Spring 2021

A Customizable Speech Practice Application for People Who Stutter

Eric Grimm
egrimm@rollins.edu

Nikola Vuckovic

A Customizable Speech Practice Application for People Who Stutter

Eric Grimm and Nikola Vuckovic
Rollins College

Abstract

Stuttering is a speech impediment that often requires speech therapy to curb the symptoms. In speech therapy, people who stutter (PWS) learn techniques that they can use to improve their fluency. PWS often practice their techniques extensively in order to maintain fluent speech. Many listen to audio recordings to practice where a single word or sentence is played on the recording and then there is a pause, giving the user a chance to say the word(s) to practice. This style of practice is not customizable and is repetitive since the contents do not change. Thus, we have developed an application for iPhone that uses a text-to-speech API to read single words and sentences to PWS, so that they can practice their techniques. Each practice mode is customizable in that the user can choose to practice certain sounds that they struggle with, and the app will respond by choosing words that start with the desired sound or generate sentences that contain several words that start with the desired sound. We generate sentences in two ways: an AI approach using recurrent neural networks and a "fill-in-the-blank" approach. When generating sentences, the goal is that the sentences are relatively short and contain simple words, so that the user does not struggle to repeat the word or sentence.

1 Introduction

Speech is a motor skill that seems simple upon first glance. However, speech is incredibly complex, and it requires that one's brain, lungs, diaphragm, vocal cords, and articulators all work in unison to produce fluent speech. Additionally, it is estimated that the average adult has around 15,000 words in their active vocabulary (Webster & Wohlberg, 1992). To put this in perspective, 15,000 words equates to 15,000 unique muscle movement patterns that we must be able to perform without difficulty to produce fluent speech. Thus, it is not surprising that this process can sometimes malfunction. This malfunction is what results in speech impediments. The speech impediment that we will be focusing on in this paper is stuttering.

Stuttering is a speech impediment that has no cure. Instead, people have been trying to ease the symptoms with therapy for centuries. Through therapy, people who stutter (PWS) learn techniques that they can use to make their speech more fluent. After the invention of computers, more therapy options started to become available. There are devices that provide feedback to the user and there are others that are meant solely for practice. Today, there are several different applications available on the smartphones that offer stuttering therapy. However, there are no apps that allow PWS who have already gone through speech therapy to practice their techniques.

For PWS, practice involves articulating words and sentences while using techniques they learned in therapy. PWS typically need to practice every day, but it can be hard to do so for a lot of people. Many PWS listen to audio tapes to practice. In this style of practice, the user repeats a word or sentence after the speaker on the recording says it. Audio recordings are incredibly useful because PWS can practice their speech in a multitude of situations such as driving or doing chores. There are three problems with this method that could be improved. First, in this method, the recordings become repetitive since the contents of the recording cannot change. Second, most people strongly dislike the sound of their own voice and are not be willing to record practice tapes for themselves. Lastly, audio recordings offer no customization options since the contents of the recording do not change. This is a problem because PWS can often struggle with particular sounds. In the audio recording method of practice, it is difficult to practice specific types of sounds.

We have designed an application for iPhone that fills in this gap and fixes these three major problems. The app works by using the Google Cloud Text-to-Speech API to articulate words for the user. After each utterance, there is a pause to give the user a chance to say the words for themselves and practice their techniques. Since PWS have different techniques for words and sentences, the app allows the user to practice single words or sentences. The app also offers many customization options that include options to practice by sound type or sound class.[1] In the case of practicing single words, the app will only articulate words that align with the user's customization choices. Regarding practicing sentences, the app will randomly generate sentences using two different methods. One is an AI approach using recurrent neural networks and the other is what we call the "fill-in-the-blank" approach. Both approaches allow the sentences to be customizable. It is important that the sentences are not too long and do not contain extremely long words to make the sentences easier for the user to repeat. We believe that our app fixes the three problems of traditional audio recordings since

- words and sentences are randomly generated and thus are not repetitive
- the app relies on a text-to-speech API and not one's own voice
- the app offers customization options for sound type and sound class.

However, our application, in its current state, is a proof of concept. The app currently uses two different methods for generating text. In the production version, the app will only utilize one of these methods. Also, the app has only been tested on an iPhone XS Max and is not optimized for other screen sizes. The production version will be optimized to run on all iPhone screen sizes.

2 Stuttering and Speech Therapy

2.1 Overview of Stuttering

Stuttering is speech characterized by hesitations, repetitions, harsh sounds, elongated sounds, and a disruption of the natural rhythm or flow of speech. It is estimated that around 1% of the population stutters (Van Borsel, Maes, & Foulon, 2001). The disorder affects males much more than females with a ratio of about 3:1 (Yairi & Ambrose, 2013). Also, the disorder tends to arise during childhood with most people growing out of it (Radonjić et al., 2020). Not everyone shows the same degree of severity. In fact, many people who stutter only have trouble with particular sounds. These sounds often belong to the same class of sounds.

There are four different sound classes, which are shown in Table 1. This classification system is derived from the Precision Fluency Shaping Program developed by Webster &

[1] We will define this terminology in the next section.

Wohlberg (1992). It is apparent that there are some letters that are missing from the table. For example, Q and X are not present. This is because these letters produce sounds that are found elsewhere on the table. The Q letter produces the "kw" sound in English, and thus is grouped into the plosive sound K/C. Likewise, in English, words beginning with the sound X produce a Z sound are grouped with Z. Another quirk with this classification system is that a word of a certain sound type may not start with that same letter. For example, words beginning in PH belong to the F sound type.[2] All these exceptions are outlined in more detail in Table 2. Also, in stuttering, we are primarily concerned with the first sound of a word. Going forward, we will only be considering the initial sound of a word when assigning a word to a particular sound and sound class.

Table 1
The Four Sound Classes of Speech

Sound Class	Sounds
1. Vowels	A, E, I, O, U
2. Voiced Continuants	J, L, M, N, R, V, W, Y, Z, voiced TH
3. Fricatives	F, S, H, SH, CH, unvoiced TH
4. Plosives	B, P, D, T, G, K/C

Note. This table lists the sounds that are associated with each sound class in the English language.

PWS have a variety of techniques that they may use to achieve fluent speech. Some of these techniques are specific to certain types of sounds and others are specific to word chains. Thus, PWS must practice single words and sentences. Our application is designed to accommodate both needs. There are options to practice sentences and single words, and both modes offer customization options since PWS often struggle with certain sound types or sound classes. Our application is one of many different forms of computer-aided speech therapy, which are described in further detail in the next subsection.

[2] The terms sound type and individual sound are used interchangeably.

Table 2

Rules for the Sound Types

Starting Letter(s)	Exceptions
C	- CE, CI, CY words belong to sound type S - CZ words belong to sound type Z - most CH words belong to sound type CH - all remaining words belong to sound type K/C
G	- GE and GY belong to sound type J
K	- KN words belong to sound type N
P	- PH words belong to sound type F - PS words belong to sound type S - PT words belong to sound type T
Q	- all words belong to sound type K/C
S	- SH words belong to sound type SH
T	- TH words belong to unvoiced TH or voiced TH - TZ words belong to sound type Z
W	- WH words belong to sound type H - WR words belong to sound type R
X	- all words belong to sound type Z

Note. This table lists all the words beginning with a letter that does not match the letter of its sound type in the English language.

2.2 Computer-aided Speech Therapy

Since the advent of computers and modern technology, the area of speech therapy has seen significant advances. Speech language pathologists began to use computer enabled devices to aid in speech therapy. Since computers have continued to become more and more prevalent, this area of speech therapy has continued to grow as well. Today, it is more prevalent than ever since so many of us carry a smartphone everywhere we go. There are many different apps available on the App Store and the Google Play store that offer speech therapy resources. Here we provide an overview of computer-aided speech therapy for stuttering.

The Precision Fluency Shaping Program by Webster & Wohlberg (1992) uses a device called a voice monitor to aid in therapy for stuttering. The voice monitor is used to analyze the initial vibrations of the vocal folds when the user speaks. The device will alert the user if their onset was correct or incorrect. This device is one that provides meaningful feedback to the user, which the user can use to adjust their speech patterns. However, there are some devices that do not provide any analysis and are meant solely for practice or to improve fluency. The voice monitor is not meant for everyday use. Instead, it is supposed to be used for intensive speech therapy and never used again, ideally.

Gordon W. Blood (1995) at Pennsylvania State University conducted a study using a computer-aided device for speech therapy. Four adults participated in the study and all saw a significant reduction in the number of stuttered syllables. All participants used a Computer-Aided Fluency Establishment Trainer (CAFET). The CAFET "uses a microcomputer, circuit board, respiratory sensor, and a clip-type microphone" (Blood, 1995, p. 166). The respiratory sensor is used to measure airflow and the microphone is used to record the user's speech. The results are shown on a monitor. In the study, Blood was trying to assess and improve diaphragmatic breathing, continuous airflow, pre-voice exhalation, easy onset, initial prolongation, continuous phonation, phrasing, and monitored speech. He had the patients practice one target at a time and would not move on to the next one until the current one was fully mastered. The CAFET was used to provide feedback to the patients. For example, to practice diaphragmatic breathing, the CAFET displayed a breathing curve using the respiratory sensor. Using this information, the patients were able to practice breathing in a way that would produce a favorable curve on the monitor.

During the next decade, researchers developed more types of computer-aided therapy devices. Ooi Chia Ai and J. Yunus (2006) outline the four types of computer-based stuttering therapy devices. These devices are

- devices that alter auditory feedback (AAF)
- devices that provide feedback on physiological status or production patterns
- devices that alter speech motor production patterns

- pacing/metronome devices.

