The Efficiency of Programming Through Automated Speech Recognition

Presented to the
Department of Computer Science

In partial fulfillment of
the requirement for the degree of
Bachelor of Science

Haverford College
Matt English
2015

Abstract

The goal of this thesis is to present information which will support programming through speech recognition by finding a limit for the size of commands in the given language of the program. This limit is the point at which it will take too long for speech to be converted into text by the Viterbi Algorithm, based on the size of a Hidden Markov Model. The paper begins by introducing how speech recognition works, presenting some common issues when attempting to program by voice, and the overall motivation behind the research presented. A history of speech recognition is shown to support how programming by voice has evolved positively over time, allowing users to do more by voice alone. The algorithms which convert speech into text are formally discussed to provide an understanding of their runtimes which will be the main focus of the experiment. The future of speech recognition is also discussed based on speculations of its advancement technologically and based on the results of the experiment itself.

Acknowledgements

I would like to thank John Dougherty, my thesis advisor, for motivating and inspiring me throughout the year. I would also like to thank Professor Steven Lindell for giving me pragmatic advice that was instrumental to the completion of my thesis. I would not have made it without your continued guidance and support over the course of this project.

Table of Contents

Chapter 1 - Introduction

1.1 Introduction to Speech Recognition

The constant use of the hands while working on a keyboard puts a great deal of stress and strain on the fingers that, if left untreated, can become a more serious issue. There is a rise of RSI (repetitive strain injury) which correlates to the high amount of repetitive coding of programmers and the increasing amount of time spent on a keyboard as technology advances. Studies have also shown that jobs which require dependency on computer usage, specifically "working with a computer … more than six hours per day was associated with WRULDs[1] in all body regions" (Blatter & Bongers, 2002). In many cases, programmers are left unable to perform because of the immense pain caused by RSI. The usage of the mouse and keyboard to program may also be an inefficient way to code when going back and forth between navigating the computer screen with the mouse and typing with both hands. This exchange prolongs the amount of time that you are working as each switch causes a pause in work-flow. There are techniques available to minimize the use of a mouse to navigate code or select certain sections of your code, but these may not be the most suitable options for improving programming efficiency because they still require multiple keystrokes to execute.

What if there was a way to program efficiently without having to use the keyboard and mouse as much, or perhaps even at all? This would benefit computer users who are not physically able to type or control a mouse due to disability or even those who may work faster or better with vocal software rather than by touch. This technology would also take care of the mouse to keyboard to mouse problem, eliminating one form of delay.

Speech recognition software can be used to convert dictation into text. The user simply talks into a microphone or headset which will generate a word-for-word transcription of their

[1] "work related upper limb disorders"

speech, perform actions based on what was said, or even create code based on predetermined templates. These templates are pre-determined by a speech recognition program or personalized by creators of macro commands. Speech opens new possibilities by allowing programmers to generate the same length of code in fewer spoken words than they would end up typing. Programmers may also be able to code quickly and with enough accuracy where the vocal software may replace keyboard and mouse programming altogether.

1.2 Functionality of Speech Recognition

There are many steps in the process of converting speech into functioning code. First, sound waves are taken as input through a microphone or headset. These audio signals must be transformed into some form of data through a process known as Fast Fourier Transformation. The frequencies found at each small time interval analyzed are given numerical values (*i.e.,* data) that correspond to specific phonemes:

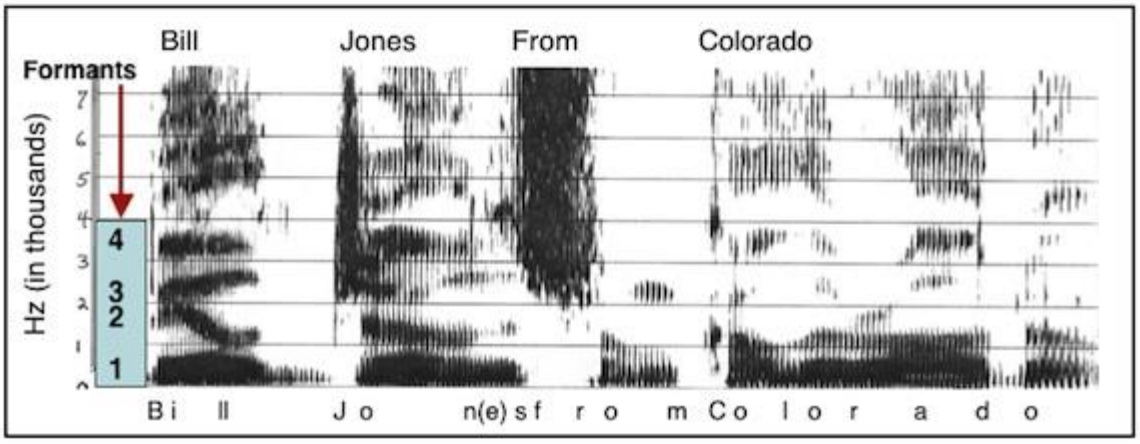

Fig. 1 – This shows the spectrogram for "Bill Jones From Colorado." It has already been transformed from its waveform into a spectrogram so its frequencies over the whole phrase (measured in kilohertz) can be utilized to produce phonemes. [2]

These phonemes, of which there are roughly 40 in the English language, are the most basic building blocks of language, even deeper than syllables. For example, the word "bat" has one

[2] Staab, Wayne. Spectrogram. 2014.

syllable, but in that one syllable there are multiple phonemes that generate each sound within the solitary word “bat” (*i.e.,* b/æ/t phonemically). The speech recognition program must piece together these phonemes one at a time by looking back at the previous phonemes chosen (otherwise accuracy would be extremely low without any context for what phonemes should normally come next in a series).

