18-551, Spring 2005

Group 9, Final Report

Hand written characters to ASCII text

Isaac Dekine, idekine@andrew.cmu.edu,
Ashley President, apreside@andrew.cmu.edu
Chris Valarezo, cjv@andrew.cmu.edu

Table of Contents

Introduction

Problem

With the increasing move to more mobile technology, the use of touch screens, such as those used for PDA's and Tablet PCs, to make data input easier and more natural, handwriting recognition is becoming increasingly more important. Handwriting recognition using a touch pad or screen has typically been done using shortcuts or simulated handwriting as a communication method. However this method causes users to focus more on learning a new communication method than the message they are trying to communicate and can also be a system that intimidates those less comfortable with new technology.

There are two main ways of recognizing handwriting, offline and online. Offline handwriting recognition involves using a picture of something that has been written and then analyzing the picture to determine what has been written. Online handwriting recognition takes time position data that has been collected from a touch screen with a stylus or other instrument and interprets this data into text. Each of these methods of recognizing handwriting provides it own sets of problems and must be solved in a different way.

Solution

A more effective solution for data input using a touch screen is to use natural handwriting and transfer this message into text. This would help mitigate many of the current problems with commercial online handwriting recognition using pseudo handwriting codes, like graffiti. There are a few commercial products that already implement this solution. Microsoft has a handwriting recognition built into Window XP and Decuma has written a handwriting recognition program that's works with Palm OS PDA's.

In the overall scheme of things, the solution of on-line handwriting recognition requires some steps. The overall system takes an input from a tablet digitizer, in our case a touch screen, and tries to handle the data to decipher text from the data drawn in the tablet. The main steps in the processing can be described under three categories: Preprocessing, Shape Recognition (where the actual shape would be mapped to a character) and Post-processing.

As we will further describe, our project explored some algorithms to do the Shape Recognition part. We did a very small part on the scaling of the input, and hence the preprocessing scheme was not as complete as that of an overall system. At the end we found that modeling characters via Hidden Markov Models was a nice way to attempt a solution using our C67 EVM. In this method, the algorithm would attempt to estimate an empirical pattern of so many states, decided by the design. The Character Recognition would occur when there would be a pattern match.

Prior Work

551 Projects

Two past projects have been done in 551 regarding handwriting recognition. The first project was "Automatic Machine Reading of Text" which was done by Group 8 in spring 2002. This group implemented an offline handwriting recognition program using Fourier Descriptors. The second project "Write on! Character Recognition" was done by group 5 in spring 2003. This group also implemented an offline handwriting recognition program using there own original algorithm and also used a dictionary to help increase accuracy. While all of these projects have a similar goal of handwriting recognition the main difference between this project and these two projects is type of recognition system, online in comparison of to offline.

Other Projects

Other projects in this area have been done at other university by student both undergraduate and graduate and also by a few research groups. The Nijmegen Institute for Cognition and Information at Nijmegen University (http://hwr.nici.kun.nl/) in the Netherlands has a group that had focused in the area of online cursive handwriting recognition for the past ten years. They have several focus areas within this subject, UNIPEN a handwriting recognition database and software, combining the areas of handwriting recognition with that of speech recognition, using the information available in online recognition to expand offline recognition, and addressing these issues of a particular user's writing variability and also the variability within groups of individuals. The Pen Technologies team at the IBM Thomas J Watson Research Center

(http://www.research.ibm.com/electricInk/) focuses on a new dimension in the area of online handwriting recognition, through the use of a special tablet called CrossPad, which resembles a normal notepad but function in the same way as touchpad. There are many other groups in addition to the Nijmegen and IBM Pen Technologies groups that do research in this area Cedar Center at the University of Buffalo and the CENPARMI group at Concordia, Montreal. Each of this group has specific research areas however they both do research in a similar fashion to the research done at Nijmegen.

