Tad the Therapist: A Virtual Assistant that Recognizes Human Speech using Hidden Markov Models

Erik Hodges, *Student, UCLA*, and Ivan Manan *Student, UCLA*

***Abstract*—We built a therapy bot that recognizes single words and provides a response via a web user interface. The various technical challenges of this project allowed both of us to exercise our strengths as well as learn about areas that we are unfamiliar with. We used Hidden Markov Models (HMMs) and Mel-Frequency Cepstrum Coefficients (MFCCs) for speech recognition. This technique provides excellent results for words that the program has been trained to recognize but is unable to recognize other words.**

***Index Terms*—Forward Algorithm, Hidden Markov Model, K-means Algorithm, Mel-Frequency Cepstrum Coefficients, Speech Recognition**

I. Introduction

THIS section covers the history, sources, and global constraints.

A. History

One of the earliest recorded instances of automatic speech recognition was done at Bell Labs in 1952, when Three Bell Labs researchers, Stephen Balashek, R. Biddulph, and K. H. Davis built "Audrey" to understand speech recognition with digits only. Their implementation involved locating the formants in the power spectrum of the spoken digit. A formant is the spectral shaping that results from an acoustic resonance of the human vocal tract [1].

In the 1980s, automatic speech recognition was shifting to the use of hidden Markov models. Hidden Markov models proved to be more efficient than locating formants because they are a statistical method that does not rely on templates or spectral distance measures [1].

However, during the past decade, there has been a breakthrough in technology due to the successes of deep learning. Hence, most modern speech recognition systems utilize some form of recurrent neural networks to identify full sentences [1].

Automatic speech recognition is currently ubiquitous in society. Consider the popularity of Siri, Alexa, and Google Assistant and their integration in homes and smartphones. With technology developing at a rapid pace and breakthroughs emerging in machine learning, automatic speech recognition can only become more accurate and faster than ever before.

While Hidden Markov Models are no longer the state of the art, for single words and small vocabularies, they are more computationally efficient than recurrent neural networks due to their simplicity. Additionally, while recurrent neural networks take a long time to train using GPUs, Hidden Markov Models can be quickly trained without the need for high computing power.

B. Global Constraints

The speed of the JTAG emulator used by the LCDK causes high latency when the LCDK writes or reads text files on the computer. This prevented us from having the LCDK transfer MFCCs to the computer in real time. Instead, we transfer the entire batch of MFCCs for one recording at the same time.

II. Motivation

Both of us come from very different backgrounds. Ivan's background is in software development and Erik's background is in theory and hardware. We wanted to choose a project that would allow both of us to exercise our strengths. Our project had a lot of challenging software development related to the web interface, chatbot, and communication between the computer and LCDK. We chose Hidden Markov Models as our speech recognition technique because, unlike neural networks, they are practical to implement and it is possible to design a system that works using probability theory. These two aspects of the project allowed both of us to exercise our strengths as engineers and create something much more impressive than what either of us could have created on our own, while also learning about what the other person was working on along the way.

The project result is rather crude as a therapy bot and would

This paper was submitted on March 23, 2019. This project was supported by the Electrical and Computer Engineering Department at the University of California, Los Angeles. This project was completed as part of the Digital Signal Processing Capstone Design course led by Dr. Dennis Briggs.

The authors are with the Department of Electrical and Computer Engineering at the University of California, Los Angeles, Los Angeles, CA 90095 USA. (e-mail: eehodges18@g.ucla.edu; ivanmanan@ucla.edu)

definitely not pass the Touring test but the basic speech recognition functionality could be useful in an “umm” detector, which is used to identify and delete instances of a podcast host or other speaker saying “umm” between phrases. Our method would work well for this because it runs very fast and could scan an entire audio file more quickly than a complicated technique using neural networks.

III. Approach

This section covers the team organization, plan, standards, theory, software, hardware, system build and system operation.

A. Team Organization

Erik’s responsibilities were to design and implement the Hidden Markov Model infrastructure and algorithms in C++, implement the training algorithm, and define the training protocol. Ivan’s responsibilities were to design and implement the communication protocols between the TI L138/C6748 Development Kit (LCDK) and the computer, implement the computer program infrastructure, build the web server, and design the chatbot. The MFCC extraction code was mostly recycled from fall quarter’s mini project with some adjustments, so both of us had relatively equal contribution to that.

B. Plan

The plan was to create a virtual assistant that recognizes human speech and provide counseling by voicing an appropriate response. The first approach was to create a recurrent neural network that would follow a similar architecture to Mozilla’s open source DeepSpeech. The recurrent neural network would take speech spectrograms and generate English text transcriptions. The reasoning is that modern speech recognition systems use recurrent neural networks and provide less time complexity than hidden Markov models. However, developing an entire recurrent neural network and obtaining speech spectrograms within a couple of months was deemed infeasible due to resource consumption and complexity of such implementation. Since the entire system is scaled to only identify single words, using hidden Markov models was a more appropriate alternative to use for speech recognition.