Phonemes are pieced together through either two approaches: Hidden Markov Models (HMMs) or Neural Networks. In both of these techniques, there is a preset system of probabilities between states which is used to generate the text that was hidden from the system before the final result is displayed. With HMMs, nodes are connected from start to end when all of the phonemes from the original sound wave are pieced together. Neural networks work like HMMs where there are input nodes of possible phoneme choices and weights are assigned to each connection from one phoneme to another, but over time, the network updates the weights of its connections through algorithmic learning. Both techniques are feasible solutions to speech recognition, but this thesis will focus solely on applications with HMMs and how the Viterbi algorithm uses utilizes its parameters. The complex part of these processes is each node branches off to all possible nodes from a system’s predetermined dictionary for what the next word or phrase could be (or when a single word/phrase should end as well). This becomes more complex quadratically (“the time complexity of this algorithm is $O(N^2T)$” for the search algorithm where N is the number of hidden states in the HMM and T is the length of the observation sequence of phonemes) as the vocabulary increases (Chatterjee, 2012). Nodes at each step of the algorithm branch out to more nodes, creating more possibilities and decreasing guaranteed probabilities for certain phrases: “If a program has a vocabulary of 60,000 words (common in today's programs), a sequence of three words could be any of 216 trillion possibilities” (Grabianowski, 2006). Each phoneme is given a preset probability of when it

should come next in a phrase based on its previous path of nodes.

As phonemes are chosen, probabilities are constantly updated through Bayesian Inference and the Viterbi Algorithm. The Viterbi Algorithm finds the proper path to take by piecing together phonemes, while Bayesian Inference accounts for the change in probabilities over time. Bayesian networks "can model an arbitrary set of variables as they evolve over time" where probabilities are updated as more information is input into the system (Zweig, 1998). The Viterbi Algorithm looks at a state and finds the optimal next state to move towards based on its previous path and the probabilities of moving from the given state to all possible future states. Each edge stores the probability from one state to the next. This process of updating probabilities and choosing phonemes is repeated until the final phoneme completes a full path from a start state to a final state. A pseudo code representation of the algorithm will be presented in chapter three.

When the full word is translated from the user's speech into text, the computer has two final choices. It can either output the exact words that were spoken or it can pass these words through another function that will map the textual output to programmable code. The translators for speech into code work by maintaining a created library of commands that will output the desired code. Users create their own languages of commands as templates by designating a specific command as a String. These phrases are then translated into code as another String of text or algorithms that are able to be compiled.

1.3 Common User Issues

There are many issues with speech recognition which range from user error to issues with the software itself. One big issue is the need to be in a quiet space or to have a device that is able to reduce the effects of background noise. There are also problems that arise from the

generation of text within a text editor (from a programming perspective). Even simple differences in speech and its complexities can cause errors when attempting to translate a spoken phrase into text.

If there is any background noise, it could interfere with the speech input causing errors in word/phrase detection. These errors can be reduced somewhat by audio normalization within the computer itself, but only to an extent. This process of normalization adds gain to an audio recording so it pinpoints higher amplitudes which, if the background noise is softer than the speech into the headset or microphone, will target phonemes and essentially ignore other sounds that the headset or microphone picks up. Professional headsets and microphones also help reduce unwanted noise, but only if the noise is quiet enough.

New complications arise when attempting to program by voice alone. The need to create new variables or recall previously used variables may cause issues as the speech recognition program may not be able to decide which terms should be new text or which should be seen as already-defined variables. This can be solved through backtracking to previously used variables that have been stored or even by declaring variables with special commands to eliminate type errors. Error correction is also an issue because you have to navigate the code by voice, delete the mistake that was made, and repeat what you were trying to say previously. Jumping back and forth to where errors may have occurred is time consuming. Code navigation itself is tricky as the location that you are dictating could be confused with other areas within the code. Numbers can also be problematic as the program will have to decide whether to give the numerical or alphabetical representation as output (*e.g.,* “two” versus “2” which could cause some type errors for strings and integers, respectively). One of the hardest issues to overcome is the want to wait for text to appear as you say it instead of speaking and having confidence that the words you say will be transcribed correctly.

The final issues with speech recognition are those that have to do with the speech itself. People have varying accents, which make the choosing of words difficult for the speech recognition program. For example, depending on the dialects or accents used, the phrase "recognize speech" could also be heard as "wreck a nice beach." Homonyms can be an issue as well. Consider, how would a program decide between outputting "to" or "too" or "two"? Modern speech recognition systems use many techniques to reduce possible errors caused by choices between homonyms. One way this is fixed is through looking at the context of the previously chosen parts of a dictated phrase. For example, if you have the word "going," the system will give a higher probability to "to" rather than "two" as the next possible word. The language model portion of speech recognition eliminates errors as well when it comes to sentence structure and how certain words are said. "To," "too," and "two" sound similar, but their frequencies vary enough to pinpoint the differences in how each are said. The easiest solutions to most of these issues are user and program training. The user can calibrate their voice to the program so their frequencies are normalized and can be picked up easier rather than having to generalize the given frequencies for all users.

1.4 Motivation Behind Speech Recognition in a Programming Context

I believe that speech recognition may be beneficial in a programming context because of its capabilities, along with its benefits to users who are not able to type, to reduce unnecessary typing and to improve input speed. In text editors, you can say a phrase or even a word to generate a large chunk of code through the use of templates and macros. The use of speech will also eliminate the necessity to type out special characters like semi-colons, brackets, and other characters that may require multiple keystrokes simultaneously.

The difference in speed between typing versus speaking, measured in words per minute,

also benefits the argument that coding by speech may be more efficient than typing simply because we can speak faster than we type. The average speaking speed for speech recognition software is “at about 105 words per minute” (Karat, Halverson, Horn, and Karat, 1999; Lewis, 1999). When you factor in error correction, this speed drops drastically, but typing speeds of which “many jobs require keyboard speeds of 60-70 words per minute” are shown to be much slower than speaking (Human Factors International, 2000). When it comes to programming, this unit of measurement may seem out of place, as most programmers view full files of code by how many lines of code are present rather than how many words were used. For one person, ten lines of code could take a short amount of time to type while for another, ten lines of code could take a much longer time. Code can also vary on the amount of lines used for the same amount of words. For example, the two varying codes below use the same amount of words, but have different lengths of lines used:

1)
```
int main ()
{
      std::cout << " Hello World!";
}
```

2)
```
int main () { std::cout << “ Hello World!”; }
```

Typing will produce less error if you are careful, but with user training, the accuracy of speech recognition software could be improved to the point where the user may be able to confidently speak at the average words per minute required.

