. For example "M" and "W" look exactly alike if they are turned upside down for some handwriting styles. Why this problem occur is obvious if one looks at the feature extraction process.

6.2 Forward-classifier

The results in Section 5.2 clearly show the importance of having enough training data. When using the count-based initialization method, the accuracy actually becomes worse after training the model with Baum-Welch when using less than 800 training examples. The effects of not having enough training data when using the Baum-Welch algorithm is further discussed in [8]. One way to potentially solve this problem is add some kind of smoothening of the probability matrices after training. The smoothening could be done by for example setting all transitions with probability zero to a small value that is greater than zero. The count-based initialization method gives almost perfect accuracy on the test set without training with Baum-Welch.

The vocabulary used by the Forward-classifier is quite small as it only contains 20 words. The accuracy would probably be worse with a larger vocabulary with many words that are similar to each other. The classifier implementation would also have performance problems for many applications with large vocabularies. This is because the time complexity of classifying an example grows linearly with the size of the vocabulary. This is easy to see if one considers that the classifier contains one HMM for every word in the vocabulary and that the forward calculation algorithm needs to run for all HMMs when an example is to be classified. When the training examples are generated as previously described, it is probably not very useful in real applications. For most applications it would be better to use a spell-checking algorithm that can find words similar to a string in an effective way. However, if the training data instead is the result of a handwritten recognition system it could be more useful, because then the model could learn to correct mistakes that the handwritten recognition system does.

6.3 Viterbi-classifier

In general, the Viterbi-classifier performs well. However, when we evaluated with a lexicon that contains long words, the performance improvement is very small compared to just using the output from the character classifier directly. But for the lexicon that contains shorter words, the performance improvement is larger. We believe this is because all the letters are of the same length and therefore the classification becomes harder. With shorter words, a single letter that is wrong can make a big difference.

7 Future Work

While the results show that using HMMs for building a handwritten recognition system is a viable alternative, there is a lot of room for improvement. One way to potentially improve performance is by extending the feature extraction to consider segments from top to bottom as well as from left to right. This means that the observation sequence would become twice as long, given a square image.

During scaling, the lines can become wider than one pixel. Because we are categorizing the strokes based on how many pixels they contain, this is a problem. This could be fixed by adding a thinning phase after the scaling is completed to make the strokes one pixel wide again.

Another way to improve performance is by using a different feature representation. For example, vector quantization can be used to map vectors into a smaller space which can then be used as observations in the HMM. Using this method we would not be reliant on arbitrary constants to map features into observations. It would also make it easier to the extend the system to use additional features.

We saw in our results that adding more training data improves performance so this is an easy way to enhance the system. Finally, to make the system more general we should allow for words that are not pre-segmented into characters.

References

[1] Mohamed Cheriet, Nawwaf Kharma, Cheng-Lin Liu, and Ching Y. Suen. *Character Recognition Systems. A Guide for Students and Practioners.* Wiley-Interscience. Wiley, 2007.

[2] Wongyu Cho, Seong-Whan Lee, and Jin H. Kim. Modeling and recognition of cursive words with hidden Markov models. *Pattern Recognition*, 28(12):1941–1953, December 1995.

[3] A. El-Yacoubi, R. Sabourin, C. Y. Suen, and M. Gilloux. An hmm-based approach for off-line unconstrained handwritten word modeling and recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 21:752–760, August 1999.

[4] Nicholas C. Laan, Danielle F. Pace, and Hagit Shatkay. Initial model selection for the baum-welch algorithm as applied to hmms of dna sequences.

[5] Nicholas C Laan, Danielle F Pace, and Hagit Shatkay. Initial model selection for the Baum-Welch algorithm as applied to HMMs of DNA sequences . *DNA Sequence*.

[6] Umapada Pal, Tetsushi Wakabayashi, and Fumitaka Kimura. Comparative Study of Devnagari Handwritten Character Recognition Using Different Feature and Classifiers. pages 1111–1115, 2009.

[7] Réjean Plamondon and Sargur N. Srihari. On-line and off-line handwriting recognition: A comprehensive survey. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(1):63–84, 2000.

[8] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. 77(2):257–286, 1989.

[9] Ching Y Suen. CHARACTER RECOGNITION SYSTEMS A Guide for Students and Practioners.

A Source Code

The source code for the system described in this report can be found at the following address:

`http://github.com/kjellwinblad/HandReco`

The source code can be downloaded as a ZIP-archive or cloned using git[5].

