CABbAGE

An AI Generated Pink Noise Approach to Improve Sleep

Emelia Blankenship Ike Lyons Jacob Shomstein Nicole Gizzo
Sam Cohen Sen Francis

December 7th, 2020

Abstract

The CABbAGE project aims to improve sleep though using a AI generated pink noise, variable frequency noise generated to promote better sleep. Given the novel research surrounding sleep and white/pink noise. As a type of pink noise, through personal findings the project aims to experiment with podcast-like noise and compare it sleep-performance of standard white and pink noise. To generate the audio, CABbAGE uses a range of generative techniques like LSTM and Markov Models trained on a variety of seed data ranging from Science and Tech, Zen and Nature and Sports.

1 Motivation

The ultimate motivation for this application came from a desire for better sleep and, by extension, a better way to fall asleep. The first step was to look at the different things people use to fall asleep. Two commonly used media are audiobooks and podcasts. We noticed that though these are indeed very helpful for falling asleep, they weren't specifically designed for it. This niche that was missing is what CABbAGE was born from. We realized that there was no reason to risk losing your place in a podcast if we could generate the podcast audio on the fly. This method would allow the user to select the "sound" of audio that they want to fall asleep to, without having to find an actual podcast episode to fit it. This application allows a user to gain the proven benefit of falling asleep to a white noise, without the downsides associated with using podcasts or audiobooks that aren't designed for sleep.

2 Speech Text Generative Approaches

Given the lack of certainty around which model could produce text, and in turn speech, that would promote sleep the best. We opted to implement more than models that would perform text generation on our training set. This would show us the drawbacks and potential upsides of choosing one model over another.

For the two types of models, we decided to use a Markov Model approach along with an LSTM Deep Learning approach.

2.1 Training Data

All of our podcast training data was found on https://www.listennotes.com/. We sourced our training data from podcasts ESPN Daily and Wildlife Health Connections, and also various essays that were written by Paul Graham. We intended to have podcasts themes for sports, nature, and science and technology, which we sourced from the previously mentioned materials, respectively. The essays were already in the desired format, so all we had to do was copy them into one text file. For the podcast training data, we simply input each podcast into the Watson speech to text interface, and then a text file was generated containing the podcast transcript.

2.2 Markov Model

The first model we used for text generation was a Markov Model. This approach is named after Russian mathemetician Andrey Markov, and it

uses a chain of stochastic probabilities to generate text. Following tokenization, words are generated by sampling from their dependent probability distributions, given by the expression $P(x_t|x_{t-n}, x_{t-n+1}...x_{t-1})$. This means that a word's probability of being selected next is dependent on the

$$n$$

previously generated words. For our model, we selected $n = 2$ as that would permit for a reasonable degree of context-specific language while still maintaining a large enough variety of options for text generation that isn't repetitive.

A larger value of n would provide a more context-aware output but the precision of the posterior probability distributions would be quite low. Additionally, the Markov Model approach can frequently generate an input sequence that did not occur in the training data. In that instance, the model progressively shortens the window until an existing sequence is found.

2.3 LSTM Apprach

The second model that we used for text generation was an LSTM approach (Long Short Term Memory) where we had an LSTM layer with 64 neurons. By using a type of a recurring neural network, we were able to input a sequence of inputs as data. The inputs that we used for the model were normalized indices of word tokens from the corpus. We chose the last 10 tokens as the input sequence and the output vector was a softmax layer.

For training, we created sequences of the past 10 tokens and for the label or target, we used the following word. This way, the model would be able to avoid loss through training by matching what the following word would be given the previous 10 tokens.

To run the training, validation, and generation, we used a Machine Learning API Keras to simplify the work in describing the model. This allowed us to experiment and test quickly without rewriting gradient descent.

To prevent over-fitting we also added a dropout layer with a relatively high weight that provided some protection.