HMMs are crucial in speech recognition for programming purposes because they store the voice commands which can be used to produce text or perform actions. When users create new commands or generate their own grammar, more states must be added to a HMM to store possible phrase choices. There are pros and cons to having a large and small library of commands (represented by a large or small amount of hidden states within the HMM). With a

small library of commands, the user would have to code more for each template, which requires a lot of work done before the commands can even be used. However, if the commands are meant to be used frequently, then it makes sense that one would not want to have to program a long amount of code many times by hand. With a smaller library, though, the programmer has less power to perform actions by voice. With a larger library of commands, the user has more freedom to code by voice, but a large library may take too long to transfer voice into text.

The remainder of this thesis will focus on the history of speech recognition, delving into the algorithms for speech to text, and focusing on a specific aspect of HMMs that may be able to be improved through altering the size of the model. Chapter two describes general speech recognition uses along with programming specific uses that have improved over time. Chapter three will provide a more formal look into the various algorithms that take in a user's vocal input and transform these sound waves into the desired code. Chapter four will describe my experiment on the pros and cons of having a long or short library of possible phrases for the model to choose between. In chapter five, I will document my results as well as state where the future of speech recognition can go based on the results of my experiment. The final chapter will discuss possibilities for the future of speech recognition and issues that still remain unsolved.

Chapter 2 - Related Works

2.1 Previous Speech Recognition Software

Speech recognition software dates all the way back to the 1950s. Initially, speech recognition “could understand only digits” (Pinola, 2011). In 1952, Audrey was created which showed the world a system that could recognize spoken numerical values. The Shoebox machine was later created and shown at the World's Fair, improving upon Audrey by being able to recognize sixteen words. In the 1970s, the Department of Defense created a “Speech Understand Research (SUR) program” called DARPA which influenced Carnegie Mellon's Harpy system, being able to “understand 1,011 words” (Pinola, 2011). Harpy was also responsible for a newly used beam search that would “prove the finite-state network of possible sentences” (Pinola, 2011).

Over time, the amount of words able to be recognized increased in the thousands due to the new use of HMMs, allowing for probabilities to improve accuracy of textual output. Many new programs were released after this discovery that allowed for the user to dictate continuous speech rather than having to pause after each word spoken. The arrival of smart phones and advanced mobile technology brought about the use of voice search through Google Voice Search and Siri. Even gaming consoles have built-in software which allow the user to speak towards the console (or a hardware extension) to easily perform tasks without the need of a controller.

2.2 Programming Specific Uses of Speech Recognition

Over time, speech recognition systems have drastically improved for coding purposes. In a paper by Alain Desilets (2001), various issues were compiled and compared to previous systems before VoiceGrip (now VoiceCode), his own software which sought to fix or give

applicable solutions to programming-related complications: punctuation, symbol use/creation, code navigation, error correction, mouse-free operation, cross speech recognition availability, and cross editor availability. Speech recognition technology for programming came into play with John Leggett and Glen Williams. They asked if voice coding alone was more efficient (*i.e.,* faster and more accurate) than keyboard and mouse programming, and found that voice coding was less efficient:

> The results showed that the subjects were able to complete more of the input and edit tasks by keyboard (70%) than by voice (50–55%), but that keyboard input had a higher error rate than did voice input. Also, the use of voice was just as efficient as keyboard for the inputting of editing commands. These results must be viewed with the understanding that the subjects were novices with respect to voice input, but were very experienced with keyboard input. (Leggett & Williams, 1984)

This system allowed for punctuation to be automatically inserted, but their experimental techniques did not account for life-like situations. The participants in the study were given exact text to repeat rather than having to use their own knowledge to choose the correct commands to return desired code. This also eliminated the need to navigate throughout the code as the participant simply had to speak the code given to them. Also, the symbols used in the given code were predefined, so the testers would not encounter the issue of having to clarify new symbols. While the experiment was more open to the public, it did not properly simulate how an experienced programmer may have used speech recognition.

Petry and Pierce's system was even more basic, focusing on the speech recognition itself rather than fixing user issues. Specifically, their BASIC system would use "syntax-directed processing of the input string to permit the context to determine an appropriate word subset and

potentially correct speech recognition errors." (Petry & Pierce, 1981). Bettini and Chin's software introduced a new feature for debugging where the user was able to navigate through the code by dictating line numbers or subroutines at which to break. DEMACS allowed for templates and macros to be used which would generate large chunks of code from little amounts of speech and was the first speech recognition program to start the trend of being Cross SR (this means that the software could be ported to multiple speech recognition programs). For example, saying "'for loop' can be used to enter the following C code: for (; ;) { }" while simultaneously putting the cursor at the first spot after the open parenthesis for an argument (Desilets, 2001). This helped out with punctuation, but the user would still have to navigate through the code by mouse. Also, users would need to pause after each dictation, causing voice programming to be slow. ELSE (Emacs Language Sensitive Editor) improved upon DEMACS by also allowing the user to navigate through their own templates with specific commands.

CachePad was an interesting new idea because it allowed the user to create a list of new symbols that were not predefined in the software. It also allowed the user to choose from a list of symbols (predefined and ones they've created) by saying "symbol" and a number that corresponds to that symbol in a list. Also, the user could navigate code by specifying a line number. EmacsListen also introduced some interesting features that allowed for users to dictate programming commands. This allowed for easy error correction because if a command was not recognized, it would simply be deleted and would not run. Also, the user was able to combine vocal commands with mouse/pointer position to make copy/pasting pieces of code a simple task.

VoiceGrip/VoiceCode was a reaction to all of the speech recognition programs before it and tried to address all of the previously listed usability issues. VoiceGrip works by allowing "programmers to dictate code continuously using a pseudo code syntax that is easier to utter

than the code itself" (Desilets, 2001). It takes in this pseudo code like language and translates it to code that can be compiled through its Code Translator. It makes use of templates as well. For the example of the if-statement, simply saying "then" will create brackets after the "if" and its arguments. Symbols can also be said how they mean rather than trying to exactly replicate words. Instead of speaking words for each letter like "mike yankee cap hotel echo india golf hotel tango" for myHeight, you could simply say my height and the program would know you are using that symbol. In other words, the program uses context based on previously used symbols to see what you are working with. Navigation is also made easier by allowing the user to say a command "find code" and dictating to where in the code they should jump. This software was also Cross Editor and Cross SR, making it the most accessible speech recognition software at the time.