The most notable AAF device is the delayed auditory feedback (DAF) device. This device repeats what the user says with a 0.25 second delay. The other types of AAF are masked auditory feedback, frequency altered feedback (FAF), and combined feedback. The devices that provide feedback on physiological status or production patterns, as the name implies, “provide immediate feedback of voice onset patterns, duration, and amplitude/loudness” (Ai & Yunus, 2006, p. 208). Clinicians can use these devices to show patients what correct speech patterns look like on the device and then leave them to practice on their own. Some examples of these devices are the FluencyNet and the Digital Speech Aid. The Fluency Master is an example of a device that alters speech motor production patterns. The device resembles a hearing aid, and it functions by altering the way that the user hears their own speech. It allows them to hear their natural vocal tone because normally, when we speak, the voice that we hear is different than the voice that others hear. This happens because vibrations from the vocal folds must travel through bone, cartilage, and soft tissue before reaching the ear. Being able to hear one’s true voice makes it easier to control stuttering. The last device, the metronome, is straightforward. The device plays an audible click in a rhythmic fashion since speaking to a rhythm has been shown to improve stuttering. Ai and Yunus go on to elaborate on the current state of computer-based therapy in Malaysia.

Unger, Glück, and Cholewa (2012) provide a deeper analysis on AAF devices. They confirm that a DAF device plays back the user’s voice with a slight delay. This delay is adjustable. The preferred delay used to be 250 ms. However, researchers have shown that the delay can be reduced to 50 ms with similar a similar reduction in the number of stuttered syllables. The authors also posit that a FAF devices plays back the speaker’s voice at a different pitch. The research on whether this type of AAF device improves stuttering has been inconclusive so far. This specific study used a device that used DAF and FAF.[3] The researchers found that there was a statistically significant reduction in the number of stuttered syllables when using the device.

[3] According to Ooi Chia Ai and J. Yunus, this device would be considered a combined feedback device.

3 Methods of Text Generation

3.1 Markov Text Generation

There are a few different ways to generate text in a software environment. One of the simpler ones and the first one we will be examining is utilizing Markov chains. A Markov chain is a stochastic process that consists of a finite set of states. Each state has a constant conditional probability to transition to a different state; it is possible for that probability to be zero. An important feature of Markov chains is the memoryless property, meaning that the transition probability depends just on the previous state. Formally, a discrete-time stochastic process is a Markov chain if, for t = 0, 1, 2, . . . and all states,

$$P(X_{t+1}=i_{t+1} \mid X_t=i_t, X_{t-1}=i_{t-1}, \ldots, X_1=i, X_0=i_0)=P(X_{t+1}=i_{t+1} \mid X_t=i_t)$$

(Winston, 2004, p. 181). To expand on this definition, we will introduce the idea of the discrete-time stochastic process. If we observe a system where X_t represents a characteristic of the system at time t where t = 0, 1, 2, . . . , then a discrete-time stochastic process would be "a description of the relation between the random variables $X_0, X_1, X_2, \ldots$" (Winston, 2004, p. 180).

To generate text with Markov chains we must establish these finite states and transitional probabilities. We can make a choice between generation via characters or words. This will determine our states. If we choose character generation, we will have states that represent the last *n* characters while in word generation our states are the last *n* words. For each state created, we can determine the probability for the next word by studying the source material. This gives us a probability distribution that can then be used to generate new text.

One of the first people to introduce text generation with a stochastic process like Markov chains was the American mathematician, Dr. Claude Shannon (1948), in his paper "A Mathematical Theory of Communication." Shannon argues that every letter in the English language can be assigned a probability depending on the frequency that the letter appears in the seed text (Shannon, 1948). The seed text represents the material that we are using as the base for generating new text. This means that the seed text must be sufficient in length to provide relevant probabilities. Shannon introduces four orders of natural language approximation, which are shown in Table 3 (Shannon, 1948, p. 7) along with his results. From the results we can clearly see how the generated text improves by increasing the order of approximation. We can also notice the word level generation and how it improves when transition probabilities are included.

The increased performance of word level generation in Markov chains translates into AI text generation, which we will discuss further.

Table 3

Orders of Approximation and the Results

Order of Approximation	Results
Zero-order approximation: symbols are independent and equiprobable	XFOML RXKHRJFFJUJ ZLPWCFWKCYJ FFJEYVKCQSGHYD QPAAMKBZAACIBZLHJQD
First-order approximation: symbols are independent but frequencies of English text are included	OCRO HLI RGWR NMIELWIS EU LL NBNESEBYA TH EEI ALHENHTTPA OOBTTVA NAH BRL
Second-order approximation: digram, transition probabilities $p(i, j) = p(i)p_i(j)$, structure as in English	ON IE ANTSOUTINYS ARE T INCTORE ST BE S DEAMY ACHIN D ILONASIVE TUCOOWE AT TEASONARE FUSO TIZIN ANDY TOBE SEACE CTISBE.
Third-order approximation: trigram, transition probabilities $p(i, j, k)$, structure as in English	IN NO IST LAT WHEY CRATICT FROURE BIRS GROCID PONDENOME OF DEMONSTURES OF THE REPTAGIN IS REGOACTIONA OF CRE.
First-order word approximation: words are chosen independently but with their appropriate frequencies	REPRESENTING AND SPEEDILY IS AN GOOD APT OR COME CAN DIFFERENT NATURAL HERE HE THE A IN CAME THE TOOF TO EXPERT GRAY COME TO FURNISHES THE LINE MESSAGE HAD BE THESE.
Second-order word approximation: the word transition probabilities are correct but no further structure is included.	THE HEAD AND IN FRONTAL ATTACK ON AN ENGLISH WRITER THAT THE CHARACTER OF THIS POINT IS THEREFORE ANOTHER METHOD FOR THE LETTERS THAT THE TIME OF WHO EVER TOLD THE PROBLEM FOR AN UNEXPECTED.

Note. This table lists the categorization of orders of approximation introduced by Claude Shannon. It also presents the example results from Shannon’s paper for each approximation category.

3.2 “Fill-in-the-blank” Style

The fill-in-the-blank style is based on Mad Libs, which is a word game where the player fills in blanks in sentences with a random word. However, each blank is supposed to be filled in with a certain type of word. These words can be nouns, verbs, numbers, colors, places, etc. After the player fills in all the blanks, they read the sentences and the result is normally very comical. This concept can be used to make a simple text generation model. We can create template sentences with blank words where each blank is represented by a symbol that corresponds to a certain word type. For example, the template sentence “The 1 is a little 3” has two placeholders, 1 and 3. 1 is a placeholder for a noun and 3 is a placeholder for an adjective.[4] Template sentences can be fed into a program that fills in each placeholder with a random word from a list of words that match the placeholder type. Hossain et al. (2017) use this concept to generate sentences as well. However, instead of using a completely random word to fill in each placeholder, they choose a word that is likely to be funny given the context of the sentence.

3.3 Recurrent Neural Networks

To implement artificial intelligence in text generation, we will use a type of neural network called recurrent neural networks (RNN). RNN is a type of neural network that is used to model sequential data. RNNs consist of three layers: input, hidden, and output. In the hidden layer we can find a self-loop that is used to access outputs from previous states, or its “memory,” to generate new output and predictions. This is particularly important for text generation since we can use multiple sequential data points, these being characters or words, to predict the next data point. Ilya Sutskever introduces RNNs as follows:

> Given a sequence of input vectors ($x_1, \ldots, x_T$), the RNN computes a sequence of hidden states ($h_1, \ldots, h_T$) and a sequence of outputs ($o_1, \ldots, o_T$) by iterating the following equations for t = 1 to T:
>
> $$h_t = tanh(W_{hx}x_t + W_{hh}h_{t-1} + b_h)$$
>
> $$o_t = W_{oh}h_t + b_o$$
>
> In the equations, W_{hx} is the input-to-hidden weight matrix, W_{hh} is the hidden-to-hidden (or recurrent) weight matrix, W_{oh} is the hidden-to-output weight matrix, and the vectors

[4] This is according to Table 4 found in section 4.3.

> b_h and b_o are the biases. The undefined expression $W_{hh}h_{t-1}$ at time t = 1 is replaced with a special initial bias vector, h_{init}, and the tanh nonlinearity is applied coordinate-wise. (Sutskever et al., 2011, p. 2)