Training data was gathered by speaking the same word into the LCDK multiple times. The speech signals were processed by converting time-domain audio signals into Mel-frequency cepstrum coefficients (MFCCs). The K-means clustering algorithm was used to find clusters of similar MFCC vectors in the data set for a single word. Each cluster corresponds to a state and we can calculate the state transition probabilities based on cluster size. These states were then used to build the hidden Markov model for an individual word. The hidden Markov models (HMMs) are implemented using C++ object-oriented programming.

Given the MFCCs for a single word, this data set is inputted into the HMMs for every known word in the vocabulary. Every HMM produces a probability for a recognized word, and the HMM with the highest probability outputs the identified word. The identified word is then passed into a chatbot program. The chatbot maintains a probability distribution for predetermined responses to user inputs. Afterwards, the computer program exports the recognized word and the chatbot response to the web server. The web server provides the user interface for the client to view the entire conversation.

C. Standard

To perform feature extraction on recorded audio signals, we used a technique called Mel-Frequency Cepstrum Coefficients (MFCCs).

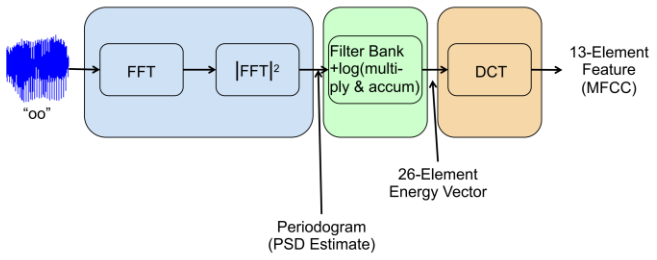

“oo”
FFT
|FFT|²
Filter Bank +log(multi-ply & accum)
DCT
13-Element Feature (MFCC)
26-Element Energy Vector
Periodogram (PSD Estimate)

Figure 3.1: MFCC Block Diagram
Courtesy of Dr. Dennis Briggs, UCLA

The first step of this technique involves taking a time domain sequence of inputs, applying a Hamming window, and taking the square of the magnitude of the Fourier transform to get an estimate of the power spectral density of the signal.

Next, the PSD is fed into a bank of 26 filters. The widths of each filter are logarithmically distributed based on the Mel-frequency scale. The conversion between real frequency and Mel-frequency is shown in Figure 3.2. By creating uniformly distributed figures in the Mel-frequency domain and converting back to real frequency, we obtain the logarithmically distributed filter bank. The purpose of this conversion is to mimic how humans perceive sound. For example, human hearing is much less sensitive to small changes in frequency at high frequency than at low frequency. For example, the frequency difference between adjacent musical notes increases logarithmically with frequency but the difference in pitch between adjacent notes sounds identical to a human regardless of whether the adjacent notes are low or high.

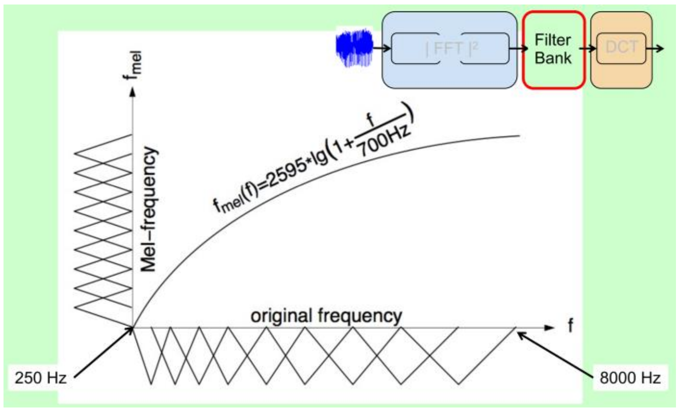


Figure 3.2: Mel-Frequency Scale
Courtesy of Dr. Dennis Briggs, UCLA

Next, we define a 26-long vector X_m using the convention that m = 1 corresponds to the lowest frequency filter

$$X_m = \log_{10}(total\ output\ energy\ of\ filter\ m)$$

Finally, we take the discrete cosine transform of X_m to obtain a 13-long vector of MFCCs

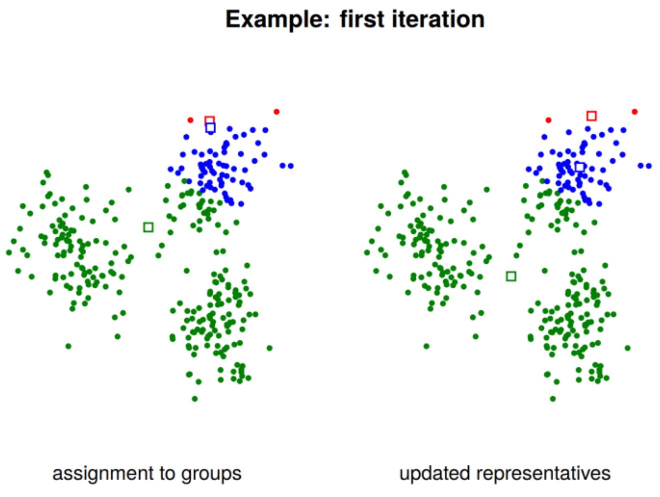
$$MFCC[i] = \sum_{m=1}^{26} X_m \cos\left(i\left(m-\frac{1}{2}\right)\frac{\pi}{26}\right), \quad i = 1,2,\dots 13$$