The Harmonia Research Project at the University of California at Berkeley designed its own text editor (SPEED[3]) which allows users to create and edit their own programs using the software Spoken Java:

> Spoken Java code is ultimately translated into Java as it appears in the program editor. Spoken Java is designed to be semantically equivalent to Java – despite the different input form, the result should be indistinguishable from a conventionally coded Java program. (Begel, 2005)

The syntax of Spoken Java appears as pseudo code, but its structure is so close to Java that it can be translated easily so it can be compiled through text editors which accept the Java language. It contextually decides how to translate the pseudo code into actual functioning code by brute force. When a user utters a command or chain of commands, Spoken Java "uses knowledge of the Java programming language as well as contextual semantic information valid

[3] "SPEech EDitor"

where the utterance was spoken" to sort elements based on their structure (sorting by spaces or homophones) and eliminate elements which could not possibly work (Begel, 2005). In its current state, Harmonia is being ported so it can be used in Eclipse as well and also supports languages such as C, Scheme, Cool, and Titanium.

Voicecode.io, a very recent speech recognition coding program created by Ben Meyer, seeks to increase user productivity by voice in many computer applications. By voice, you are able to control most aspects and programs on the computer. You are able to write "anything from emails to kernel code, to switching applications or navigating Photoshop" and program in any language based on the system's static, variable, and grammar commands (Meyer, 2015). What separates this program from other voice programming software is that "commands can be chained and nested in any combination, allowing complex actions to be performed by a single spoken phrase" (Meyer, 2015). So far, the software only works on Mac operating systems.

The most recent addition to the field of SR software is really in the hands of the user. There are many speech recognition programs, the most reliable being Dragon Naturally Speaking, with an advertised accuracy of up to 99%, and Windows Speech Recognition. However, it is the software which utilize these speech recognition programs that are of highest importance to the growing world of computing. Programmers are able to create their own libraries of commands through the use of voice-command languages which vary based on the efficiency of their templates. A powerful extension to Dragon Naturally Speaking is Dragonfly, a Python extension which allows the user to create their own custom commands to output specific text. Through Dragonfly, Rudd created his own 2,000 long list of commands for Emacs based on Shorttalk, a design for ways to more effectively navigate text files. This extension, he claims, helps him code faster than when he typed before. The only problem with Dragonfly is that it solely works with Python, so you would also need an editor that is able compile code

written in the Python language. The basic template for creating commands is listed below:

```
from dragonfly.all import Grammar, CompoundRule

# Voice command rule combining spoken form and recognition processing.
class ExampleRule(CompoundRule):
spec = "do something computer"                  # Spoken form of command.
def _process_recognition(self, node, extras):   # Callback when command is
spoken.
print "Voice command spoken."

# Create a grammar which contains and loads the command rule.
grammar = Grammar("example grammar")                # Create a grammar to
contain the command rule.
grammar.add_rule(ExampleRule())                     # Add the command rule
to the grammar.
grammar.load()                                      # Load the grammar.
```

Fig. 2 – This example creates a rule "do something computer" which will output "Voice command spoken." when the created command is said. Once the command is created, you must load it into your Grammar so the command will be recognized.[4]

Vocola is another language which allows the user to create their own local commands to specific applications based on given speech input. Commands can also be generated to work in any computer application or even to generate code, however, the commands are not normally global and will only work in the applications that they are defined for. Global commands can be defined, but it is not the main purpose for using Vocola. A basic example of Vocola would be creating a command to copy something. You could say "Copy that," a built-in command within Vocola, which executes "CTRL+c," the command used on the keyboard to copy objects. There are two versions of Vocola which function with Dragon Naturally Speaking and Windows Speech Recognition. Unimacro works just like Vocola, but it puts more of an emphasis on global grammars which can work on any computer application. Unimacro also utilizes some local commands, but its intent is to compile a set of commands that all users can access and utilize for many different aspects of computing.

[4] Python Software Foundation.

Chapter 3 - Speech Recognition Algorithms

3.1 Fast Fourier Transformation

The first step in translating speech into text is changing audio waveforms into their equivalent frequencies. The data changes from a time domain measured in seconds to a frequency domain which is normally measured in kilohertz. This is necessary because phonemes can be easily detected by the formants found in frequency spectrographs. These formants are high concentrations of energy which differ based on phonemes found at specific time intervals. FFTs are used because of their speed, which is necessary when speaking long phrases to ensure that they are analyzed and processed quickly. Normally, a Discrete Fourier Transform (DFT), the method of converting a set of data from one domain to another, performs with a runtime of $O(N^2)$ where N is the number of samples taken over a set amount of time. It runs at $O(N^2)$ because for each frequency index k, of which there are N values, it must take the sum of N time domain signals which requires N*N operations.

The FFT itself simply describes a quicker way of performing a DFT. The base formula for an FFT is listed below:

$$X(k) = \sum_{j=1}^{N} x(j) e^{\frac{-2\pi i(j-1)(k-1)}{N}}$$ [5]

Here, X(k) is the frequency domain, x(j) is the time domain, N is the number of time domain signals analyzed, j is the index or time at which the audio sample is being analyzed, and k is "corresponding to the magnitude of the sine waves resulting from the decomposition of the time indexed signal" (Abu, Ernawan, and Suryana, 2011). The frequency domain is stored as "a vector of N values at frequency index k" (Abu, Ernawan, and Suryana, 2011). In a paper by

[5] Abu, Ernawan, and Suryana. 2011.