Algorithm

Different Solutions

There are several different ways to solve the problem of online handwriting recognition. The most common ways are the use of Markov Models, Explicit Methods, and Implicate Methods. Markov Models are a stochastic process which determines the hidden process through the use of observations. This method is explained in more detail in the next section. Explicit methods and models are derived from linear discriminate analysis, principal components analysis and hierarchical clusters analysis. The Li-Yeung method uses geometrical features of the characters and uses the pressure information to creates a recognition scheme. These models are well supported mathematically, however they have two main problems first they rely heavily on assumptions about the form of characters or the statistical distribution of characters and secondly that explicit methods require extensive computations and memory resources. Implicate Methods refers to models that are based on neural networks. The classification of the model behavior is based purely on statistics of an underlying system. A recent development in this area is the Kohonen self organized feature maps (SOFM) and convolution time-delay neural networks, which are similar to the concept of k-means clustering. One benefit of this approach is its invariance to spatial and temporal variations. While each of these methods has its own benefits, none of these methods provides a total solution or perfect accuracy. A new direction in the area online handwriting recognition is to combine the different aspects of each of these solutions into one system, for example the use of Markov Models and neural networks, in the hopes of improving accuracy.

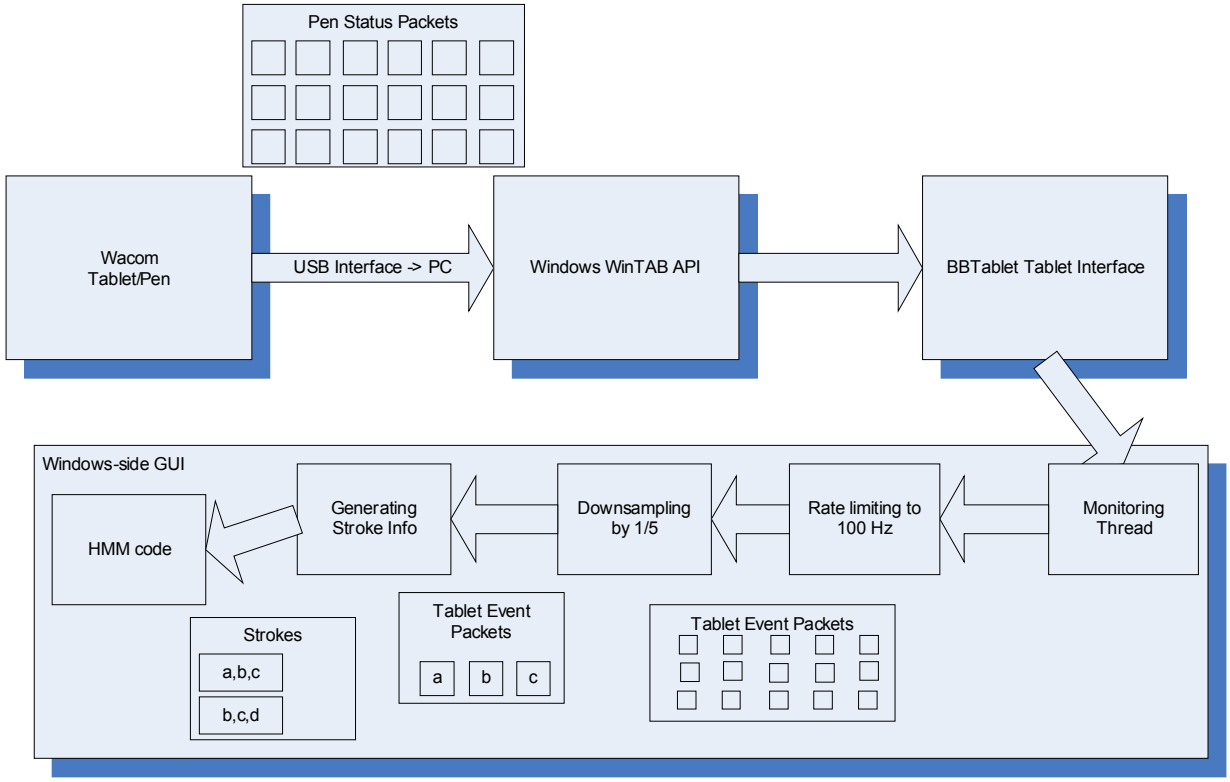
Pre-Processing

Our handwriting recognition algorithm (HMM-based) expects the input data for testing and model generation to be of a certain format. To this end, we generate time-ordered stroke data from the user's input writing. This is not a trivial task to perform correctly, given that our Windows interface receives pen events (events containing point location data, touch pressure, and time) from the Wacom tablet at a theoretical maximum of 100 Hz. It is notable that this is a ***maximum***, for the events are only generated when the pen status changes, thus if the user does not write anything or writes very slowly, the actual update rate will be slower. The method that we picked of processing this point-data into a physical representation was to classify the curves drawn by the user into stroke types, of which 24 exist. These 24 are: for each of the 4 cardinal directions and the intermediate directions (i.e. "N, NE, E, SE, S, SW, W, NW"), a straight line in that direction, a concave-up curve in that overall direction, and a concave-down line in that overall direction. Furthermore, we wanted our pre-processing to create a scale-invariance, and we wanted to eliminate error introduced by jitter or slight variations in writing. As you will see, the methods we chose of performing pre-processing eliminates most of these problems rather effectively.