We used the K-means clustering algorithm in our training method. The algorithm finds K clusters of data points in a data set. The pseudocode for the algorithm is as follows:

```
Initialization:
  define k centroids by randomly picking data points

Iteration:
  assign each data point to its closest centroid
  move each centroid to the average of all points in its
  cluster
  if any centroids moved
    repeat the iteration step
```

Figure 3.3 gives a visualization of how the K-means algorithm works.



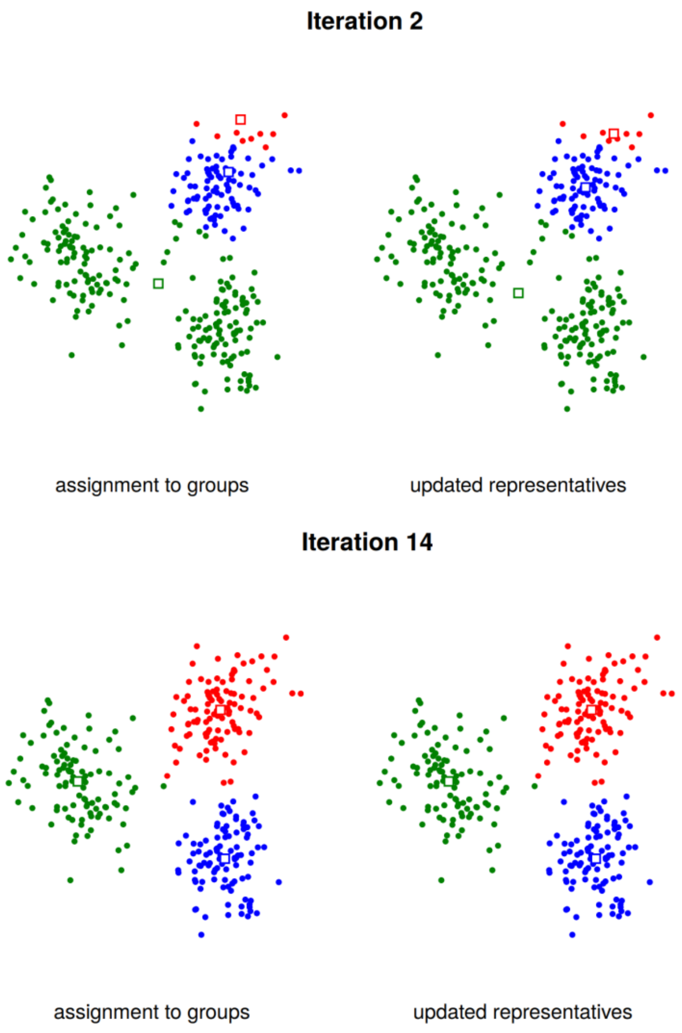


Figure 3.3: K-Means Algorithm Visualization
Courtesy of Dr. Lieven Vandenberghe, UCLA
https://tinyurl.com/vandenberghekmeans

D. Theory

The first step to understanding a Hidden Markov Model (HMM) is to understand a Markov process. A Markov process is essentially a finite state machine in which the state transitions are probabilistic. Figure 3.4 shows a Markov process with two states: 0 and 1. Each state has a probability of transitioning to each state in the system on each step. For example, in Figure 3.4, when the system is in State 0, it has a 40% chance of remaining in State 0 and a 60% chance of transitioning to State 1 on the next step.

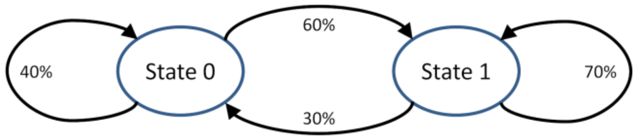


Figure 3.4: Markov Process

The transition probabilities for a Markov process are often expressed as an n × n transition probability matrix A, where n is the number of states in the system. The matrix is formatted as follows:

$$a_{ij} = P(\text{transition from State i to State j})$$

For the system in Figure 3.4, the transition probability matrix would be:

$$A = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} = \begin{bmatrix} 0.4 & 0.6 \\ 0.3 & 0.7 \end{bmatrix}$$

In many cases, the states of a Markov process have probabilistically distributed outputs. Figure 3.5 shows a modification of the system from Figure 3.4 that has this property.