James W. Cooley and John W. Tukey, they describe their algorithm, which is the most commonly used FFT, as one that "iterates on the array of given complex Fourier amplitudes," the time domain, "and yields the result in less than $2N\log_2 N$ operations without requiring more data storage than is required for the given array A" (Cooley & Tukey, 1965). The algorithm is shown below:

$$X(j) = \sum_{k=0}^{N-1} A(k) \times W^{jk}$$ [6]

$$W = e^{\frac{2\pi i}{N}}$$

In this algorithm, A is the array of time domain signals. The way that the algorithm reduces runtime to O(n*logn) is by splitting the N composite time domain signals into smaller DFTs to be analyzed and then combining them back together to complete the full time domain sequence. Much like Mergesort, the Cooley-Tukey algorithm splits its input and then dynamically builds it back together for a more efficient way of performing a DFT.

Once this algorithm terminates and the frequency domain remains, formants, which are represented by the varying frequency values, are analyzed to distinguish the uttered phonemes during short time intervals. Formants can be seen below as the varying bands within the frequency spectrogram.

[6] Cooley, Tukey. 1965.

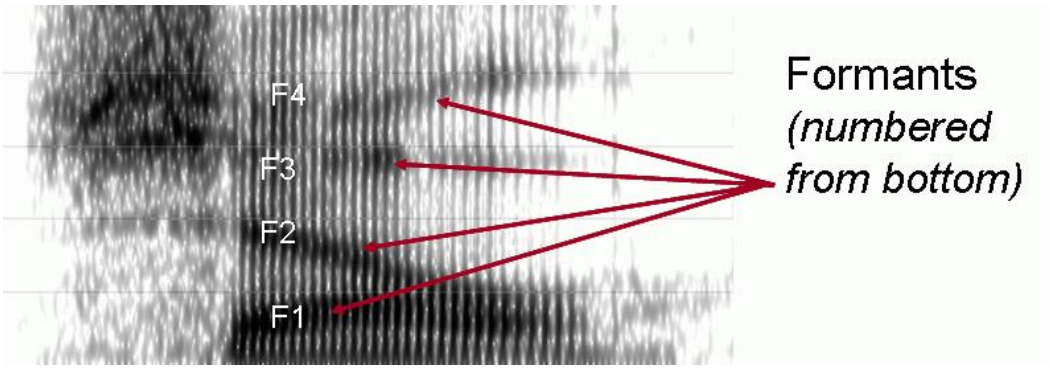

Fig. 3 – This frequency spectrogram shows four formants found at different time intervals. Each formant represents a different sound/phoneme.[7]

The time intervals analyzed within the waveform are equally spaced, so some issues of overlapping phonemes may occur. However, overlaps are accounted for by differences in energies of frequencies, so a hard or soft consonant sound will not mesh together with a hard or soft vowel sound. The phonemes for each uttered phrase (be it a word or list of commands) are then sent to a network of nodes which utilize probabilities to guess at which words they should output.

3.2 Hidden Markov Modeling

Once phonemes are generated, they are then sent to a HMM which attempts to match them to words/phrases based on probabilities designated by the model. The model satisfies the Markov property where the system can choose its next phoneme based solely on the phoneme that came before it as equally as using its full previous path. The model is additionally hidden due to the fact that the output of the system is visible, but the path of states of the Markov chain from the start node to the final output remains unknown. To increase accuracy and "model the effects of context on articulation," Markov models will attempt to match multiple phonemes at once "often by introducing states that represent triplets of phonemes ('triphones')" (Johnson,

[7] Formants. University of Iowa.

2004). These triphones can contain phoneme combinations which only use one or two phonemes as well which leave the second or second and third spaces for phonemes null.

The HMM itself consists of a set of states, a set of outputs of a language from each state including final outputs and partial outputs, two probability matrices, and an initial state distribution. The states are defined as S = {s1, s2, s3, …, sn} for a total of N states. The possible outputs for the model are defined as P = {p1, p2, p3, …, pk} for a total of K outputs. These outputs are the combinations of phonemes which produce words as output or continue the Markov chain. Say we want to find the chain for the word "potato" and we are at the point in the model where "pota" has already been built up to end at state x. The next step would be jumping to a new state x+1 with output pz = "pota" coming from state x. This means that full words and parts of words are all within the output set P. The two predetermined probability sets are stored as probability matrices. The state to state matrix stores probabilities for connections between all states. This can range from 0 to 1: 0 being when states do not connect and 1 being when there is a one-to-one relation between two states (meaning the first state only connects to the second state). The output to state matrix represents the probability that you will enter a state based on the previous output observed. The initial state distribution accounts for the probability of a state at time t. Specifically, it is defined as "$\pi = \{\pi i\}$ where $\pi i = P[q1 = Si]$, $1 <= i <= N$" where π is the initial state distribution, and πi stores the probability of starting in a specific state for the first step of a search algorithm (Rabiner, 1989). This initial state distribution is used only once to choose a state once the search algorithm to complete a Markov chain begins. If there is no initial state distribution in the system, then the start state is chosen based on the output symbol distribution. The HMM can finally be represented by its three probability sets as "$\lambda = (A, B, \pi)$" where A is "the state transition probability distribution," B is "the observation symbol probability distribution," and π is the initial state distribution (Rabiner, 1989).

There are three main questions which arise from the generation of a HMM. These problems are listed below and are crucial to being able to use HMMs practically:

1. Finding the probability of an observed sequence, in this case the sequence of phonemes entering the HMM. This is defined as $P(O|\lambda)$ which is the probability of the sequence $O = \{O1, O2, O3, \ldots, Ot\}$ of length T given a HMM λ.
2. Finding the optimal sequence of states based on the given observable phoneme sequence as input into the HMM. It would not make sense for a HMM to output a phrase with a low probability as it will most likely not match the speech input into the system. There will be a lower rate of error correction the closer you can get to a 100% match between your speech and the textual output.
3. Finding a way to adjust the parameters of $\lambda = (A, B, \lambda)$ so $P(O|\lambda)$ is optimized. This essentially is "viewed as training a model to best fit the observed data" (Stamp, 2012). This is a necessary problem to be solved because within the language model of a system, the probability of certain used words may need to change if those words are used often. The speech recognizer can more accurately lean towards using those terms rather than those that are not used as much.