B Result Reproduction

The test results presented in Section 5 are produced by scripts written in the Jython[6] programming language. The following steps will run the test scripts:

1. Follow the instructions in appendix A to download the source code for the system.
2. Follow the instructions in the `README.md` file in the root of the source code directory. These instructions will help you to set up your environment for running the test scripts.
3. Open a system terminal and execute the following commands (Notice that the path may look different in your system):
   (a) `cd /path/to/HandReco/src/test`
   (b) To run tests for word classifier:
   (c) `jython word_classifer_tester.py`
   (d) To run tests for character classifier:
   (e) `jython character_classifier_tester.py`

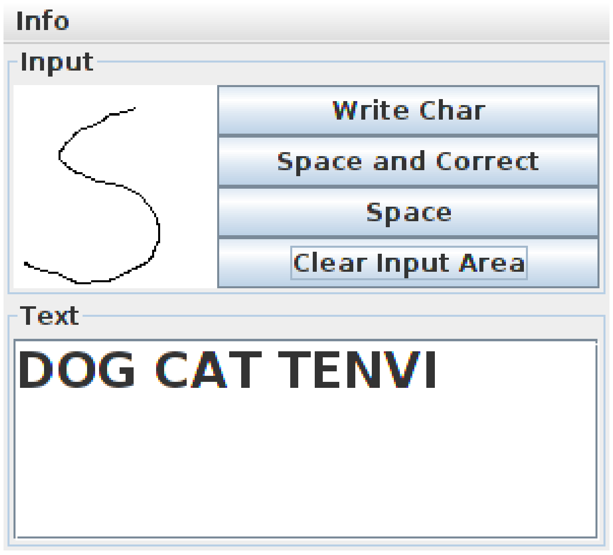
C Testing Handwriting Recognition in Graphical User Interface

A graphical user interface (GUI) has been created in order to test the handwriting recognition system in practice. See Figure 9 for a screenshot of the graphical user interface. The following steps can be used to run the GUI:

1. Follow the instructions in appendix B to step 2.
2. Open a system terminal and execute the following commands (Notice that the path may look different in your system):
   (a) `cd /path/to/HandReco/src/gui}`
   (b) `jython hand_reco_writer.py`

[5]http://git-scm.com/
[6]Jython is a version of Python for the Java Virtual Machine (http://www.jython.org/)

The GUI can only recognize capital Latin letters. To see which words are available for word corrections click on the **Info⇒Available Words...** menu item. To input a character, first paint the character in the paint area and then press the **Write Character** button. To do a space, press the **Space** button. To do a space and let the word classifier correct the last word, press the **Space and Correct** button.



DOG CAT TENVI

Figure 9: Screenshot of HandReco Writer.

D Example Words for Two Stage Classification

Example set 1:

pig,dog,cat,bee,ape,elk,hen,cow

Example set 2:

asexual, amoral, anarchy, anhydrous, anabaptist, anachronism, abnormal, abduct, abductor, abscission, agent, agency, agenda, antipathy, antitank, anticlimax, aquarium, aqueous, automatic, automaton, bisexual, biennial, binary, benefit, benevolent, benefactor, beneficent, biology, biography, circumference, circumlocution, circumstance, democracy, theocrat, technocracy, diagonal, dialectic, dialogue, diagnosis, dynamic, dynamo, dynasty, dynamite, exotic, exterior, extraneous, extemporaneous, exophalmic, exogenous, exothermic, federation, confederate, fraternize, fraternity, fraternal, fratricide, geology, geography, geocentric, geomancy, graphic, graphite, graphology, heterogeneous, heterosexual, heterodox, heterodont, heterocyclic, heterozygous, homogeneous, homogenized, homozygous, identity, idiopathic, individual, idiosyncrasy, idiopathic, incredible, ignoble, inglorious, inhospitable, infinite, infinitesimal, immoral, interact, interstellar, interpret, interstitial, legal, legislature, lexicon, lexicography, liberty, library, liberal, locality, local, circumlocution, mission, transmit, remit, monocle, monopoly, monogamy, monovalent, monomania, monarchy, oligarchy, oligopoly, paternal, paternity, patricide, peripatetic, periscope, perineum, peritoneum, political, metropolitan, premier, preview, premium, prescient, project, projectile, public, republic, pub, publican, psychology, solo, solitary, synchronize, symphony, sympathy, syncretic, syncope, subterfuge, subtle, subaltern, subterranean, telegraph, telephone, teleology, transport, transcend, transmogrify, utility, utilitarian, video, vision, visible