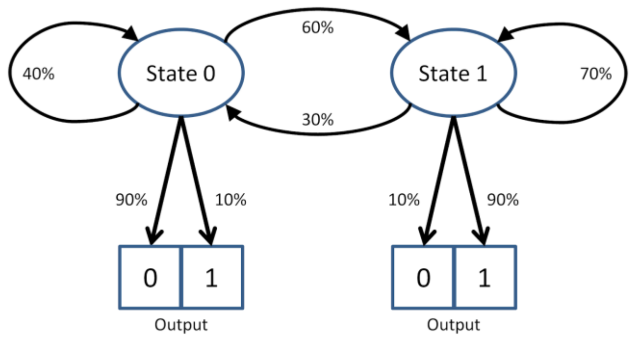


Figure 3.5: Markov Process with Probabilistic Outputs

An example of a real application of this particular process is a communication link over a channel with errors. The current state corresponds to the bit that the transmitter sends over the channel and the output corresponds to the bit recorded by the receiver. In this case, the channel would have an error rate of 10% for both 1s and 0s.

All that we need to do to transform the system in Figure 3.5 into a Hidden Markov Model is to make the state of the system invisible to an observer, i.e. the only thing that an observer sees is a sequence of outputs.

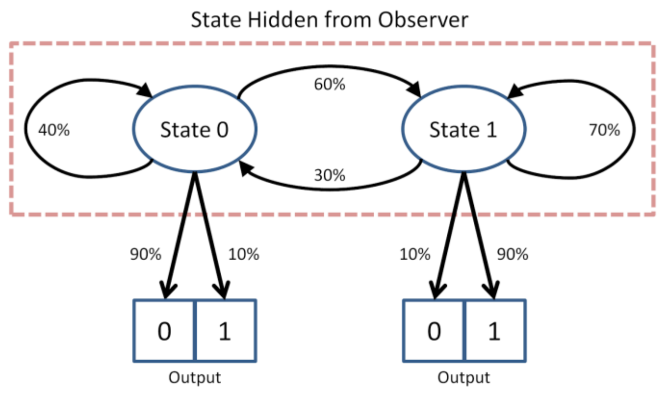


Figure 3.6: Hidden Markov Model

Note that the HMM now perfectly describes the situation of the receiver in a communication link with errors since the receiver never knows for certain what bits the transmitter sends. This is one of countless application for HMMs.

One of the most important uses of an HMM is to compute the probability of the model generating an arbitrary sequence of outputs. Before we describe this process, we must define some parameters. We say that the system has N states, $S_1, S_2, \ldots S_N$. We have already defined the transition probability matrix A. We denote the M possible outputs as $V_1, V_2, \ldots V_M$. The probability of a state producing each outputs is denoted as b. The probability distribution of the starting state is denoted as π. Finally, we will refer to the state of the system at time t as q_t. In summary:

$$a_{ij} = \mathrm{P}[q_{t+1} = S_j \mid q_t = S_i], \qquad 1 \le t \le T$$
$$b_j(V_k) = P[V_k \; at \; t \mid q_t = S_j], \qquad 1 \le t \le T$$
$$\pi_i = P[q_1 = S_i]$$

If we are given a sequence of outputs $O = O_1, O_2, \ldots O_T$ and we somehow know the sequence of states $q = q_1, q_2, \ldots q_T$, calculating the probability of the model generating O as its output sequence is done by multiplying the probabilities of the state at time t producing the output at time t for all t.

$$\mathrm{P}[O \mid q] = b_{q1}(O_1) b_{q2}(O_2) \ldots b_{qT}(O_T)$$

Using the total probability theorem, we can find the probability of the model producing O:

$$P[O] = \sum_{\substack{q = \, all \; possible \; state \\ sequences}} P[O \mid q] \, P[q]$$

While this approach works, for a output sequence of length T and a model with N states, there are N^T possible state sequences, making this an exponential time algorithm. Therefore, this approach is impractical, since computation will take too long.

In order to improve the efficiency of this computation we use the Forward Algorithm. We must first define a parameter alpha that represents the probability of observing the first t outputs in O given that the system is in state S_i at time t:

$$\alpha_t(i) = P[O_1 O_2 \ldots O_t \;\&\; q_t = S_i]$$

To find the probability of $O_1, O_2, \ldots O_t$, we apply the total probability theorem and sum alpha over all states in the system:

$$P[O_1 O_2 \ldots O_t] = \sum_{i=1}^{N} \alpha_t(i) \quad (\mathbf{1})$$

Additionally, we can define alpha for a sequence of the first t outputs in terms of the alpha values for a sequence of length one shorter:

$$\alpha_t(j) = \left[\sum_{i=1}^{N} \alpha_{t-1}(i) \, a_{ij} \right] b_j(O_t) \quad (\mathbf{2})$$

The expression $\alpha_{t-1}(i) a_{ij}$ is the probability of $O_1, O_2, \ldots O_{t-1}$ and that $q_{t-1} = S_i$ multiplied by the transition probability from S_i to S_j. Therefore, summing this quantity over all states gives the probability of the model producing $O_1, O_2, \ldots O_{t-1}$ and transitioning into S_j on the next cycle. $b_j(O_t)$ is the probability that O_t is produced by S_j, so this whole expression gives $P[O_1 O_2 \ldots O_t \;\&\; q_t = S_j] = \alpha_t(j)$ [2].