3.3 The Viterbi Algorithm

This algorithm finds the path of highest probability within a HMM by dynamically building up the path of maximum probability at each state transition. This chain of hidden states based on the given observation sequence (phoneme sequence) is called a Viterbi path. The algorithm takes in as input the parameters of a HMM, an observation sequence which in this case is a set of found phonemes, and a transition index which maintains at which point you are in the observation sequence. Its output is a set of states which represents the optimal path found

in the HMM.

At each state transition, the algorithm checks the probabilities of all of the states the current state could reach next while also checking the probability that the observation sequence at the current time could go to the next possible state simultaneously. Both of these probabilities, combined with the probability of being in the state you are currently in based on the previous observation sequence, will choose the optimal next state. If there are multiple transitions of equal probability from a single state, "… then one of the transitions is chosen randomly as the most likely transition" (Ryan, 1993). A pseudo code translation of the Viterbi algorithm is listed below:

```
Viterbi_alg(S, P, A, B, π, O, t, X):

// S is the set of states of the HMM. P is the set of outputs or observations within the
// HMM. A is the state probability distribution. B is the output symbol distribution.
// π is the initial state distribution for when t is 0 when we are starting the algorithm. O
// is the observation sequence. t stores what index you are at in the observation
// sequence. X will store the final state sequence which is the optimal state transitions
// for the given observation sequence.

Initialize X as an empty list of states
Initialize t to be 0
Initialize L to be length of Observation Sequence

if  t equals L:
// Algorithm is finished as all observation elements have been checked
        Return X, the final set of optimal states

else:
        if t equals 0:
        // We are at the beginning of the HMM and the Viterbi search. Use initial
        // state distribution here.
        For all states s' in S:
            • Store probability of initial state distribution (from set π)
            • Store probability of state s' given Ot, the observation at time t (the output
              from set P and the probability from set B)
            • Choose state with the maximum probability based on multiplication of
              both values
```

else:
// We are somewhere else in the observation sequence
For all states s' in S:

- Store probability of entering this state s' based on the current state you are in (from set A)
- Store probability of current state based on Ot, the observation at time t (the output from set P and the probability from set B)
- Store probability of whole previous path up until this point
- Choose state with the maximum probability based on multiplication of all values

Add the optimal state to X
return Viterbi_alg(S, P, A, B, π, O, t + 1, X)
// Recursively return Viterbi_alg until t equals L

The runtime of this algorithm is $O(T*N^2)$. T is the length of the observation sequence while N is the amount of hidden states in the HMM. To derive this value, the algorithm must go through the full observation sequence and check each of its elements once, so we have O(T) where T is the number of elements in observation sequence O. In the worst case, all N hidden states of the HMM can connect to all other hidden states including its own state. For each observation element in the sequence O, it must check the probabilities of N hidden states, updating the runtime to be O(T*N). The final piece is that for each hidden state analyzed, you must also look to the N hidden states that it could branch out to, making the final runtime of the algorithm $O(T*N^2)$.

3.4 HMM and Viterbi Example

In order to better understand how the Viterbi algorithm works within a HMM, we will look at a basic example of choosing different colored balls from hidden urns. This is based on various urn examples demonstrating how HMM are generated and searched but with my own values and parameters. In this case, the urns represent the hidden states of the HMM because we do not know from which urn a ball is chosen, we only know the sequence of balls chosen. The choice of balls represents the possible outputs between states. For our example, each urn will

have the same probability of being the first to have a ball chosen from it, so the initial state distribution is not necessary. Say we start with three hidden urns u1, u2, and u3 within the set U. These urns each contain three balls b1, b2, b3 within the set B. For set B, b1 can represent red, b2 can represent yellow, and b3 can represent blue. The state transition is the probability of choosing a ball from the next urn (represented by columns in Table 1) based on the urn you just chose a ball from (represented by rows in Table 1). The output to state transition is the probability that a ball will be chosen from a specific urn based on the previous ball chosen. We can give a state transition distribution and an output symbol distribution as shown below:

Table 1 - State Transition Distribution Table From One Urn to Another

	U1	U2	U3
U1	0.0	.7	.3
U2	0.4	0.4	0.2
U3	0.6	0.1	0.3

Table 2 - Output Symbol Distribution Table for a Ball to an Urn

	B1 = red	B2 = yellow	B3 = blue
U1	0.8	0.1	0.1
U2	0.3	0.4	0.3
U3	0.5	0.3	0.2

Say we observe a yellow ball and then a red ball as our observation sequence. We now run the Viterbi algorithm on the sequence O = {yellow, red} to find the optimal path X of states. To start, we look at the probabilities of entering states based on seeing a yellow ball. Urn 2 has the highest probability of 0.4, so U2 is added to X. The index of the observation sequence is also

increased by one. The probability so far is set to 0.4. The next observed element in the sequence is a red ball. We know that we are coming from urn 2, so we must use probabilities for coming from urn 2 to all other states, the probability that we will enter a state based on seeing a red ball, and the probability stored so far. The probabilities are (0.4)(0.4)(0.8) = 0.128 for choosing from urn 1, (0.4)(0.4)(0.3) = 0.048 for choosing from urn 2, and (0.4)(0.2)(0.5) = 0.040 for choosing from urn 3. The highest probability is that a red ball is chosen from urn 1. We thus add U1 to X so X = {U2, U1} and the final stored probability is 0.128. Now t equals the length of the observation sequence, making X = {U2, U1} for the Observation sequence O = {yellow, red}.

4 - The Limit to the Size of a Hidden Markov Model

4.1 Introduction to Experiment

When programming by voice, in order to remain efficient, the algorithms used must be able to quickly translate speech into text to reduce time spent waiting for the program to process the speech input. Being that the runtime of the Viterbi algorithm is solely dependent on the length of the observation sequence and the amount of hidden states, we will focus on how changing these values can slow down a system. We can find an upper bound on when the two values are so large that the time it takes for the Viterbi algorithm to terminate will be impractical for speech recognition. If the length of observation sequence, or how many words are spoken in one set of commands, is linear within the runtime of $O(T*N^2)$, then increasing T will make the system slower. However, this will not be as much as if we added more hidden states to the system. Doing so would cause a polynomial increase in time as N is being squared within the runtime of the algorithm. Increasing T, however, may not be an issue as the algorithm for a FFT attempts to break down the input signals into smaller transformations. This can reduce the time it takes for each run of the Viterbi algorithm by splitting the observation sequence into smaller sequences to be analyzed. We must always deal with all hidden states as each run of the Viterbi algorithm must check all N hidden states (no way to dynamically break this down yet). I hypothesize that even though a long observation sequence will make a Viterbi Search slow as the length of the sequence increases, it could be broken into many smaller sequences to be processed in parallel time. However, the amount of hidden states of the HMM can reach a limit where a Viterbi Search will take too long to process even a small observation sequence.