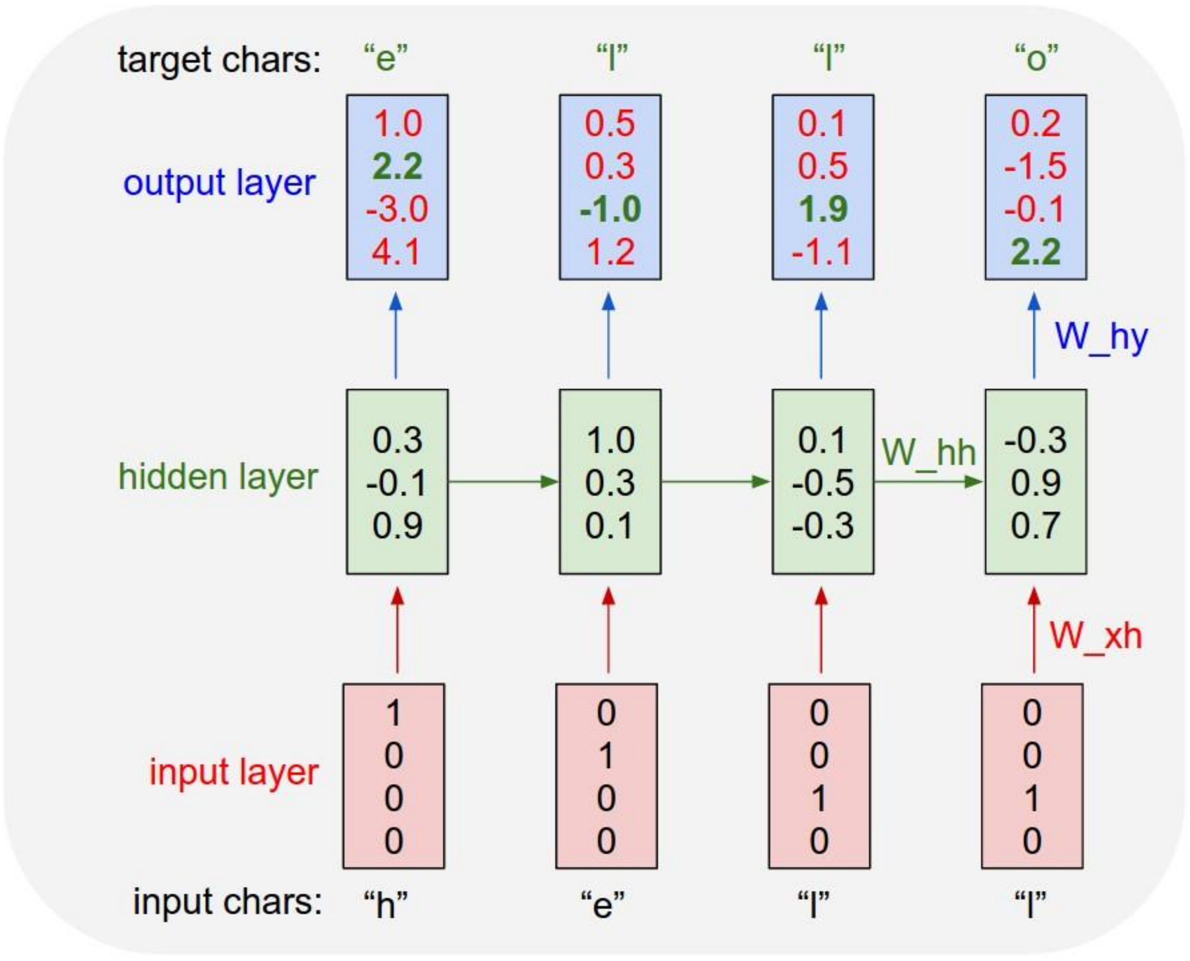
Their paper demonstrates the power of RNNs when they are trained with the new Hessian-Free optimizer by applying them to character-level language modeling tasks. RNN based models are showing excellent performance and present the future of AI text generation. Further work on RNNs and the performance overview of these models can be found in papers by Lu (2018) and Mikolov (2010).

Alex Graves (2014) introduces how Long Short-term Memory (LSTM) recurrent neural networks can be used for generation of long-range structured sentences by only predicting one data point at a time. LSTM RNNs use purpose-built memory cells to store information; hence, they are better at storing and accessing data. He defines RNNs as dynamic models that "can be trained for sequence generation by processing real data sequences one step at a time and predicting what comes next" (Graves, 2014, p. 1). Graves further explains that RNNs will iteratively sample the networks' output distribution and then use that sample for input in the next step. This allows the network to generate novel text sequences. Additionally, Graves presents the challenges that come from generating with regular RNNs and how LSTM RNNs can be used to obtain better results. However, some of the most significant work on character-level text generation via LSTM RNNs was done by Andrej Karpathy. He created a model, char-rnn, that takes a large source text and uses it to determine the next character depending on the previous sequence of characters (Karpathy 2015). A graphical representation of this model is shown in Figure 1 (Karpathy 2015).

The model we will be using in our application, textgenrnn, was created by Max Woolf (2020), and it is a direct product of Karpathy's work on char-rnn. Textgenrnn is a Python package that utilizes the TensorFlow and Keras Python frameworks and makes significant improvements to char-rnn. These improvements include attention-weighted averaging, decaying learning rate, and character embeddings (Woolf 2018). The default texgenrnn model still uses character-level generation, and its implementation is displayed in Figure 2 (Wolf 2018). However, in our model, we will be utilizing word-level training. More information on this method will be provided in the next section.

Figure 1

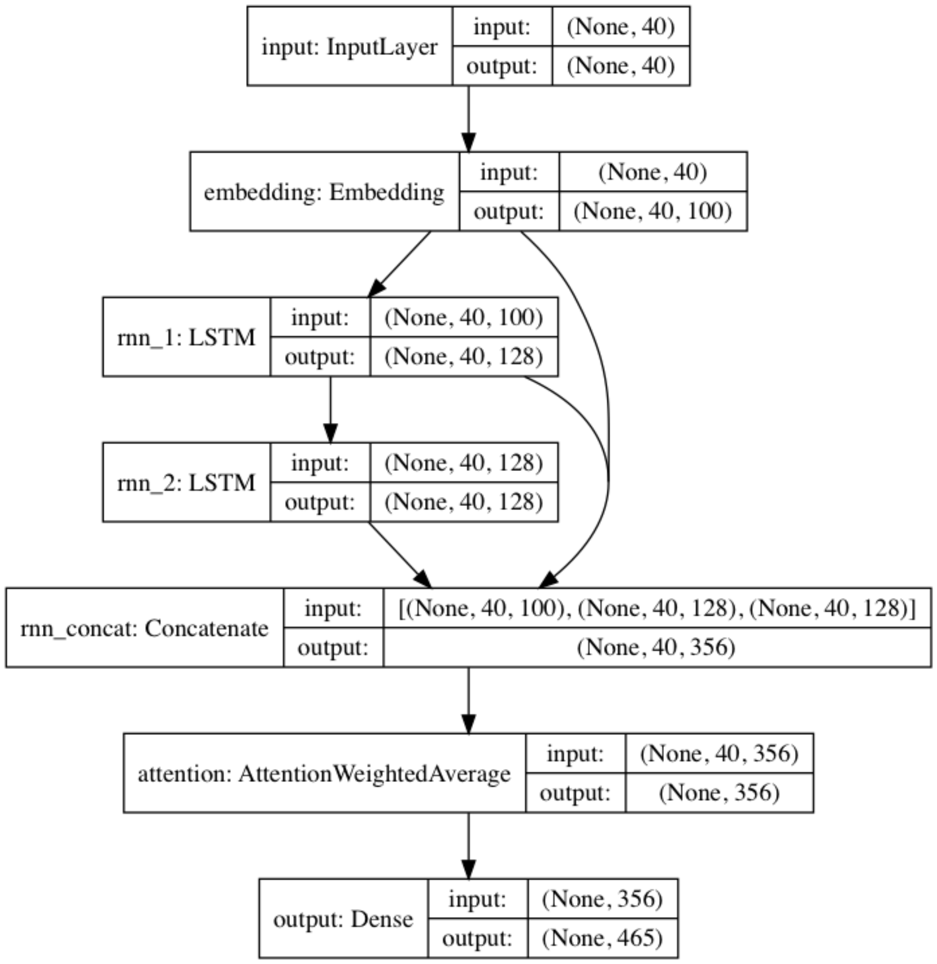
Example of Char-rnn Implementation



Note. The figure represents a forward pass through the char-rnn model when the model is provided with characters "hell" as input. Each character is assigned a vector using 1-of-k encoding (i.e., all zero except for one at the index of the character in the vocabulary). Then in the hidden layer, the vectors are fed into the RNN with the step function (the code is available in the source material). The step function then generates the output vector, with one dimension per character, which represents the confidence that the RNN currently assigns to each character coming next in the sequence. The process is then repeated until the confidence values converge to values that will predict the correct character every time.

Figure 2

Overview of Textgenrnn Implementation



Note. The default implementation will take input of up to 40 characters. It will then go through the embedding layer, followed by two LSTM layers. The next step takes it to the attention layer, where the most critical temporal features are weighted and averaged together. Finally, the model produces the output mapped to probabilities for up to 394 different characters.

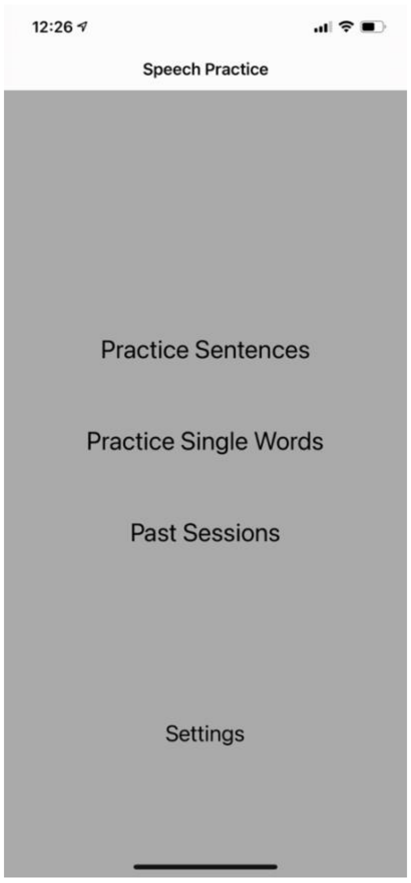
4 Description of the System

4.1 Overview of the System

We developed the Speech Practice application using the Swift programming language through Apple's XCode integrated development environment (IDE) with the intention of deploying the app on iPhone. The application has two main features: practicing single words and practicing sentences. When the app is launched, the user can choose the type of practice session that they want. The functional options on the home screen are: Practice Sentences and Practice Single Words shown in Figure 3. Tapping these icons will either lead to the sentences menu or the words menu. Both menus offer different customization options which we will discuss further in the following sections.

Figure 3

Speech Practice App Home Screen



Note. This is the screen that first opens when the app loads. The user can tap on either Practice Sentences or Practice Single Words. Past Sessions and Settings are not completely functional yet.