We can also define alpha for a sequence of length 1:

$$\alpha_1(i) = \pi_i b_i(O_1) \quad \mathbf{(3)}$$

Now, we can see that we have all of the elements of a recursive algorithm. We have the high level function (1), the recurrence relation (2), and the base case (3). This doesn't help with efficiency but makes the code a lot cleaner and easier to debug. To calculate the efficiency of the forward algorithm, we note that each of the T time steps has one alpha value for each of N states. So, we have to calculate NT alpha values. Each alpha value is calculated by summing N alpha values for a sequence of length one shorter. Therefore the efficiency of this algorithm is $O(N^2T)$, which is a dramatic improvement over $O(N^T)$ [2]. The increase in efficiency stems from the fact that many state sequences have fragments that are identical. The forward algorithm implicitly accounts for this by not re-computing alpha values that are already known.

We used the K-means algorithm to train our HMMs. The algorithm finds K clusters of MFCC vectors and each cluster represents one state. Theoretically, each state represents either silence, or a phoneme. Phonemes are the most basic element of human speech. Basically, every consonant and vowel sound is a phoneme. Every word's HMM was given a different number of states based on the its number of phonemes. We assigned the number of states to be the number of phonemes in the word, plus two states representing silence on either end of the word. For instance, the word "cat" has 3 phonemes: 'k', 'aa', and 't', so we assigned the HMM for "cat" 5 states.

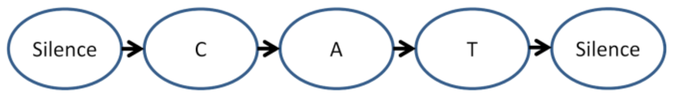


Figure 3.7: State Diagram for a Speech Recognition HMM

Another special property of our HMMs is that the sequence of states must be highly ordered. For example, in the word "cat", each state can only transition either to itself or to the next phoneme. They can never transition in reverse. The two silence states must be completely different states because if they were the same state, then it would be possible for the model to go in cycles. The same applies if a word has duplicate phonemes, e.g. "professor" contains the phoneme 'r' twice. Because each state can only transition to one other state, the calculation of the transition probabilities based on the training data becomes very simple:

$$P(transition) = \frac{number\ of\ transitions\ out\ of\ the\ state}{number\ of\ samples\ spent\ in\ the\ state}$$

Since the system can only enter and leave a certain state once during an utterance of a given word, then the number of transitions out of the state is the same as the number of utterances of the word are used to train the model. The number of samples spent in the state is equal to the cluster size.

$$P(transition) = \frac{number\ of\ training\ instances}{cluster\ size}$$

The only exception to this rule is the final silence state, which is never allowed to transition.

The training also needs to give us a way to calculate $b_j(O)$, the output probability distribution of each state. However, unlike the examples of HMMs presented earlier in this section, the outputs of our states are MFCC vectors, which have continuous values. Therefore, we need to obtain a continuous probability density function of the MFCC output vectors for each state. We do this by calculating the mean and standard deviation of each cluster to obtain a Gaussian distributed $b_j(O)$ for each state.

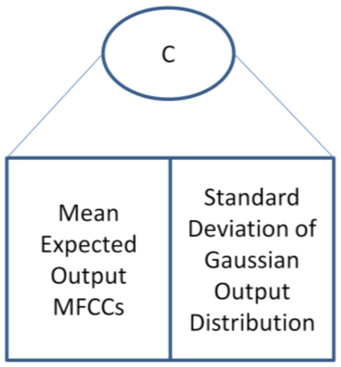

C
Mean Expected Output MFCCs
Standard Deviation of Gaussian Output Distribution

Figure 3.8: State Parameters for a Speech Recognition HMM

Another requirement for training our HMMs is that the states must be in the correct order. However, simply performing K-means on all of the MFCC vectors doesn't give any information about the ordering of the states. To fix this, we introduce timestamps as a 14th dimension in our training data. When we average the vectors in each cluster, we also obtain the average time of that cluster, which allows us to put the states in order.

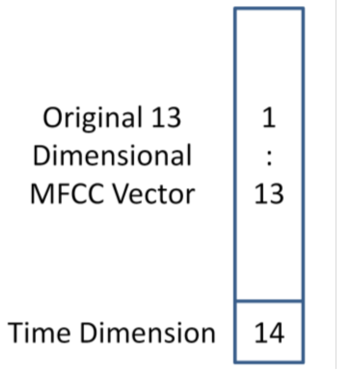


Figure 3.9: Time Dimension for Training

The time dimension has the added benefit of encouraging clustering of MFCC vectors that occur around the same time. This is desirable because the MFCC vectors that make up a phoneme should occur in one contiguous block. This also allows the algorithm to find more than one cluster for the same phoneme in words in which the same phoneme occurs more than once e.g. "professor". In order to normalize the effect tht the time dimension has on clustering, we find the maximum and minimum valued MFCC coefficients (not vector of coefficients) out of the entire training set. These define the range of timestamp values. The earliest sample will have a value equal to min and the latest sample will have a value equal to max. Each sample increments the time dimension by $\frac{max-min}{\#\ of\ samples}$. The time dimension is thrown out after training because keeping it as part of the HMM would make the model less tolerable to different variations on how a word is spoken *e.g.* “cat” vs “caaat”.