The following Keras code was used to represent the neural network.

```
Sequential([
    LSTM(
        64,
        input_shape=(
            ready_input_data.shape[1],
            ready_input_data.shape[2])),
    Dropout(0.10),
    Dense(
        ready_target_data.shape[1],
        activation='softmax')
])
```

2.3.1 Word Tokenization

One issue we faced was in generating text was the model only creating stop words, the most common words in natural language. To avoid the model from continuously selecting stop words, we abstracted tokens to not only be words but to be joined with common stop words like "to", "a", ... that would otherwise, lead to text with just "to to to to to to".

In doing so we improved the model's performance.

2.3.2 Limited Training Set

Due to time and computing constraints, we only had a relatively small size of training data for the channels of text. This has lead to repetitive generated text. To avoid repetitiveness, we opted to select not the argmax of the resulting output vector as the following word, but rather a random choice between the 10/20/30/40/50 highest activated neurons of the output vector. This has lead to a more diverse output but started to degrade the grammatical validity. We settled on 30 as this felt like a decent trade-off between grammatical correctness and randomization.

2.3.3 Text Generation

Text generation is done through taking a seed of tokens of length l, The same l as used in training. Then, the model predicts the next word, shifts the sequence forward by a word, then uses that trailing seed that now has one predicted word. As this process continues, the model will then be

predicting words based on other predicted words leading to text-generation of any arbitrary length.

3 Use of Cognitive Technologies

IBM Watson technologies played an integral role in this product. The technologies we used were the Speech to Text and the Text to Speech resources. These resources made training the models much easier. The Text to Speech service provides APIs that use IBM's speech-synthesis capabilities that synthesize text into natural-sounding speech. This feature was vital to our product because we wanted our audio to sound as natural as possible rather than the built-in services which sound robotic. The IBM Watson Speech to Text service provides APIs that use IBM's speech-recognition capabilities to produce transcripts of spoken audio. This service allowed us to even transcribe speech from various languages and audio formats.

In order to use the Text to Speech and Speech to Text services in python scripts, we had to first install the IBM Watson package and then authenticate ourselves using the API key and URL. Once the API connection was established, calls could then be made to either the synthesize function. The synthesize function could be used for either Speech to Text or Text to Speech. The synthesize function for Speech to Text was used by importing podcast audio into the model as plain text. The synthesize function for Text to Speech was used to generate a flac or MP4 file given a text file.

4 Application

The application was integral to providing a useable interface to the audio. The first bit of consideration that went into this was deciding what frontend framework was going to be used by the group. We wanted a few things out of our framework: easy development, cross-platform compatibility, and support for different packages. We also had to consider if this was going to be a web application or a desktop app. In the end, we went with ElectronJS as it fit all of our needs and allowed us to build a desktop app using web technologies. After deciding on our framework, we started designing the app.

For the majority of the app, we valued simplicity above all else. We went with Pico CSS as a CSS framework, so we could spend minimal time working on styling while still getting a good looking UI. We decided that showing the minimal amount of choices to the user (selecting one of three "channels" and the amount of minutes to generate) would allow for the greatest clarity and ease of use. After submitting these values the application then serves you the audio where, once again, we chose simplicity by simply showing an audio player with the and nothing else.

In terms of how the app functions, we separated out most of the machine learning from the app, to speed up running times and to keep complexity low. In order to do this we pre-generated text from each of the three podcast styles and then select a random starting place from those to generate our audio. For the actual audio generation, we spawn a python process that generates a .flac file using the Watson text-to-speech API and serves that to the app to be listened to.

Overall, in our application we valued simplicity. This allowed us to develop quickly given the time constraints of the project. The simple design also will allow for additions to be made in the future without having to wade through a lot of complex code. This design choice allowed us to more evenly spread our time to different parts of the project.