The first step that occurs in pre-processing is a downsampling of the pen events. This is achieved in two steps: first, a time downsampling, which ignores input events that occur within a smaller time interval than 10ms (i.e. we rate limit to 100 Hz). Second is an event-based downsampling, which further downsamples the number of events which reach the algorithm by a factor of 5. These two downsampling rates are important and necessary, as simply downsampling in time would fail, given that the pen events are not received at a regular rate. Additionally, while the tablet is theoretically sending at a maximum rate of 100 Hz already, this step serves to ensure that the data is actually at 100 Hz, which might not be the case if the tablet driver is "bursty" in receiving data, or if the code is later used with a faster tablet. Maintaining the point data input speed below 100 Hz is important as opposed to simply downsampling the pen events does not help in smoothing out "bursty" input data, and this was an efficient way of dealing with it. The effect of all of this downsampling and rate limiting is to reduce the effects of jitter. As will be shown shortly, the more points that are received, especially nearby points, the more (possibly erroneous) strokes are generated.

Moving along, after the input is downsampled, we store a history buffer of points received from the pen. Once our buffer contains at least 3 points, we can begin processing it. The 3 most recent points in the buffer are analyzed, and determine from it the general direction and concavity, which we henceforth will call a ***stroke***, which we store in a stroke history buffer. When an additional point is added, we generate the next stroke event based on this new point, and the two previous points. This is done until the user stops writing. If the pen is lifted during any part of this procedure, causing the pressure seen in a event to fall below a certain threshold, we generate a pen-up "stroke" (while it is not technically a stroke, it is treated as one for the purposes of HMM estimation and analysis).When the pen returns to the tablet, we clear the point buffer and begin processing points into stroke history again as before. Included in this stroke generation is a simple redundancy filter: when multiple strokes of the same type are generated, sequentially, the code reduces it to one stroke of that type. This is where the entirety of our scale-invariance comes into play. If a person draws a particular character, and then proceeds to draw it again at twice the size, this presumably generates twice the number of sampling points, and likewise would generate twice the number of strokes. However, the redundancy filter eliminates this multiplicity of strokes, and should bring the set of ordered strokes down to its simplest representation. The only bane of this is jitter, in which a jittery hand can generate, in place of one stroke, two similar strokes with perhaps different concavities. This is reduced somewhat by downsampling, though it still ends up being perhaps our biggest problem in recognition, due to the fact that it causes the models to differ somewhat.

Finally, we do not actually store stroke data for any length of time. As soon as we are finished receiving input data, we either send the ordered stroke history to the EVM to perform recognition, or we run Baum Welch HMM model creation code on it to generate an HMM model which we store locally and send to the EVM as training data.



Overview of the signal path of sample pen/tablet data from the hardware level to the HMM level.

Hidden Markov Models

Motivation

Our solution uses a Hidden Markov Model Based Algorithms. The motivation is that handwriting is an empirical process. There is no real set pattern of what happens in a person's mind when they attempt to write a character. For a moment we thought, that by modeling a pattern of stroke movements in a given direction, would sufficiently be able to tell us that this is a correct pattern. From this one would think that we are after modeling a Finite State Machine of stroke transition movements. However, one can't really map these states to a stroke type, so then the model can have fewer states and still be precise enough to describe the writing pattern of a character. From the fact that there is little intuition of what real event the states in the model map to, comes the idea of *hidden* model.

The motivation to use Hidden Markov Models is that it would be helpful in describing states in empirical processes. In fact Hidden Markov Models have been used to predict different processes from speech all the way to war conflicts. We can think handwriting is another empirical process where the written data does not signify a given state in the process, but are rather time observations while a presence in some empirical state.

The Design

The idea is that Handwriting can be modeled as a random process. The states have transitions to other states, each with different probabilities. At each state a physical observation of the process can be made. In our project the observations are actually the strokes recorded as detailed in the previous section. This model looks real close to the working of Markov Chains. In our research we learned that Hidden Markov Models have three main parts

Matrix A: State Transition Probability Matrix, which models the likelihood of switching from some state to another.