E. Software / Hardware

The TI L138/C6748 Development Kit (LCDK) is a development kit for an embedded system that optimizes digital signal processing. Although the LCDK has only 128 MBytes in memory running at 150 MHz, the computer architecture of the LCDK is designed for efficiency in multiply and accumulate operations. Hence, the LCDK is considered the premier device for speech signal processing since obtaining MFCCs involves numerous multiply and accumulate operations. Code Composer Studio 8 is an integrated development environment used to develop software applications for the LCDK in the C programming language. The LCDK maintains a function that periodically interrupts the main program and executes its code. This function, called the interrupt handler is commonly used for dealing with input/output from the ports on the LCDK. In our project, the interrupt function processes speech signal from a microphone and suspends normal program execution until the interrupt function is completed.

An important consideration in developing Tad the Therapist was the computer number format. Due to the memory limitation of the LCDK, 32 bit floating-point numbers were used instead of doubles to save memory and to increase speed. However, the long double type was used in the C++ program on the computer because the precision mattered. The forward algorithm calculates the probability of an HMM to produce a given sequence of MFCCs. Because of the Gaussian distributions assigned around the expected MFCC values for each state, MFCC values far from the expected values can result in very low probabilities, sometimes as low as 10^{-10} or less for a single alpha value. Since all of the alpha values get multiplied by each other, this can drive the output probability to $\sim10^{-700}$. The precision of the double data type is around 10^{-300}. Therefore, these small alpha values can drive the probability to zero due to double overflow in some cases. Hence, we instead use the long double type. We also introduced a scale factor in the probability calculation to guarantee that overflow cannot occur. The program is formatted such that the main function calls the probability functions of each HMM, and if overflow occurs (if all of the probabilities are zero), then it will increase the scale factor and repeat the computation. The code snippet below shows how the Gaussian probability is calculated. This is where we introduced the scale factor to prevent double overflow.

```
long double State::gaussProb(const vector<long double>&
input, const long double& scale) const
{
     vector<long double> distSquare = squareDist(input);
     long double result = 20;
     for (int i = 0; i < NUM_MFCCS; ++i)
     {
        result *= ((scale / (root2pi * m_StDev[i])) * exp((-
        1) * distSquare[i] / (2 * m_StDev[i] *
        m_StDev[i])));
        result /= 58;
     }
     return result;
}
```

The C++ computer program utilized a makefile that contains a list of instructions on how to build the entire program. The makefile made use of g++ to compile C++ code. The code repository for Tad the Therapist became so large that it would not be reasonable to fit the entire source code into a single file. Therefore, a makefile was used to link multiple C++ files together. The makefile created four executable programs: chat, train, trial, and main. The chat executable was used to test the functionality of the chatbot. The train executable was used to perform the K-means algorithm on the MFCCs training set in order to produce the mean MFCCs and standard deviations for the HMMs. The trial executable tested the LCDK speech input to the preprocessed HMMs with the known words. Lastly, the main executable combined the functionalities of the trial and chat executables. The below code snippet displays a part of the makefile and how the executable programs depend on different C++ files.

```
chat.o: chat.cpp chatbot.h
   $(CXX) -c chat.cpp
trial.o: trial.cpp duplex.h ml.h
   $(CXX) -c trial.cpp
train.o: train.cpp duplex.h ml.h
   $(CXX) -c train.cpp
main.o: main.cpp duplex.h chatbot.h ml.h
   $(CXX) -c main.cpp
```

The hidden Markov models and states were each classified as objects in C++. This paradigm follows the principles of object-oriented programming. Every HMM and state object maintained its own set of private variables and methods to handle computations. These objects also had constructor functions to initialize its private variables. For instance, the state object has arrays containing the mean and standard deviation of the MFCCs given from the K-means algorithm, while the HMM object maintains an array of its states and a transition probability matrix. Object-oriented programming simplified the development process because all the probability calculations were confined within the objects themselves. The below code snippet conveys the class declaration of an HMM.

```
class HMM {
  public:
   HMM(string word, vector<State> states, vector<vector<long
   double>> transProb);
   // Returns probability that the input was produced by the
   // HMM
   long double prob(const vector<vector<long double>>&
   input, const long double& scale) const;


   // Debug functions
   void printTransProb();
   void printStates();


   // Returns the word of the HMM
   string word() const { return m_word; };

private:
   string m_word;
   int m_numStates;
   vector<State> m_states;
   vector<vector<long double>> m_transProb;

   long double getAlpha(const vector<vector<long double>>&
   input, const int& tailIdx, const int&
   stateIdx,vector<vector<long double>>& alphaVals, const
   long double& scale) const;
};
```

The chatbot receives the identified word as input and outputs a predetermined phrase. This function uses a hash table that maps a recognized string to an array of phrases that the chatbot chooses from. A hash table is a data structure that maps keys to values and computes a hash function to find the associative values of the key. In our case, the key is the recognized word and the value is the array of possible phrases.