All the different practice options allow the user to input their customizations and choose the number of words or sentences that they want to practice. When the user has entered everything, they can start a session. For every practice option, a word or sentence will be selected. Once the string of text is chosen, the string is converted into speech using the Google Cloud Text-to-Speech API. Once the iPhone articulates the text, there is a pause to allow the user to say the word back.[5] The length of the pause depends on the type of practice session that the

[5] There is no analysis happening at this stage. The app is simply acting as a tool to help people practice their fluency. The app is there to provide random words and sentences that adhere to some rules specified by the user.

user is in. Sentences have a longer pause since they take longer to say back while single words have a shorter pause.

The user can customize a practice session based on the four different sound classes mentioned earlier. Each practice mode (sentences and single words) includes two main options: Practice by Sound Classes and Practice by Individual Sounds. In the former, the user can choose one or multiple sound classes to practice. In the latter, the user can choose individual sounds to practice. The implementation of these features for the two different modes is explained in further detail in the next subsections.

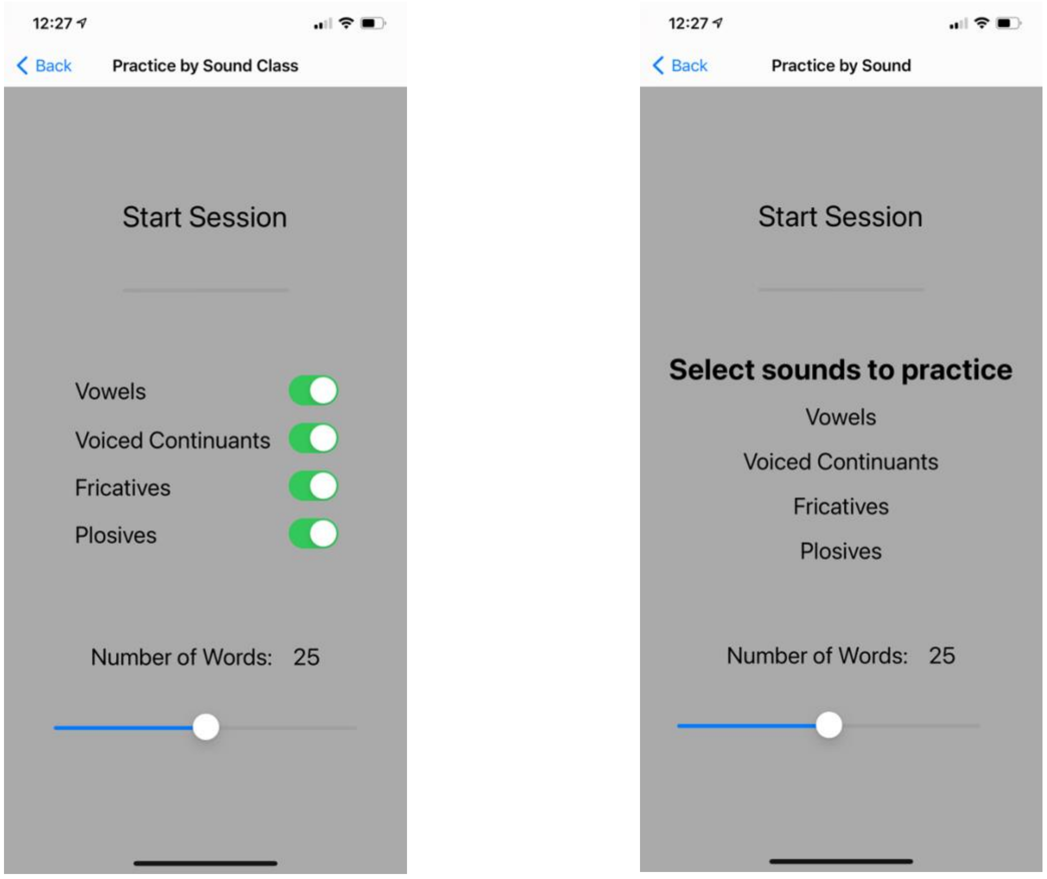
4.2 Practice by Single Words

The Practice by Single Words mode has two options: Practice by Sound Classes and Practice by Individual Sounds. The user can select one of these two options after selecting Practice Single Words on the home page of the application. Both options rely on the same word database, which is derived from the Moby Project (Downey, 2016). We divided the master list into four smaller lists based on the sound class of the initial sound in each word. Then, we divided each sub-list into lists based on each specific initial sound rather than the sound class. This means that the app can easily find a word matching a specified initial sound or sound class. The app will choose a word at random from the list.

After selecting Practice by Sound Classes, the user is brought to a screen where they can enter their customizations and start the session, shown in Figure 4. The user will be presented with four switches. When a switch is on, the words read to the user will have initial sounds pertaining to the chosen sound class. The user can select more than one sound class. For example, if the user chooses Vowels and Plosives, then the user will hear words with initial sounds A, E, I, O, U, B, P, D, T, G, and C/K. If all switches are on, then the user will get a completely random word meaning that the word could have any initial sound. A progress bar updates after each word is articulated. This gives the user an idea of where they are in the session. All practice options within the app have a progress bar.

Figure 4

Practice by Sound Classes and Practice by Individual Sounds for Single Words

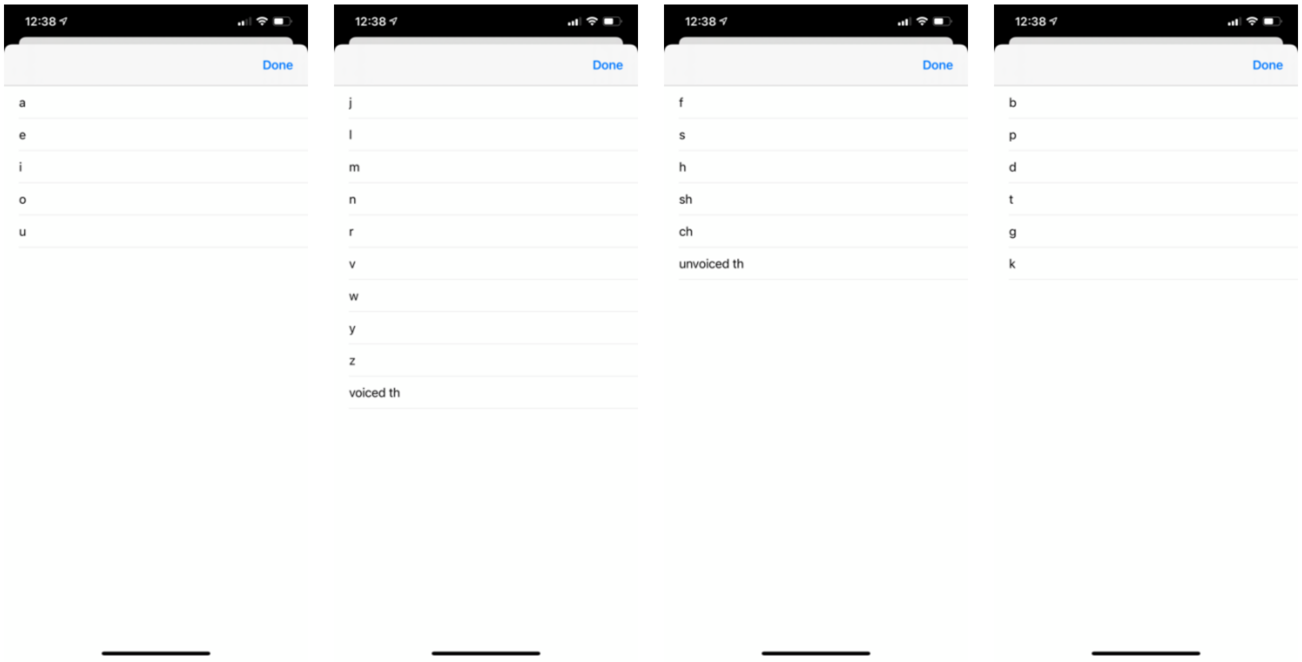


Note. The left image is the screen for the Practice by Sound Classes option for words. The right image is the screen for the Practice by Individual Sounds option for words.

When the user selects the Practice by Individual Sounds option, a screen similar to the screen found in the Practice by Sound Classes option will appear. This is also shown in Figure 4. Instead of four switches, there are four buttons that correspond to the four different sound classes. These buttons allow the user to select sounds that they want to practice. When a button is pressed, a selection menu will appear, all are shown in Figure 5. Each selection menu lists all the sounds that are in the corresponding sound class. The user can select and deselect sounds from multiple lists. Once the user has chosen all the sounds that they want to hear, they can choose the length of the session and start. The user will hear words that only have initial sounds matching the sounds that they chose. For example, if the user chose A from vowels, M and N from voiced continuants, and S from fricatives, then words spoken in the session will have an initial sound of A, M, N, or S.