Matrix B: Observation Probability Table, which models the likelihood of having an observation when the process is at a given state.

Matrix π: Initial Probability Distribution, which models the likelihood of a state being the first state in the state sequence.

Since the number of states is based on an empirical model, we can randomly select a number of states N to perform our analysis. We chose to use 16 states for all of our models for two reasons. The first reason is the use of a paper by a PhD student at Berkeley. He researched the numbers of states necessary for each character and come up with a best number. The character m had the maximum number of states at 16. The reason whey we did not use the approximated state number for each letter was to help optimize our code to generate static arrays.

Training

In our design we wanted to have a single HMM for each character we wanted to test. We wanted to test all 52 uppercase and lowercase English letter, the 10 digits, and 10 other symbols. Our system would verify against 72 models. We believed that one HMM per character was fine, since there are re-estimation algorithms, that improve the HMM model information as more data is trained with it. The main algorithms that could be used for HMM training and estimation were

Viterbi, Forward-Backward and Baum Welch. Our design finally chose Forward-Backward with Baum Welch training done on the PC side. At the end we had to tweak our training scheme to have nice use of parallelism on the EVM.

The algorithms first analyze the probability of each stroke from all the strokes received, and then the algorithm tries to match this stroke sequence to state transition sequence. As each possible sequence is examined the AND probability is compounded from previous transition to future transition. This way the Transition Matrix A is modified and re-estimated. As this sequence is being evaluated one can see that a stroke observation is being mapped to a state. Basically, the stroke probability can be multiplied to its respective part in the B Matrix. This way the B Matrix is re-estimated. In a similar manner, the initial distribution is re-estimated in the given code.

The training algorithms tweak the probability values of each of the state transitions according to the new possible sequences it gets from the set of collected training data. Theoretically a single Hidden Markov Model that would have been heavily trained would be able to recognize handwriting of characters independent of the users. At this point, this would imply the Hidden Markov Model has accounted for the majority of the ways anyone can write a given character, and hence regardless of the test user, the system should be able to recognize it. Of course, due to constraints of processing time and space on the EVM, having training done on the EVM would be a long task. So we decided to train as much as we can before hand, store the HMM in a file, and initially load this to the EVM. Then, when a new user wants to train, he can re-estimate this already trained HMM.

Recognition

Now, the next simple stage of this algorithm is to identify the actual shape character. The basic algorithm we followed is to use the forward algorithm. The idea is that given an input test sequence, the algorithm would go through the entire state transition matrix A. As it would traverse the matrix, it would compute the probability of a given state sequence that is mapped according to the input sequence. Note that matrix A has mapped out the probabilities for theoretically every single possible sequence that would represent a given character. Therefore, we see that summing all the computed probabilities, would give you an interesting value. The

sum of all the computed probabilities would give you the OR probability of the input sequence being any of the possible input sequences described in the HMM model. Thus, this probability would be a good representation of how likely the input sequence matches the character represented in the HMM.

From our research and readings, we have found that since the complexity of HMM probability computation is very complex, probability values come awfully close. Virtually, on all papers, the probability results are expressed in terms of the logarithm of the probability value mentioned on the previous paragraph.

Overall Recognition and Thresholds

The mentioned algorithm describes basically, the computation of a simple probability. This only answers the question "How is likely is the thing I wrote a letter 'blah'?" Now our design goals were to compare which letter is the one that was written. Basically, we tackled this problem by computing the probability of a sequence against each of the 72 models we tried to provide. Soon, we realized more models would be needed. We would keep this in a basic "for" loop, and always keeping the best score and the character model such scored belonged to. That way we at the end of the loop, the EVM would have found the best model, with the best match.

At the end we realized this was almost a complete solution. But in one of our demos the TAs asked whether this method was able to find or determine when someone is just doodling rather than attempting to write a character. From running multiple experiments of the real characters and attempting to recognize them, we found the recognition log probabilities go as far as -27. Therefore, we found it appropriate to have a threshold of -30. This means if the best log probability is below -30, then we believe it is highly likely that the input data is just doodling. This way we implement another feature to our system.