Both the computer program and the web server maintained an event listening function. The C++ computer program was placed in an infinite while-loop and checks every second whether the LCDK created new MFCC inputs. The web server has a library that automatically detects changes in the output text file that stores both the recognized word and the automated response.

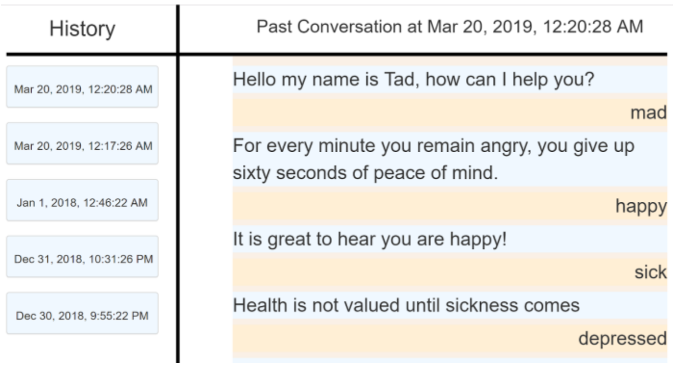
The web server handles the logic of retrieving previous conversation histories, initializing new conversations, and appending new messages from the computer program. Additionally, the web server also utilizes a text-to-speech engine in order to voice phrases. The web server was developed on a Node.js framework. The front-end was developed using a React.js framework that provided an intuitive interface for the client to interact with the virtual assistant. All conversations are stored in a text file. A MySQL database was used to store file names of these text files. The user always has the option to save the conversation with Tad. The MySQL database uses SQL statements in order to perform create, read, update, and delete operations on all text names. The web server respectively deletes the text files when needed. The snippet of JavaScript code below demonstrates the SQL statements used to store and modify these text files.

```
const query_one   = 'SELECT * FROM Conversations;';

const query_two   = 'UPDATE Conversations SET File_Name =
"' + new_file_name + '" WHERE Time_Stamp="' + time_stamp +
'";';

const query_three = 'INSERT INTO Conversations (Time_Stamp,
File_name) VALUES ("' + time_stamp + '", "TEMP_NULL");';

const query_four  = 'DELETE FROM Conversations WHERE
File_Name="TEMP_NULL";';
```



History
Past Conversation at Mar 20, 2019, 12:20:28 AM
Mar 20, 2019, 12:20:28 AM
Mar 20, 2019, 12:17:26 AM
Jan 1, 2018, 12:46:22 AM
Dec 31, 2018, 10:31:26 PM
Dec 30, 2018, 9:55:22 PM

Hello my name is Tad, how can I help you?

mad

For every minute you remain angry, you give up sixty seconds of peace of mind.

happy

It is great to hear you are happy!

sick

Health is not valued until sickness comes

depressed

Figure 3.10: Web User Interface

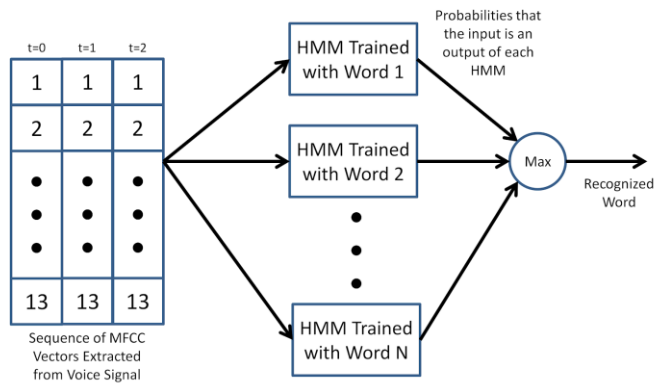
F. *System Build / Operation*

The entire system is comprised of three main components: the LCDK, the computer program, and the web server. The LCDK processes the speech signals, the computer program handles the speech recognition and automated response, and the web server provides the user interface for the client interaction.

The user says a word into a microphone. The LCDK records the input from the microphone as a time-domain audio signal. The LCDK computes the necessary transformations in order to process the speech signal as MFCCs. In order to train the system, every known word in the vocabulary list is spoken 10 times into the microphone. These MFCCs data sets are saved as text files, with the file name corresponding to the known word. For instance, the MFCCs for the word “cat” would be saved as cat1.txt, cat2.txt, cat3.txt, …, cat10.txt. The makefile in the computer program provides a train executable to read the saved MFCC text files and run K-means. After training is complete, the transitional probability matrix and the states associated with a given word are saved in the /computer/transProb/ and /computer/words/ folders.

HMMs and states are preprocessed in the main program. All communications between the LCDK, the computer program, and the web server are done using text files.