4.2 Experimental design

I will be conducting an experiment through the use of the Viterbi algorithm on a simulated HMM. The HMM will contain probability matrices where each entry is of even probability based on the amount of entries in the system. This does not affect the Viterbi algorithm as each probability must be checked anyway, so the values within the matrices are irrelevant. The variables which can change are the length of the observation sequence and the number of hidden states within the HMM. The output set will remain static while the other two variables (length of sequence and number of hidden states) change. I will test the two changing variables against various output set sizes as well. A clock will be utilized when running the Viterbi algorithm to time how long it takes to complete.

Within the Viterbi algorithm itself, I will use a set S = {s1, s2, s3, …, sn} to represent N hidden states in the HMM. A set P = {p1, p2, p3, …, pk} will store k outputs within the system. The matrix A will store the state probability distribution. For an element a_{ij}, where i represents the rows and j represents the columns, the value in the matrix shows the probability of going from state i to state j. The matrix B will store the output symbol distribution where each elements b_{ij} stores the probability of entering state i based on the output in column j. This matrix will contain N rows (of hidden states) and K columns (of outputs). The initial state distribution π can be ignored as in our example, we are making all probabilities even, so a start state will be randomly chosen. The observation sequence O = {o1, o2, o3, …, ot} stores T observations entering the HMM. The output for this experiment will be measured in seconds based on the way that S, O, and P change. All other parameters will stay constant throughout the experiment.

For the experiment, I used Elana Perkoff's python Viterbi implementation and tested to see how long it ran with my own inputs. For the HMM class that she wrote, I inserted a static

output space which does not have to change as the output space has no effect on the runtime of the Viterbi algorithm. This output space contained possible outputs 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10. For the state space, I used integers to represent varying states. I tested state spaces of sizes 10, 100, 1000, 10000, and 20000. I had to stop at 20,000 states because there was not enough memory to test a larger amount of states. I also used observation sequences of varying lengths of integers to test its effect on the runtime of the algorithm. I tested observation lengths of 10, 100, 500, 750, and 900. I was limited to stopping at 900 because higher values would exceed the maximum recursion depth. I generated both matrices A and B to hold probabilities equal to 1.0/(the number of columns) to ensure that each element within each respective matrix was equal. I averaged each entry in the graph based on five separate entries with the same inputs.

4.3 Results

Based on the trends in the graph, it is clear that as the length of the observation sequence and the size of the HMM both increase, the runtime of the Viterbi Algorithm increases as well. As the observation sequence stays static and the size of the HMM increases by a factor of ten, the runtime also tends to increase by a factor of ten. When the size of the HMM goes from 10,000 states to 20,000 states, it changes by a factor of two in all cases. For cases where the length of the observation sequence increases on the same size HMM, the runtime increases linearly which reflects the linear effect of T in the runtime $O(T*N^2)$. The graph for all of the experimental results can be found in the appendix.

5 - Analysis of Results

5.1 Discussion of Results

There is a large range of runtimes based on the varying inputs used for the simulated HMM. The smaller the observation sequence and the smaller the size of the HMM, the faster the Viterbi algorithm will search through a given HMM. When the runtime is low, then voice programming seems to be efficient enough where the user will not have to wait long for their speech to be converted into text as given by the final state sequence found by the Viterbi algorithm. If the criteria of having to wait too long is 1.5 seconds or higher, then the values where voice coding can become inefficient are listed in the format of (Observation length)(Size of HMM): (500)(10,000), (500)(20,000), (750)(10,000), (750)(20,000), (900)(10,000), and (900)(20,000). These values may be irrelevant for voice coding as the observation sequence length is very high. When coding by voice, it seems highly illogical that a user would be speaking phrases that are at least 500 phonemes long as there would be many commands in one utterance to be resolved. Because this is a simulated HMM, its parameters and their sizes should not be treated as if they were going to be used by a speech recognition program. However, the results provide evidence that as an HMM grows, it can reach a point where the Viterbi algorithm will take a long enough time where the user would be waiting for their speech to be converted into text before being able to move on to dictating new commands.

5.2 The Next Step

With the given results, we can see that even though HMMs allow for large sets of hidden states, which store possible command or textual outputs, they cannot be used to store a very large grammar. This means that, in the context of programming, the set of commands that can be added to an HMM is finite. Many systems already account for large grammars which

hold states for all possible language outputs, so the created commands must thus be created carefully so as to not approach the limit of too many stored phrases. For example, commands that are rarely used should be eliminated from a system as they will take up useless space. If the command is necessary but not used frequently, a quick solution would be combining it with another command somehow. This will reduce the size of the HMM while still maintaining the rarely used phrase.

The Viterbi algorithm becomes slow with large inputs because it must check all hidden states from the HMM. This research thus proposes that solutions need to be found to make the Viterbi algorithm even more efficient in order to be able to handle HMMs where N approaches infinity in the best case. This would allow users to create commands arbitrarily without having to worry about the size of their grammars. If there was a way to eliminate the need to check states where probabilities end up being 0.0, a useless step would be removed from the algorithm entirely. The only case where these 0.0 probability values would be important are when all probabilities of all states ended up being 0.0, but then there would be no possible next state. This case would cause an infinite loop so the algorithm would never terminate.