Implementation

Data Flow

Our implementation of computer-based handwriting recognition was not quite straightforward. The project was organized as follows. On the PC resides a GUI which has numerous modes, including a training mode, a recognition mode, and a standard user-interface mode. The standard user-interface mode is used for typing text (to associate with training data) and clicking various buttons. The training and recognition modes are very similar in that they both monitor the tablet for input data; they differ in where the data is sent after the mode is completed. The training mode generates an HMM model of the input data after being pre-processed (see ***pre-processing*** section), and sends it to the EVM using a PCI transfer. The recognition mode sends the stroke data (not converted to HMM model) to the EVM using a PCI transfer. The reason for the difference in transmission format is that we wanted as much data to be stored as HMM models when possible – thus the training data gets run through the Baum-Welch algorithm for forward-backward HMM estimation/re-estimation on the PC-side to create an HMM model before sending it to the EVM. However, for recognition, we run the input strokes through each HMM model using the Forward algorithm only. This requires having the ordered list of strokes, not an HMM model. Thus this is the one place where stroke data, not an HMM model, is transmitted to the EVM.

Looking more in depth at recognition, the following steps are executed. Upon receiving the ordered stroke data, we loop over all characters and perform the following steps:

1) Page all models for that character from SDRAM into local heap memory (for speed)
2) For each model, run the strokes through the forward algorithm using that model and calculate the log probability of the strokes matching that model.
3) This letter's score is the largest value seen in step 2 over all models for the letter.

After analyzing all of the letters, the one with the largest score (from step 3 above) is declared the winner, if its log probability exceeds the threshold (which we have set at -30).

Memory usage – HMM models

In more detail, the HMM models are handled as follows: on the PC, at load time, the GUI attempts to load from disk 5 models for each character (there are 72 defined characters). These files are stored as plaintext matrices, named based on the character that it represents and the model # of the particular model. It sends as many as it could load, and dummy models (basically uniform transition probability matrices) for those that were not found, to the EVM via a PCI transfer. Each HMM model contains an A matrix, which is 16x16, a B matrix, which is 16x25, and a pi matrix, which is 1x16. The purpose of these matrices is explained in depth in the algorithm section of the paper. This translates to 672 values, stored as 8-byte doubles, per model. There are 72 characters, each of which have 5 models associated with it, thus the transfer size (and consequently, the SDRAM memory footprint of this data on the EVM) is approximately 1.9 MB. Thus it should be obvious as to why it is stored on SDRAM – it definitely wouldn't fit on chip. When the EVM code uses these matrices, generally for recognition, it pages in from SDRAM as needed, performing recognition on a set of data already paged in, while paging in the set to be worked on next. For ease of use, the PC side models are maintained in memory while the program is running, and updated if the user performs training. When the training is performed on a character, one of the 5 models that are maintained for that character are overwritten, in all 3 places:

- on disk
- on PC memory
- on EVM memory (it arrives via a PCI transfer)

The PC and EVM both eliminate data starting from model #1, and as each model is overwritten (assuming the user keeps training for that character), the model number that is overwritten is incremented and wrapped around. Thus in general, the model that is least recently created will be overwritten. However, this information about which model was written least recently is only maintained until the PC GUI exits.

Communication

Finally, a look at the PC⇔EVM communications. The PC and EVM operate in what can be viewed on as a master/slave communication configuration. The EVM essentially runs as the master, sending commands to the PC. The first command sent is a

SEND_ALL_TRAINING_DATA command, which tells the PC that the EVM just started and needs all of the HMM data to load onto memory, for later use in recognition. The PC acknowledges by sending all of the training data. After that, the EVM sends the PC a REQUEST_COMMAND message, which tells the PC that it is done, and is ready to receive commands from the PC. This is where the GUI and the user comes in. When it receives the REQUEST_COMMAND message, the GUI alerts the user to the EVM's readiness by updating the small "Rockstar" logo on the left side to green. At this point, the user can enter a training or a recognition mode. When this occurs, the PC sends to the EVM a command, and depending on the command, the EVM then sends a command to the PC requesting the associated data for the mode (either the new HMM training model, or stroke information). While the EVM technically is the master by controlling the mode in which both operate, the EVM generally is sending the PC the REQUEST_COMMAND command, which yields control to the PC. Whenever the PC has not received this packet, the "Rockstar" logo on the GUI is purple, to indicate that the EVM is not ready for a new command.