When all the training is complete, the user can start interacting with the therapist program. The user turns on the LCDK. The LCDK performs a bit flip in the /lcdk/start.txt file to indicate the start of a therapy session. The computer program recognizes this bit flip and immediately outputs Tad’s first phrase into an empty file called /computer/output.txt. The web server detects this change and voices Tad’s first message for the conversation. The user then flips the switch 5 on the LCDK to on. The user is given 1.3 seconds to speak a single word into the microphone. The LCDK then exports the MFCCs as /lcdk/input.txt and performs another bit flip in the /lcdk/done.txt file. The computer recognizes the bit flip in /lcdk/done.txt and triggers the event listening loop to feed the MFCCs from /lcdk/input.txt into every HMM and determine the word with the greatest probability.



t=0
t=1
t=2
1
2
13
Sequence of MFCC Vectors Extracted from Voice Signal
HMM Trained with Word 1
HMM Trained with Word 2
HMM Trained with Word N
Probabilities that the input is an output of each HMM
Max
Recognized Word

Figure 3.11: Using HMMs to Identify a Word

The computer automates a response for the recognized word using a hash table. The computer program appends the recognized word and the selected phrase into /computer/output.txt. The web server detects changes to this file and appends the newly made messages onto the browser interface. The user continues switching the flip on and off however many times to continue the conversation. The interaction with the computer program terminates when the user says "goodbye". In order to initialize a new therapy session, the user must restart the LCDK. The user also has the option to save the conversation history,

When the user interacts with the system, the user is able to do four things: start the LCDK, flip a switch on the LCDK, speak into a microphone, and view the interaction on the browser interface. The mathematical complexities involved in the system is hidden away from the end user. This was designed to make the operation as simple as possible.

IV. Results

In this section, we present our results. Our project was mostly successful with a few limitations.

A. *Description of Results*

The entire system as a whole is durable and has great reliability in transmitting information among the LCDK, the computer, and the web server. The system was able to distinguish the following words with 100% accuracy over 30 trials: dog, cat, happy, professor, angry, sick, miserable, depression, goodbye. Even with accents or variations in pronunciation of a word (e.g. "cat" vs. "caaat"), the system does identify the spoken word correctly.

Despite all the successes, the system does have some fallbacks. The system could not identify the word "sad" properly; instead, the system identifies the word "sad" as "sick. Additionally, a noticeable issue is the latency that occurs between the user speaking and viewing the results on the web browser. Additionally, HMMs are not capable of identifying new words that have never been recorded before.

B. *Discussion of results*

The system was able to distinguish between a variety of different words because of the way the HMMs were trained. Varying the clustering size did improve the distinguishment between words. Since the HMMs were designed to only identify words based on phonemes, any accents or varying pronunciations will likely only slightly lower the probability value determined by the HMM. Regardless, the HMM associated with the known word would remain to have the greatest probability.

The system fails to identify the word "sad" and instead outputs the word "sick". There are a number of sources of error that could cause this misidentification. For instance, the training set used to pronounce the word "sad" could be faulty or not spoken within the LCDK time frame. As a result, additional training sets should had been used. Both words "sad" and "sick" were given a cluster size of 5, so this also added a similar trait between the two words. Lastly, the forward algorithm of the HMM may be computing overflow since different scale values were used. As a result, when speaking the word "sad" into the microphone, the HMM associated with "sad" may have a lower probability value than "sick" because of this overflow and scale factor. The solution to overcome these sources of error is to add more training sets across every word. Instead of having every word have 10 sets of MFCCs, this could be changed to 100 sets of MFCCs in order to guarantee accuracy. Moreover, a uniform implementation of the forward algorithm with the same scale factor for every HMM would make every probability computation standardized. Currently, every HMM may use a different scale factor until the forward algorithm avoids integer overflow.

Another issue to consider with the system is the latency between the user speaking a word and viewing the response on the browser interface. The bulk of the latency is due to the LCDK writing the MFCCs onto a text file. This is due to the JTAG emulator speed for writing and reading files between the LCDK and the computer program. The alternative is to replace the JTAG emulator with a different JTAG emulator that has a higher data transfer rate.

The last flaw that does not make this system practical is the inability for the system to recognize new words. Every HMM was associated with a known word. However, this proves to be impractical because when a user speaks a single word into the microphone, every existing HMM has to compute a probability. The only way for a system to learn from new words is to be able to identify the phonemes associated with the new words. Hence, an entirely different paradigm must be used, such as recurrent neural networks or associating every HMM with a phoneme instead of a word.

It is important to note that given the scale of the system and the limitations of the LCDK, the system does achieve its intended goal of single word speech recognition.

REFERENCES

[1] B. H. Juang and L. R. Rabiner, “Automatic Speech Recognition – A Brief History of the Technology Development,” Georgia Institute of Technology, Atlanta, GA, USA, Rutgers University, New Brunswick, NJ, USA and the University of California, Santa Barbara, Santa Barbara, CA, USA. Oct. 8, 2004. https://web.ece.ucsb.edu/Faculty/Rabiner/ece259/Reprints/354_LALI-ASRHistory-final-10-8.pdf

[2] N. S. Uchat, “Seminar Report on Hidden Markov Model and Speech Recognition,” Department of Computer Science and Engineering, Indian Institute of Technology, Bombay Mumbai, Mumbai, India.