Figure 5

The Four Selection Menus in the Practice by Individual Sounds Option



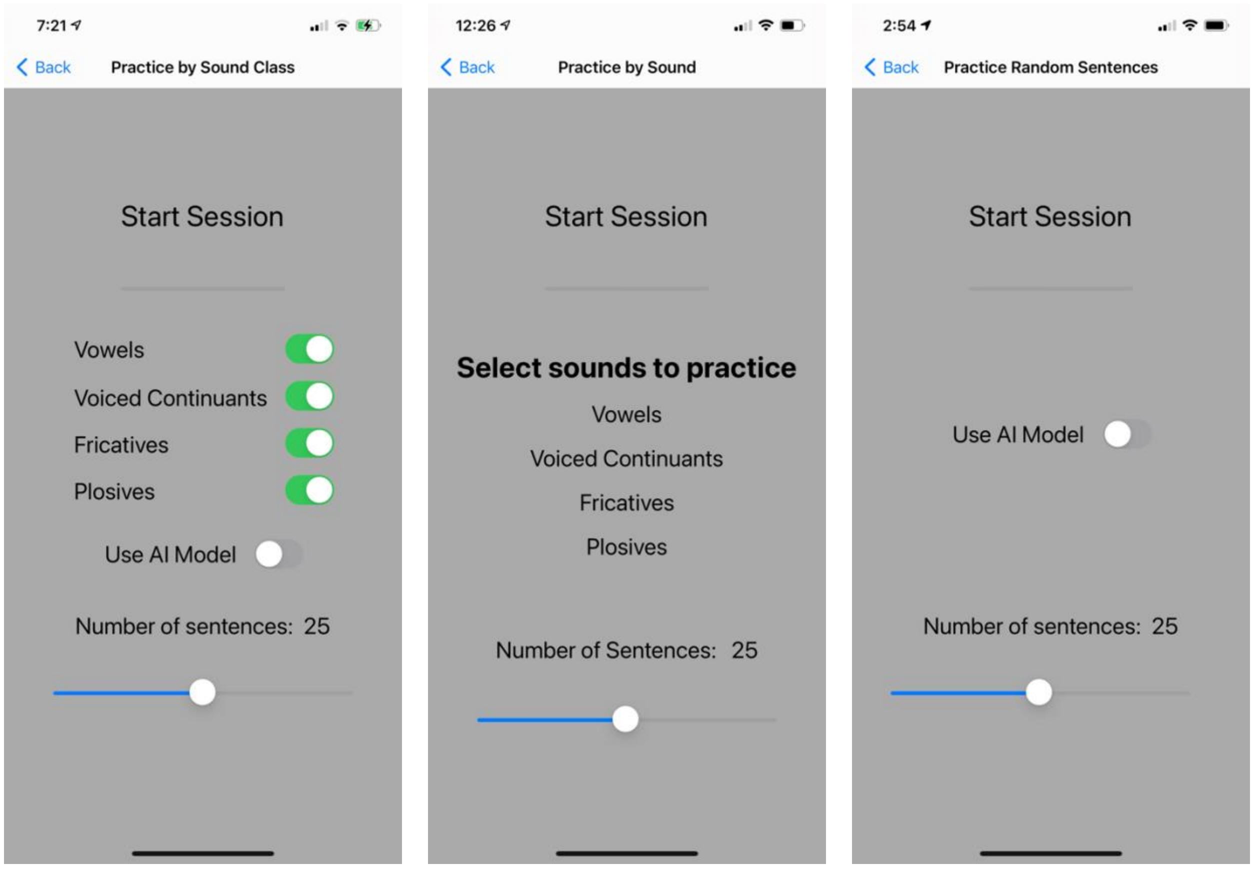
Note. Each screen contains all the individual sounds for a specific sound class. The left screen contains class 1 sounds, the left middle screen contains class 2 sounds, the right middle screen contains class 3 sounds, and the right screen contains class 4 sounds.

4.3 Practice by Sentences

The Practice by Sentences mode has three options: Practice by Sound Classes, Practice by Individual Sounds, and Practice Random Sentences. The user interface for the first two options is virtually identical to those found in the Practice by Single Words mode except for the addition of the switch to toggle between the two different approaches for generating sentences. This is shown in Figure 6. However, the underlying code is very different since we are generating sentences instead of words. The third option, Practice Random Sentences, has a simple user interface since the purpose of this mode is just to produce random sentences without customization. This mode only contains a slider to adjust the length of the session, the start session button, a progress bar to show the user where they are in the practice session, and the toggle for switching between the fill-in-the-blank approach and the AI approach, which is also displayed in Figure 6. First, we will give an overview of the fill-in-the-blank approach and then we will explain the AI approach.

Figure 6

Practice by Sound Classes, Practice by Individual Sounds, and Practice Random Sentences for Sentences



Vowels [✓ on]
Voiced Continuants [✓ on]
Fricatives [✓ on]
Plosives [✓ on]
Use AI Model [off]
Number of sentences: 25

Select sounds to practice
Vowels
Voiced Continuants
Fricatives
Plosives
Number of Sentences: 25

Use AI Model [off]
Number of sentences: 25

Note. The left image is the screen for the Practice by Sound Classes option for sentences. The middle image is the screen for the Practice by Individual Sounds option for sentences. The right image is the screen for the Practice Random Sentences option.

Fill-in-the-blank Approach

The first way that we can generate sentences is by using the fill-in-the-blank approach. In order to use this approach, we must have template sentences and word lists of different word types. In our case, we have five different types of words: noun, verb, adjective, adverb, and name of a person (Fellbaum, 1998; Roche). Each template sentence has some predefined words and placeholders. There are five different placeholders that correspond to the five different types of words. These pairings are shown in Table 4. Each list of words is sorted by sound class and turned into smaller lists that are sorted by initial sound. This is very similar to the sorting method used in section 3.2.

Table 4

The Five Placeholders for the Fill-in-the-blank Approach

Placeholder	Word Type
1	Noun
2	Verb
3	Adjective
4	Adverb
5	Name of a Person

Note. This table lists the placeholders for each word type. Each placeholder is a number between 1 and 5. When a sentence is constructed, each placeholder is filled in with the corresponding word type. For example, the “1” placeholder is always filled in with a noun.

When the user starts a practice session, the application will pull a random sentence from the list of template sentences when it needs a sentence. The app checks each string in the sentences and determines if the string is a word or a placeholder. If the word is a placeholder, then it is replaced by the matching word type as shown in Table 4. After every string is checked, the sentence is rebuilt with the placeholders having been swapped out for actual words. The sentence is then read to the user and the process repeats itself. The manner in which the words are chosen varies between the three practice options within Practice by Sentences.

Practice Random Sentences is the least complex out of the three since it does not involve any user customization. When a placeholder string is found, a random word is chosen from the master list of the corresponding word type. This means that the chosen word could have any type of initial sound and belong to any of the four sound classes. As a result, it is likely that the words added to the sentence will not be of the same initial sound type or class. This ensures that the sentences feel random to the user.

In Practice by Sound Classes, the user specifies what sound class(es) they want to practice. This means that when a placeholder is changed out for a word, that word could be of any sound type from the chosen sound class(es). When a placeholder string is found, a random sound type is chosen, and then a word from the corresponding sound class is put into the sentence. This means that it is unlikely that all the new words will be of the same sound type. If the user chooses to practice multiple classes at once, then the sentence will contain words from any of the chosen classes.

In Practice by Individual Sounds, the user can specify the sounds that they want to have appear in the sentence. The selection screen is similar to the one in Figure 5.[6] However, this practice option works a bit differently in that every placeholder will end up being a word from the same sound type. When the application needs a sentence, a random initial sound is chosen from the list specified by the user. The chosen initial sound is the one that will be used to fill in all the placeholders. For example, if the chosen initial sound is R, then every word that is added to the sentence will have the initial sound of R. Thus, each individual sentence allows the user to practice one particular sound. This is different from Practice Random Sentences and Practice by Sound Classes where the placeholders are filled in with any type of sound.

AI Approach

The second way that we can generate sentences is through an AI approach based on recurrent neural networks. More specifically, we are using a Python package called textgenrnn developed by Max Woolf (2018). One of the advantages of using textgenrnn is access to training through Google Colaboratory, a product by Google Research that allows users to write and execute Python code in the form of Colab notebooks in their browser. Google Colaboratory also provides free access to GPU training, which significantly reduces training times compared to CPU-based training.

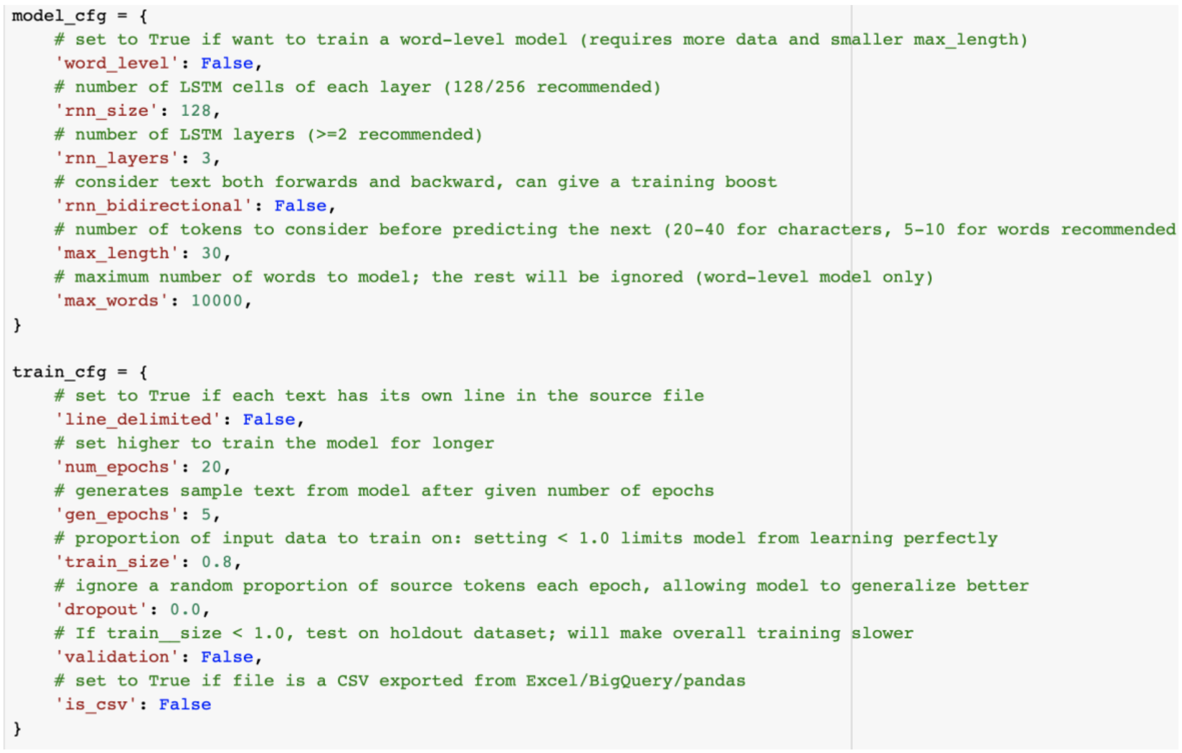
At the beginning of our project, we followed the Woolf's Colab notebook's primary training method, displayed in Figure 7. This method included generating text on a character level from long continuous texts such as Shakespearean plays or famous novels. However, this method of training did not perform as desired. The generated text was not natural sounding and did not have the desired effect in speech practice. The ineffectiveness was due to multiple different factors. Most of the open-source files that were large enough to be a seed text were older novels and plays. Hence there was an issue with the vocabulary used not being recent enough. Additionally, due to the seed text format, the text generated was grouped in paragraphs, most of

[6] In this practice option, the selection screen for class 2 does not include voiced TH. This is because there are only about 50 voiced TH words in the English language. This makes generating a sentence that includes several instances of those words extremely difficult.

which included a significant amount of dialog. This presented an issue since the focus of our app is primarily on individual sentences.