Lab Demo

GUI

The GUI was designed for simplicity and ease-of-use. There are four modes in which it operates: user-interface, free-writing, training, recognition. The modes are relatively self-explanatory, but we will include a brief explanation nonetheless:

- User-interface: All buttons and controls are enabled. The user can enter a character to train for, or select a button (training, recognition, or quit) to enter another mode. Pressing escape enters free-writing mode.
- Free-writing mode: The buttons and controls are disabled. The GUI monitors the tablet device for point data, and when it is received, draws it on the screen, connecting the discretized dots with line segments. Pressing escape returns to user-interface mode.
- Training mode: After entering a character into the text box and selecting "Train", the user's input is monitored much like in Free-writing mode. However, when the user

presses escape, the stroke data is converted to HMM model format and send to the EVM, and also stored locally on the PC (to disk).

- Recognition mode: After selecting "Recognize", the user's input is monitored much like in Free-writing mode. However, when the user presses escape, the stroke data is sent to the EVM and run through the recognition algorithm on the EVM. The resulting letter is displayed in the text box in the GUI.

When the EVM is ready for commands (the EVM⇔PC command structure details are outlined further in the ***Implementation*** section), the "Rockstar" (R*) logo turns green. When the EVM is processing or transferring data to/from the PC and not yet ready, the logo turns purple.

Demonstration

The lab demonstration consisted of one of the individual showing our recognition system recognize characters that had been trained using the system. The demo also had an untrained individuals try to recognize characters on the system.

Analysis and Conclusions

Algorithm

The algorithms described in the previous section were implemented as best as possible as described in the Implementation Section. Yet as we made experiments, we found that the training of HMM did not behave as expected. Once the HMM was trained with a recent sample, the HMM would discriminately favor the last sample trained. So we found that the HMM re-estimation works, but does not show a true training. Basically our experiments showed, that the log probability of previously sampled data would fall tremendously (below -30), when new data was trained on the HMM. There was no true sense that a single HMM would account for all possible ways a character would be written anymore. We attribute this failure to perhaps a lack of training data. We know the data sets considered for HMM recognition is huge, and perhaps more data was needed to stabilize the Matrix values in A, B and π.

Therefore, we went with the idea of having for each character a single HMM for a way a character can be written. We realized this set was infinite, and so we went ahead and tried 5 training samples for each character. Each of those samples would create a discriminate model to a different way the character was written. This way we attempt to cover different ways the character can be written. This way, we went from having 72 training HMM models on the EVM, to having 72*5=360 models. This made the problem more interesting, because we believed the computation would be way slow for the par of the project and its goals.

Here we realized we should do Training on the PC side. This was done because besides having a lot of code that would risk size constraints on the EVM, now we were faced with having to do 360 training calls when training stages would occur.

Another interesting approach we used is to have input paging using both of the DMA burst channels. We realized that a character set of HMM models (five models per character) would be about 25 Kb in size, and therefore we can fit 2 characters in the ON-Chip memory. With paging, we can be doing recognition analysis on one character, while loading up the data for recognition analysis on the next character in the other DMA channel. Theoretically this paging scheme would reduce our run-time by a factor 2.

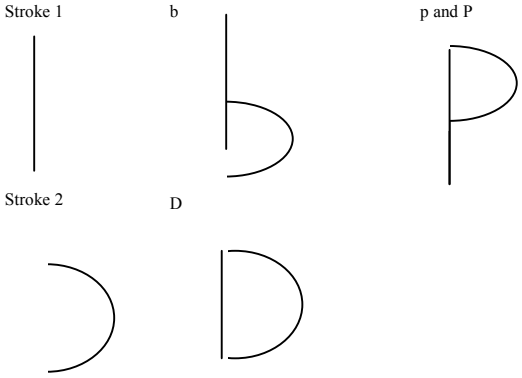
These two changes in the algorithm helped translate the project from pure PC code, to something that can robustly run on the EVM.

Results

We tested our system using three individuals attempted to write each character three times. One of the individuals, User A, trained the Markov models that were used, while the other two individuals, User B and User C, used models that the first person trained.

The data was classified in two different ways. The first was an exact match. This happens when the system matches the exact character that was written. The second type of match is a similar match. This occurs when the system matches to either the exact character that was written or a character with a similar writing pattern. For example the letter b, D, p, and P all have a similar

stroke pattern. As the figure below shows, the first stroke is a downward line and the second stroke is a curve. This pattern can be combined into several different letters.