Chapter 6 - Conclusion

6.1 The Future of Speech Recognition

Speech recognition is a relatively new technology, even though it has advanced recently. Currently, speech recognition is at a point where phonemes are still pieced together by probabilities, so there is no way to guarantee a 100% accuracy of speech to text. I believe that 100% accuracy will not be able to be reached in the near future because the issue has to do with speech itself rather than computing power. Speech varies between different cultures and even different people within the same culture, making it difficult to be perfectly accurate for many different types of sounds. Differences in incoming speech are still not accounted for in modern systems. In essentially all languages, there are slang phrases or dialects which may not be able to be programmed as commands. The ability for systems to recognize any language is not available as well. There are still many unresolved issues which could make speech recognition more globally accessible. To even activate most speech recognition programs, you need to use the keyboard and mouse, which would be impossible for those who are unable to use their upper body at all. Those without the ability to use keyboard and mouse would need to be able to use speech recognition before the computer even turns on. There are also no programs which are ready to use instantly after downloading. Speech recognitions require some amount of user customization in order to get them to work within text editors. This turns away those who want a one-click solution to being able to code by voice. Command libraries are usually not straight-forward either, which requires a lot of user training and practice to comfortably use and remember the predetermined abstract language of commands.

With greater computing power, speech recognition has grown over time and could excel to become faster and possibly more accurate. As more data is compiled over time, trends can be found in speech patterns which could boost accuracy of speech recognition systems. The

amount of memory a computer can hold and use at once has risen exponentially. With this growth in memory, speech recognition systems could store more commands into their grammars and quickly parse through larger networks to build up probable phonemes. The most advanced improvement to speech recognition would be a system that could understand and interpret your speech and respond to you based on the given input. There have been some systems implemented which have set responses to certain speech inputs, but they are not as advanced as a system which would exhibit human behavior and interactions by responding to the meaning behind the text. Speech recognition is slowly becoming a part of many aspects of our daily life: accessing menus in automobiles, research into Smart TVs, speech recognition with technologies like Google Glass, automated telephone calls, etc. Until these applications can be finalized and successfully implemented universally, there is still much more to be learned about speech recognition and how it can help benefit our lives and allow us to do things more efficiently.

References

Bailey, Robert. "Human Interaction Speeds." Cool Stuff and UX Resources(Aug. 2000): n. pag. Human Factors International.

Begel, Andrew Brian. "Spoken Language Support for Software Development." Thesis. University of California, Berkeley, 2005.

Bongers, Paulien M., and B. M. Blatter. "Duration of Computer Use and Mouse Use in Relation to Musculoskeletal Disorders of Neck or Upper Limb." International Journal of Industrial Ergonomics 30.4-5 (2002): 295-306.ScienceDirect.

Chatterjee, Shaunak, and Stuart J. Russell. A Temporally Abstracted Viterbi Algorithm. Thesis. University of California, Berkeley, 2012.Cornell University Library.

Cooley, James W., and John W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series." Mathematics of Computation (1965): 297-301. American Mathematical Society.

Desilets, Alain. "VoiceGrip: A Tool for Programming-by-Voice." International Journal of Speech Technology 4.2 (2001): 103-16. Springer Link. Web. 2014.

"Dragonfly 0.6.5 Speech Recognition Extension Library." Python. Python Software Foundation

Ernawan, Ferda, Nur Azman Abu, and Nanna Suryana. "SPECTRUM ANALYSIS OF SPEECH RECOGNITION VIA DISCRETE TCHEBICHEF TRANSFORM." Proc. of International Conference on Graphic and Image Processing. Ed. Yi Xie and Yanjun Zheng. Vol. 8285. 2011.

Formants. Digital image. University of Iowa

Grabianowski, Ed. "How Speech Recognition Works" 10 November 2006. HowStuffWorks.com.

Johnson, Mark, and Stuart Geman. "Probability and Statistics in Computational Linguistics, A Brief Review." Mathematical Foundations of Speech and Language Processing. N.p.: Springer Science & Business Media, 2004. 14.

Joyce, James, "Bayes' Theorem", The Stanford Encyclopedia of Philosophy (Fall 2008 Edition), Edward N. Zalta (ed.), URL = http://plato.stanford.edu/archives/fall2008/entries/bayes-theorem/

Leggett, J., & Williams, G. (1984). An empirical investigation of voice as an input modality for computer programming. International Journal of Man-Machine Studies,21, 493-514

Meyer, Ben. Voicecode. N.p., 2015. Web. https://voicecode.io/#about

Perkoff, Elana Margaret. "The Viterbi Algorithm and Sign Language Recognition." Thesis. Haverford College, 2014.

Petry, Frederick E., and William Pierce. "Input Of Tiny Basic Programs By Voice." Southeastcon '81. Conference Proceedings (1981): 245-47. IEEE Explore Digital Library. Web.

Pinola, Melanie. "Speech Recognition Through the Decades: How We Ended Up With Siri." Web log post. TechHive. IDGTechNetwork, 2 Nov. 2011.

Rabiner, Lawrence R. "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition." Proceedings of the IEEE 77.2 (1989): 257-86. Electrical and Computer Engineering. University of California Santa Barbara.

Ryan, M. S., and G. R. Nudd. "The Viterbi Algorithm." Thesis. University of Warwick, Coventry, CV4 7AL, England, 1993.

Staab, Wayne. Spectrogram. Digital image. Wayne's World. Hearing Health & Technology Matters, 12 Aug. 2014.

Stamp, Mark. "A Revealing Introduction to Hidden Markov Models." Thesis. San Jose State University, 2012.

Zweig, Geoffrey G. "Speech Recognition with Dynamic Bayesian Networks." Thesis. University of California, Berkeley, 1998.

APPENDIX

A1. Runtime Results

Length of Observation Sequence	Number of Hidden States	Runtime of Viterbi Algorithm
10	10	0.001
10	100	0.001
10	1000	0.004
10	10000	0.030
10	20000	0.063
100	10	0.001
100	100	0.004
100	1000	0.041
100	10000	0.320
100	20000	0.643
500	10	0.002
500	100	0.018
500	1000	0.177
500	10000	1.753
500	20000	3.257
750	10	0.003
750	100	0.027
750	1000	0.261
750	10000	2.439
750	20000	4.863
900	10	0.005
900	100	0.031
900	1000	0.306
900	10000	2.914
900	20000	6.371