Figure 7

Model and Training Configuration for the Basic Model



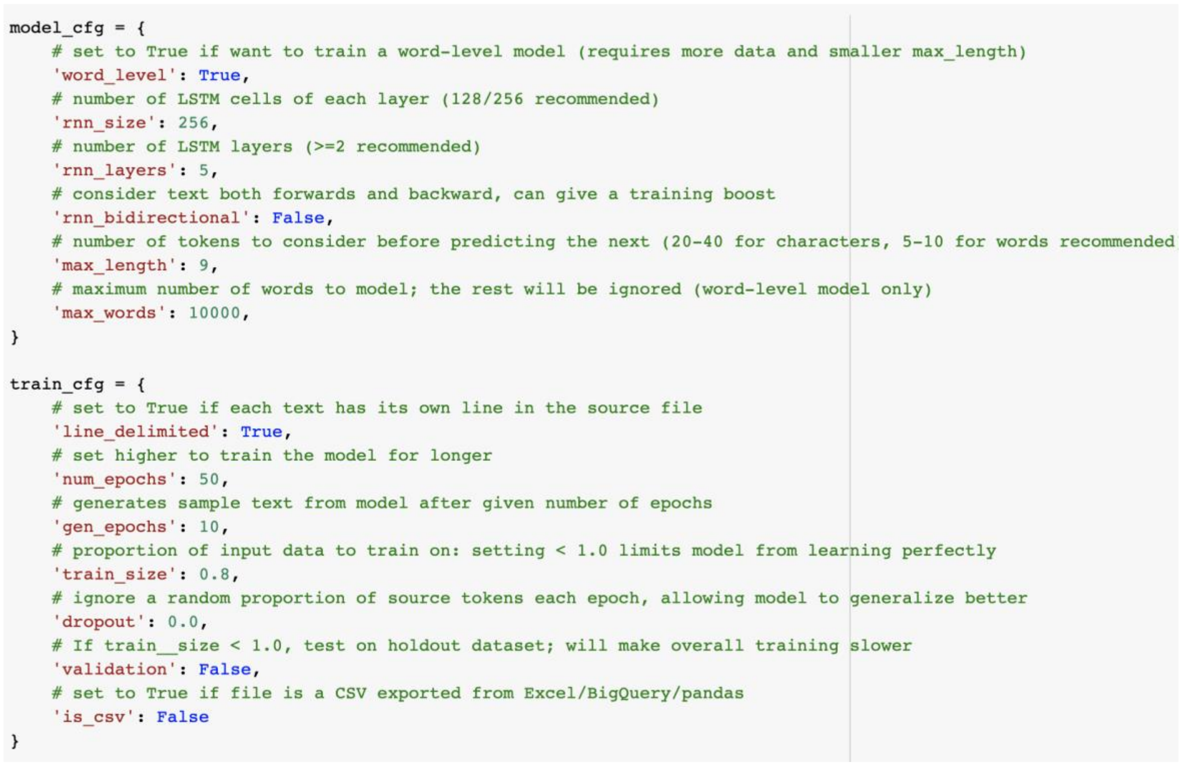
```
model_cfg = {
    # set to True if want to train a word-level model (requires more data and smaller max_length)
    'word_level': False,
    # number of LSTM cells of each layer (128/256 recommended)
    'rnn_size': 128,
    # number of LSTM layers (>=2 recommended)
    'rnn_layers': 3,
    # consider text both forwards and backward, can give a training boost
    'rnn_bidirectional': False,
    # number of tokens to consider before predicting the next (20-40 for characters, 5-10 for words recommended
    'max_length': 30,
    # maximum number of words to model; the rest will be ignored (word-level model only)
    'max_words': 10000,
}

train_cfg = {
    # set to True if each text has its own line in the source file
    'line_delimited': False,
    # set higher to train the model for longer
    'num_epochs': 20,
    # generates sample text from model after given number of epochs
    'gen_epochs': 5,
    # proportion of input data to train on: setting < 1.0 limits model from learning perfectly
    'train_size': 0.8,
    # ignore a random proportion of source tokens each epoch, allowing model to generalize better
    'dropout': 0.0,
    # If train__size < 1.0, test on holdout dataset; will make overall training slower
    'validation': False,
    # set to True if file is a CSV exported from Excel/BigQuery/pandas
    'is_csv': False
}
```

Note. In the basic model, word_level configuration is set to False, signifying character_level generation. The training is set to 20 epochs, and the model is set to rnn_size of 128 with three rnn_layers. Training time is approximately 40 minutes.

Due to issues presented in the previous paragraph, we resorted to the other training model available with textgenrnn, which includes word-level training. This means that the model considers a set number of previous words to predict a new one. This results in a model that trains much quicker. We experienced training times of under 30 minutes, even when training on 50 epochs with an rnn_size of 256 and 5 rnn_leyers. The lower training times allowed us to train the model better and to produce a higher number of text-generating models contributing to the diversity of the text. Another benefit of training on the word level is that it mostly eliminates spelling errors, which we experienced when training on the character level. Woolf points out issues with punctuation when training on word level; however, this was not an issue in our case. The model produces a file with a large number of individual sentences. The sentences are later read by the Google Cloud Text-to-Speech API, not displayed to the user, removing the need for perfect punctuation in the sentences. The exact model and training configuration we used in the final training is shown in Figure 8.

Figure 8

Model and Training Configuration for the Actual Model used in the App



```
model_cfg = {
    # set to True if want to train a word-level model (requires more data and smaller max_length)
    'word_level': True,
    # number of LSTM cells of each layer (128/256 recommended)
    'rnn_size': 256,
    # number of LSTM layers (>=2 recommended)
    'rnn_layers': 5,
    # consider text both forwards and backward, can give a training boost
    'rnn_bidirectional': False,
    # number of tokens to consider before predicting the next (20-40 for characters, 5-10 for words recommended
    'max_length': 9,
    # maximum number of words to model; the rest will be ignored (word-level model only)
    'max_words': 10000,
}

train_cfg = {
    # set to True if each text has its own line in the source file
    'line_delimited': True,
    # set higher to train the model for longer
    'num_epochs': 50,
    # generates sample text from model after given number of epochs
    'gen_epochs': 10,
    # proportion of input data to train on: setting < 1.0 limits model from learning perfectly
    'train_size': 0.8,
    # ignore a random proportion of source tokens each epoch, allowing model to generalize better
    'dropout': 0.0,
    # If train__size < 1.0, test on holdout dataset; will make overall training slower
    'validation': False,
    # set to True if file is a CSV exported from Excel/BigQuery/pandas
    'is_csv': False
}
```

Note. In this model, word_level configuration is set to True, signifying word_level generation. The training is set to 50 epochs, and the model is set to rnn_size of 256 with 5 rnn_layers. Training time is approximately 30 minutes.

More improvements followed from changing the seed text. We realized that we needed a modern text to generate relevant sentences. Woolf (2018) discusses using data from Reddit to generate text. His example output looked close to what we wanted to establish. He provides a script to retrieve the required data. To do so, we must utilize Google Cloud BigQuery. There we can adjust the subreddit headlines and adjust the time period we want to retrieve. The final output is a .csv file with an average size of 2 MB that combines two subreddits. However, not all subreddits work well together. For example, /r/sports and /r/askhistorians did not produce satisfying output, but combinations of /r/askhistorians and /r/science or /r/news and /r/technology did.

Now we had a trained model that would generate files with thousands of satisfying sentences. Hence, the next step was to integrate it into the app. The Colab notebook allows us to download a Python script and required weights to generate text on our computers. However, textgenrnn is currently only a Python package, so it does not work natively in Swift. Furthermore, there is no efficient way to run Python3 scripts in Swift for iOS development. This meant that we needed to find a way to deliver new text to the app without running the actual model in the app. Thus, we decided to deploy a web server that can be used to retrieve sentences.

The web server comes in the form of a Python Flask app. The Flask app reads multiple text files that the model from the Colab notebook generated. The text is initially read into a list with the condition that the sentence is less than 12 words. The sentences under 10 are preferred for speech practicing purposes. However, after testing, we determined that the limit of 12 gives us the optimal output. The list of sentences is later converted into a set to eliminate any duplicate sentences. In my testing, we would have around 50 sentences with a duplicate in 1000 sentences. Sentences are then stored in class 1, class 2, class 3, and class 4 lists depending on the class criteria. Each sentence must have at least three instances of that sound class's words. It is possible for a sentence to be in more than one category. Sound classes are then converted into an HTML message and returned to the page with the class-specific path.