This was also used for characters where the upper and lowercase version of the letter stroke pattern is similar but size is different. The HMM Model pattern does not distinguish between the size of letters. Also some characters were grouped into similar match groups because the system has a hard time distinguishing when an additional line is added into the system. For example the difference between the character T and the character I is an additional line. The system will sometimes match these into the same group based on a single line. Bellow is a grouping of characters that were put into a similar match group. Some characters appear in multiple groups because they have a pattern that is similar to features in both groups.

- a, g, q, 9
- b, B, p, D, P
- B, k, K
- d, k, K
- E, F, I, t, T, +
- G, 6
- h, m, M, n, N
- j, J
- l,1, L
- o, O, Q, @, 0
- s, S, 8, $
- u, U, v, V, w, W
- x, X
- z, Z

The table below shows what the match percentage was for each user in each match category, exact and similar. The table separates the characters into lowercase, uppercase, numbers, others and then a total for all characters.

	Exact Match			Similar Match		
	User A	User B	User C	User A	User B	User C
Lowercase	42.3%	16.7%	15.4%	94.9%	51.3%	34.6%
Uppercase	55.1%	29.5%	25.6%	94.9%	71.8%	41.0%
Numbers	83.3%	26.7%	10.0%	96.7%	50.0%	23.3%
Other	93.3%	50.0%	43.3%	96.7%	66.7%	46.7%
Total	59.7%	27.3%	22.2%	95.4%	60.6%	37.0%

The table shows that the recognition percentage for User A is higher in all categories, including recognition for exact and similar matches. Another key thing to note is that the recognition in the Other category is much higher. This can be explained because the characters in this group are more unique than characters in the other groups. The recognition percentage also increases drastically when the Similar Match scheme is used. They HMM system works and correctly identifies the different patterns created by a character and then matching that pattern to an appropriate model. The models do not however distinguish between characters with a similar stroke pattern. This is fault of the HMM recognition system and could be improved by adding in position and size data.

Future Work

Future work that could be done in the area of online handwriting recognition for future projects in this course would be implement an online recognition program that used one of the other two methods, explicate or implicate. Another expansion in this focus would be to improve recognition aspect of the program using a dictionary or vocabulary system or to add weight different characters differently based on their expected use, for example the character e would have more weight than the character z, sense the character e is used more often in the English language. An interesting expansion to this project would be to combine the Hidden Markov Model with the use of position on the tablet. This would help distinguish between the characters like D, b, P, and p.

References

Papers References

L.R. Rabiner, "An Introduction to Hidden Markov Models", in IEEE ASSP Magazine, pp. 4-16, January 1986.

This paper provides a general introduction to the concept of HMM and also explains in general terms how HMM can be used in Speech Processing. This document was found using IEEExplore.

Rabiner, L. R. (1989). *A tutorial on hidden Markov models and selected applications in speech recognition*. Proc. of IEEE, 77, 257--286.

This paper proves a more details introduction to HMM than the "An Introduction to Hidden Markov Models." This paper also provides a detail description of how HMM can be used to incorporate into a Speech Processing.

Shankar Narayanaswamy. *Pen and Speech Recognition in the User Interface for Mobile Multimedia Terminals*. Ph.d. thesis, University of California at Berkeley, 1996.

This paper showed an implementation of using HMM for online handwriting recognition. It was usefully even though he showed no code, because it gave us incite into the number of states N that were necessary. It allowed us to minimize the amount of time spent using trial and error trying to determine the level of N.

Code References

Wintab

http://www.pointing.com/WINTAB.HTM

This is a Windows API for interfacing with various tablet input devices for PCs. It provides a wide range of functionality, much of which was unnecessary for our project, however would be useful depending on the features required of the tablet. To be useful, we combined it with the BBtablet software below.

BBtablet

http://www.billbaxter.com/projects/bbtablet/

BBTablet is an interface to the Wintab API, providing a simpler interface and an message-based interface to the tablet device. The code makes it very easy to receive tablet information (such as pen location and pressure).

HTK

http://htk.eng.cam.ac.uk/

This site provides detailed code that can be used for Speech Recognition. The HTK system provides a detailed Speech Processing Toolkit along with a Hidden Markov Model Toolkit. While this system could be useful for a Speech Recognition project, the basic HMM function are embedded within speech processing toolkit.

UMDHMM-v1.02

http://www.cfar.umd.edu/~kanungo/software/software.html

University of Maryland software that provides basic HMM functions in c. This code was used for our final implementation. This code however is very basic and does not provide a good re-estimation algorithm for models.