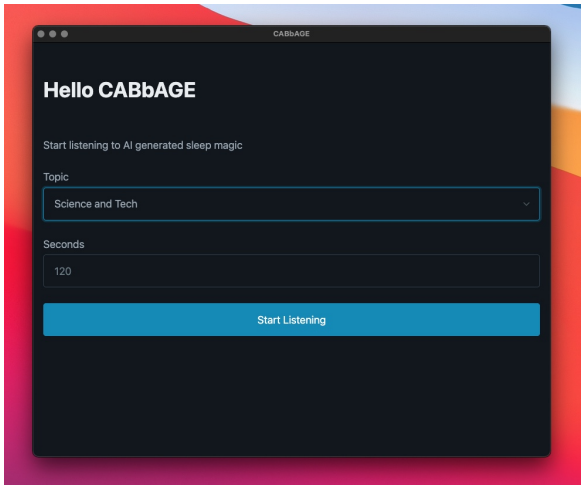

Figure 1: Image of the ElectronJS App

4.1 ElectronJS

To generate the application, we chose to build a native application that would possible to download and run on a variety of machines. This would make it possible for us to not have a constraint on a host machine's operating system due to the way that ElectronJS is essentially an encapsulated web view. In addition to the interoperability benefit, ElectronJS allowed our team to use a widely used language that allowed us to quickly build an app in the limited time frame.

5 Problems not Solved

Given the limited time scope of the project, 3 weeks was not enough time to collect and train the models on enough data to create perfect text. Through we see the promise, the text often feels like it is missing grammatical marks that could be detracting from the goal of creating a fluid sounding speech to go to sleep to.

Additionally, we did not parametrize or customize the type of speech output. This lead to very robotic-sounding speech that is more difficult to go to sleep to. Though the speech had human-like vocal inflections, the robotic sound alike to the grammatical issues detracted from the end goal.

These problems could potentially be solved by the longer processing time and with more training data. We did not have a strong machine learning fleet of machines to train the models on, so we did the best we could with our personal machines. This problem was further expanded since we were using the CPU version of Tensorflow that does not support massive parallelism with the GPU version that would vastly speed up training and generation times leading to higher-quality text, hence speech output.

Lastly, the system architecture used for deployment lacked the Model API that would serve the model output on a separate server. To simplify the workload, we generated the text-generation from the three types of output and randomized the start points to provide essentially a stub of the API. If we chose to deploy this application, a more service-based deployment approach would be necessary to allow for independence of changes between the Electron application and the two models.

6 Future Work

Essentially, the future work would entail building on the issues of the current implantation through creating a stronger infrastructure, changing Tensorflow from CPU to GPU based along, providing more training data and training the model for more epochs.

There are however a few interesting areas that we did not have a chance to explore, one of which was implementing a custom loss function that would factor in the English grammar. This would entail appending to the loss function grammar issues like breaking in the common Subject Verb Object form of sentences, run-on sentences, and misuse of conjunctions. In theory, this would skew the output to be more grammar-focused than only trying to match the following word from the previous subsequence of words.

In addition to improving the general correctness of the text-generation, we will want to work on experimenting with the types of voices and the different variations of speech inflections or styles. This would fall into the experimental and validation aspects of the project that we did not have a chance to fully go through. This would be composed of running a standardized trial with the comparison of REM and NREM sleep between the various possibilities. Since we are using white noise as a baseline, we would use that as the control and compare the performance between standard pink noise generators, and our various channels, lengths, and voices of speech output.

This would allow us to show that our approach would create a statistically significant change in sleep that would prompt people to experiment with our technology.

7 References

- https://www.webmd.com/sleep-disorders/pink-noise-sleep
- https://www.sleepfoundation.org/bedroom-environment/white-noise
- http://xpo6.com/list-of-english-stop-words/
- https://ashutoshtripathi.com/2020/04/06/guide-to-tokenization-lemmatization-stop-words-and-phrase-matching-using-spacy/
- https://base64.guru/converter/encode/audio/flac
- https://machinelearningmastery.com/text-generation-lstm-recurrent-neural-networks-python-keras/