The Flask app is then deployed via Heroku, a cloud application platform. When the app is deployed via Heroku, it receives a unique URL, and it can be accessed via that URL by any machine on the web. This means that we can send an HTTP request from our iOS app to our Flask web server. The interface for the AI approach is shown in Figure 4 in the left and right images. In order to generate sentences using this approach, the user has to flip the "Use AI Model" switch to on. If the user starts a session with the AI model enabled, then the app will send an HTTP request to the Heroku server. The server will respond with a list of sentences that match the user's request. If the user is practicing random sentences, then the list of sentences that the server returns will be purely random sentences. On the other hand, if the user is practicing by sound class, the list of sentences that the server returns will contain sentences that each have multiple instances of words from the chosen sound class. The app will choose one of the sentences at random to read to the user.

5 Measures of Random Sentence Quality

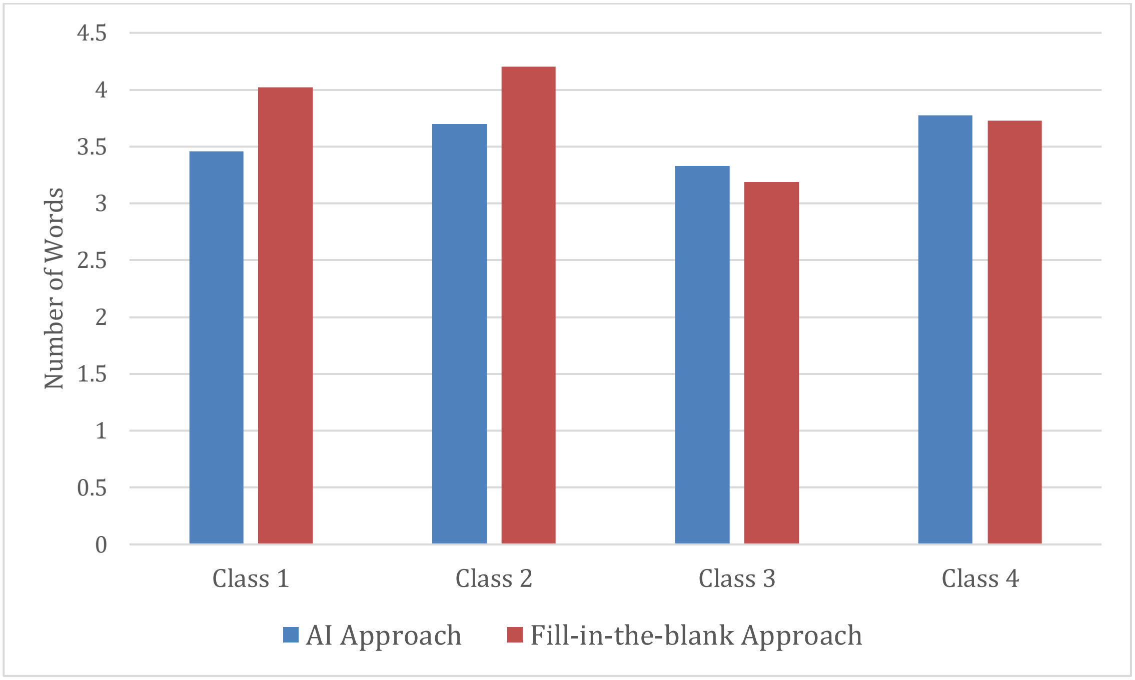
We used three different measurements to evaluate the quality of the sentences that are generated by the two different approaches. These measurements are the number of desired words per sentence, sentence length, and sentence complexity. The number of desired words per sentence is the average number of words in a sentence that satisfy the sound class requirement given by the user. Sentence length is the average length of a sentence, and we define sentence complexity to be the average length of a word in a sentence. It is desirable that the sentences are not overly complex and that the sentences are not too long since it may be difficult for the user to repeat long and wordy sentences. We will discuss each measurement in further detail in the following subsections.

5.1 Desired Words Per Sentence

In order to conduct this measurement, we generated lists of class 1, class 2, class 3, and class 4 sentences using both the fill-in-the-blank approach and the AI approach. This left us with eight different lists to analyze. It is important to note that sentences may contain words that are from different sound classes. For example, a class 3 sentence will likely only have a few class 3 words with the rest of the words being from different classes. We are interested in counting the average number of words in the sound class of interest. To do this, we used a Python script to count the number of words in each sentence that met the given sound class requirement, and then divided by the total number of sentences in the list to obtain the average number of desired words per sentence. We used this procedure for all eight lists. Our results are displayed in Figure 5. Ideally, we want a typical sentence to have multiple occurrences of words that meet the sound class requirement. Our results indicate that all practice options for each text generation approach have an average number of desired words per sentence greater than 3. This indicates that both methods of generating text are producing sentences that meet the requests of the user on average.

Figure 9

Average Number of Words from Desired Sound Class Per Sentence



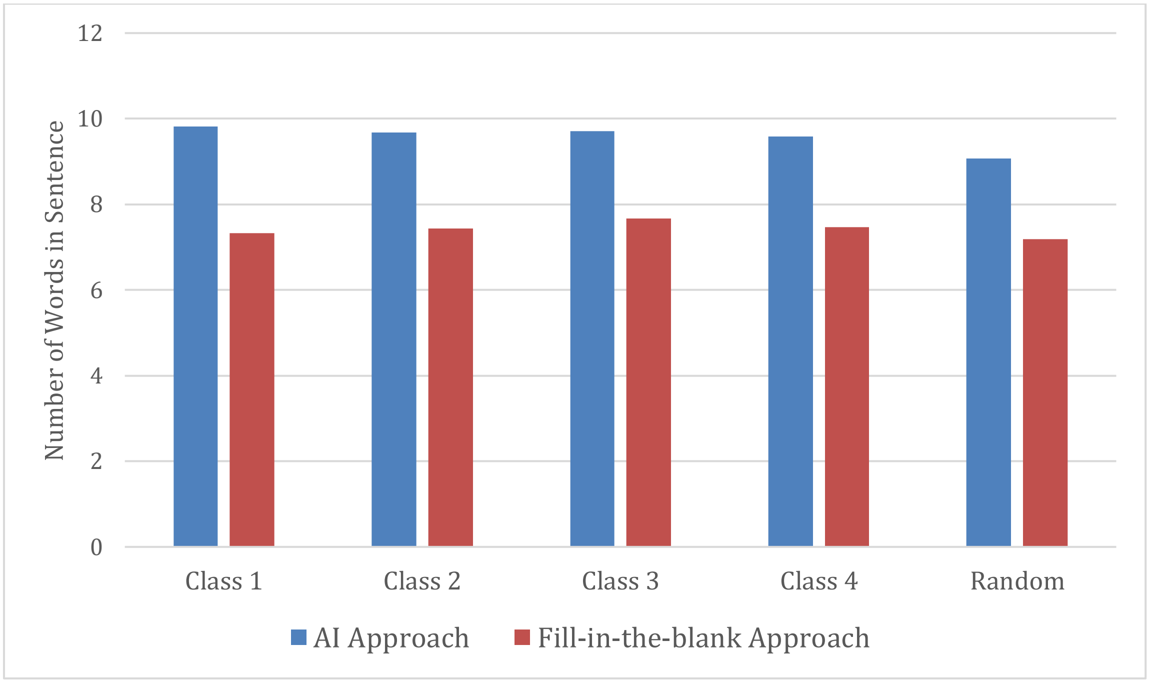
Note. This figure shows the average number of desired words per sentence for each of the four sound classes using the two different sentence generation approaches. All categories have an average number of words per sentence greater than 3, which indicates that a typical sentence is likely to have multiple occurrences of words that are of the sound class chosen by the user.

5.2 Sentence Length

The process of obtaining this measurement was very similar to the one in the previous subsection. However, since we are only measuring the average length of a sentence, we included lists of random sentences generated by each approach. The results can be found in Figure 10. Ideally, we want the length of each sentence to be less than 10 words. If a sentence is greater than 10 words, then it will become difficult for the user to repeat the sentence. Our results indicate that all instances of the fill-in-the-blank approach typically meet this requirement since the average sentence length is less than 8 words for all of them. This result makes sense since all fill-in-the-blank sentences are generated from template sentences, which means that we can control the length of the sentences. On the other hand, all instances of the AI approach average just under 10 words per sentence. This means that there are a lot of sentences that are greater than 10 words. However, to produce natural sounding sentences using the AI approach, we ended up using sentences that were less than 12 words long with most being close to 11 words long. When we lowered the sentence length further, most sentences that we generated did not sound lifelike. However, since the average sentence is less than 10 words long in all cases, a typical generated sentence will meet the ideal length requirement.

Figure 10

Average Length of Sentence



Note. This figure shows the average length of a sentence generated by each approach for all sound classes and random sentences. All categories have an average sentence length under 10 words.

5.3 Sentence Complexity

To measure sentence complexity, we generated two large lists of sentences. One list was generated using the fill-in-the-blank approach and the other was generated using the AI approach. To measure sentence complexity, we calculated the average length of each word in both lists. We did not include single letter words such as “a” or “I” because we felt that they may skew our results. We found that the average length of a word using the fill-in-the-blank approach was 4.15 characters long and 4.29 characters long using the AI approach. This means that a typical word in a sentence has about 4 letters in it, and that most words in a sentence are relatively short. Thus, most sentences will not be overly complex.

5.4 AI Approach vs. Fill-in-the-blank Approach

The sentences generated by the AI approach and the fill-in-the-blank approach are drastically different. The fill-in-the-blank approach may offer more consistent sentences, but the sentences are very basic. On the other hand, the AI approach produces more diverse sentences. Also, the length of each AI sentence is unpredictable, but the length of each fill-in-the-blank sentence is well controlled since those sentences are based on template sentences. The length of each sentence is vital to the quality of a practice session for the user. If the user keeps getting long, complex sentences, this will take away from the practice session since the user will be more focused on saying the sentence quickly instead of focusing on practicing their techniques. However, the fact that the fill-in-the-blank approach is based on template sentences is an issue since the user may become aware of patterns in the sentences.

We believe that this problem can be alleviated if we include a large number of template sentences in the app. If we can incorporate more template sentences and if we cannot improve the quality of the AI sentences, then we believe that we should use the fill-in-the-blank approach in the production version of the app. In the fill-in-the-blank approach, we can easily control sentence length, complexity, and number of desired words. This style of sentence does not offer as much variety as the ones in the AI approach; however, we do not feel that this is a major issue. The point of the app is to provide natural sounding sentences that are customizable for practicing speech. It is ideal that the sentences feel random to the user, but this is not completely necessary. It is not a major problem, if the user notices patterns in the sentences since it should not take away from the practicing experience.

6 Limitations

In the app’s current state, it offers two methods of generating sentences: the AI method and the fill-in-the-blank method. The fill-in-the-blank method is compatible with all three practice options: Practice by Sound Classes, Practice by Individual Sounds, and Practice Random Sentences. The AI method only works with Practice by Sound Classes and Practice Random Sentences. The app does not provide any analysis to the user, and it is not a substitute for professional speech therapy. The point of the app is to help PWS practice their techniques from therapy in an effective, intuitive, and customizable way. However, the app, in its current state, does have several limitations.

First, the app was primarily tested on an iPhone XS Max, which is the screen size that the app works best on. We attempted to run the app on an iPhone 8 Plus, but it did not work. Thus, there is no guarantee that the app will function properly on different iPhone screen sizes. We were unable to develop for other iPhones since we are using a third-party library called RSSelectionMenu to control our multi-selection menus. This library only runs on actual devices and not simulators. Thus, we are not able to simulate the app on different screen sizes, which hinders our ability to optimize the app for other screen sizes.

Next, there are multiple limitations with the AI approach to generating text. The most prominent one is the seed text. We used data retrieved from Reddit to train our model. It was in the right format, provided modern and relevant vocabulary, and was very accessible and not subject to copyright. However, this data is not well regulated, especially some of the subreddits we used such as /r/news and /r/technology. They contain certain words and phrases that are either inappropriate or could be considered offensive by some users. Since we are working with tens of thousands of reddit headlines, we did not develop a way to block this content from the generated text.

Lastly, the output of the trained AI model is a Python script that imports selected weights and temperatures and generates text accordingly. However, we are not able to run Python code in an iOS application as of now, which presents another major limitation of the AI model. We had to resort to retrieving AI generated sentences from a webserver. This worked fine for our testing purposes, but will not translate well into the production version of the app. The server can be easily overwhelmed by a large number of requests.

7 Conclusion and Future Work

7.1 Conclusion

In this paper, we discussed our application for PWS. This application is different from others on the market in that it allows PWS who have already gone through speech therapy to practice their techniques. It also offers customizable practice options by sound class and sound type. In our application we include two primary modes, which are Practice by Single Words and Practice by Sentences. In Practice by Single Words, the user can either practice by sound class or sound type. During a practice session, the user will only hear words that begin with the sound type(s) that they specified. In Practice by Sentences, the user can also practice by sound class or by sound type. Additionally, the user can choose to practice random sentences that do not adhere to any rules.

To generate sentences, we use a fill-in-the-blank approach and an AI approach. In the fill-in-the-blank approach, each sentence is generated from one of the template sentences included in the app. Each sentence has multiple placeholders where a word can be filled in. If the user chose to practice random sentences, then any word can replace a placeholder. However, if the user is practicing by sound class or sound type, then any word that is substituted for a placeholder must start with one of the sound types that the user specified.

In the AI approach, we retrieve Reddit data via BigQuery to create seed texts. That text is then used to train a textgenrnn model in a Colab notebook with word-level training. The trained model generates text files with individual sentences in each line. The text files are then read by a Python Flask app and filtered to create class-specific lists. Each list will contain sentences with less than 12 words in length and at least three instances of class words. The lists are further returned to class-specific paths, and the app is deployed via Heroku. The iOS app can then modify the URL and send a request to receive the list of sentences. The app will then choose one sentence at random every time a sentence needs to be read aloud to the user.

7.2 Future Work

In its current state, the app is not fully optimized for all iPhone screen sizes. The app was primarily tested on an iPhone XS Max, which is the screen size that it works best on. In the future, we hope to optimize the layout of the user interface so that it runs well on all iPhones. Additionally, we would like to develop a version of the app using SwiftUI, which is Apple's new development interface. We programmed the current version of the app using Interface Builder (or Storyboards), which Apple is hoping to move away from at some point. Transitioning to SwiftUI would help futureproof the app and may resolve the screen size issue that we discussed earlier. This will also give us an opportunity to improve the user interface since the app currently relies on an extremely basic user interface. After making these changes, we hope to publish this app to the Apple App Store, so that PWS may begin using it to practice their speech.

In the production version of the app, we should rely on only one of our two approaches to generating text. There are a lot of changes that we need to make to the AI approach if we choose to integrate that method in the production version. To do this, we need to further improve the seed text and optimize the training configurations. This will also involve trying out more temperature combinations. If we want to solely rely on the AI model for text generation, this means that the AI model would have to be used to generate sentences for Practice by Individual Sounds as well since it does not in the current state of the app. Also, we would like to find a way to deploy the AI model directly in the app. The current implementation is not efficient since the app must communicate with a server to get the AI generated sentences. Lastly, since we are hoping to publish this app on the App Store, we will need to filter out offensive words from the word lists and generated sentences. On the other hand, the fill-in-the-blank is approach is close to being production ready. The only major changes that need to be made are adding more template sentences and removing inappropriate words from the word lists. The fill-in-the-blank approach may prove to be a more viable option for a production-level application.

References

Ai, O. C., & Yunus, J. (2006). Overview of a Computer-based Stuttering Therapy. In *Regional Postgraduate Conference on Engineering and Science* (pp. 207–211).

Blood, G. W. (1995). A behavioral-cognitive therapy program for adults who stutter: Computers and counseling. *Journal of Communication Disorders*, *28*(2), 165–180. https://doi.org/10.1016/0021-9924(95)00008-2

Downey, A. (2016). *Think Python* (2nd ed.). Sebastopol: O'Reilly Media.

Fellbaum, C. (1998). *WordNet: An Electronic Lexical Database*. MIT Press.

Graves, A. (2013). Generating Sequences With Recurrent Neural Networks. *ArXiv, abs/1308.0850,* 1–43. Retrieved from http://arxiv.org/abs/1308.0850

Hossain, N., Krumm, J., Vanderwende, L., Horvitz, E., & Kautz, H. (2017). Filling the blanks (hint: Plural noun) for Mad Libs® humor. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing* (pp. 638–647). Copenhagen. https://doi.org/10.18653/v1/d17-1067

Karpathy, A. (2015). The Unreasonable Effectiveness of Recurrent Neural Networks. Retrieved from https://karpathy.github.io/2015/05/21/rnn-effectiveness/

Lu, S., Zhu, Y., Zhang, W., Wang, J., & Yu, Y. (2018). Neural text generation: Past, present and beyond. *ArXiv, abs/1803.07133*.

Mikolov, T., Karafiát, M., Burget, L., Jan, C., & Khudanpur, S. (2010). Recurrent neural network based language model. In *Proceedings of the 11th Annual Conference of the International Speech Communication Association, INTERSPEECH 2010* (pp. 1045–1048).

Radonjić, I., Davidovića, N. D., Radulović, D., Otašević, J., Šoster, D., & Davidović, D. (2020). Characteristics of Adult People with Fluency Disorder. *Human: Journal for Interdisciplinary Studies*, *10*(1), 11–21. https://doi.org/10.21554/hrr.042002

Roche, D. S. (n.d.). names.txt. Retrieved from https://www.usna.edu/Users/cs/roche/courses/s15si335/proj1/files.php%3Ff=names.txt&downloadcode=yes

Shannon, C. E. (1948). A Mathematical Theory of Communication. *Bell System Technical Journal*, *27*(4), 379–423. https://doi.org/10.1002/j.1538-7305.1948.tb00917.x

Sutskever, I., Martens, J., & Hinton, G. (2011). Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning, ICML 2011*.

Unger, J. P., Glück, C. W., & Cholewa, J. (2012). Immediate effects of AAF devices on the characteristics of stuttering: A clinical analysis. *Journal of Fluency Disorders*, *37*(2), 122–134. https://doi.org/10.1016/j.jfludis.2012.02.001

Van Borsel, J., Maes, E., & Foulon, S. (2001). Stuttering and bilingualism: A review. *Journal of Fluency Disorders*, *26*(3), 179–205. https://doi.org/10.1016/S0094-730X(01)00098-5

Webster, R. L., & Wohlberg, C. S. (1992). *Precision Fluency Shaping Program: Speech Reconstruction for Stutterers*. The Hollins Communications Research Institute.

Winston, W. L. (2004). *Introduction to Probability Models* (4th ed.). Brooks/Cole.

Woolf, M. (2018). How to Quickly Train a Text-Generating Neural Network for Free. Retrieved from https://minimaxir.com/2018/05/text-neural-networks/

Woolf, M. (2020). textgenrnn. Retrieved from https://github.com/minimaxir/textgenrnn

Yairi, E., & Ambrose, N. (2013). Epidemiology of stuttering: 21st century advances. *Journal of Fluency Disorders*, *38*(2), 66–87. https://doi.org/10.1016/j.jfludis.2012.11.002

Acknowledgements

We would first like to thank our advisor, Dr. Myers, for helping us throughout the whole process of developing this application. We also thank Dr. Summet and Dr. Yellen for serving as our thesis committee alongside Dr. Myers. Lastly, we would also like to thank Matthew Trautmann, Nathaly Espinosa Teran, and Alexander Wehr for their help in developing the initial use cases for the app and for aiding in the development of the first working prototype.