Take Me Home, Country Code: an Artificial Intelligence designed for the analysis and creation of country music lyrics

Andrew Mersinger, Emmanuel Olaleye, Aishwarya Panchumarty, and William Boodon

Group 6

Group Email: andmers1@gmail.com

**Abstract—**

We intend to create an artificial intelligence system that, using a large collection of song lyrics and song genres, can ”write” its own music that reasonably emulates the style and tone of the genre it imitates. We have chosen to create music based of the country genre, although our program should be able to produce reasonable results for songs of other styles.

Specifically, we initially will use the BeautifulSoup Python library to parse internet databases containing top country songs and their lyrics, before downloading said lyrics as a training set. Afterwards, we will design a natural language processing algorithm that analyzes word frequency and relationships, in order to create a lyrical output theoretically indistinguishable in feel from man-made lyrics.

**I. INTRODUCTION**

Writing lyrics has historically been an interest for musicians and songwriters. Recently, however, this has also become popular amongst an entirely different crowd-- computer scientists dabbling in artificial intelligence, some for amusement, others for real life applications. Not all genres can be easily replicated via artificial intelligence, because some genres tend to have songs that utilize a wide vocabulary and use various unique words to mean the same thing. While generating lyrics using a data set of songs like this is still possible, using various synonyms overcomplicates a deep-learning probabilistic language model. To avoid this issue, we will be using a genre whose lyrics average a third-grade reading level (and only that because many oft-repeated words are long and complex) - country (Coroneos).

Country music is unique amongst genres in that its songs share themes far more heavily than do those of other genres-- the music is structured primarily around telling a narrative, usually from the perspective of a small-town or “country-living” person. These, while being the only set “rules” for the genre, are near ubiquitous within it-- as such, it makes the ideal genre for artificial language generation. While other genres have complex or specific formats-- Rap, for instance, must be “on-beat” and rhyme, amongst other things-- country, at least lyrically, is simple in nature. Not only does this lead to a much lesser emphasis on the importance of specific lyrical structure, it limits the diversity of the words used. Other genres’ rules require inventiveness in word-choice, their songs consequently having a wide word-pool out of necessity-- Country music’s universal tropes and themes inherently limit the grammar involved. The variety of words at play is, of course, still vast (as is anything involving the English language), but our group believes that this genre is still the best choice when attempting to produce genre-conforming lyrics.

**II. SPECIFICS**

**A. General Motivation**

Our group took heavy inspiration from the previously completed projects given as examples; Namely, the “Is it Gangsta?” AI designed to identify gangster rap and the AI created to write poetry. This system is designed to emulate parts of those intelligences, combining separate aspects of both ideas. We specifically chose country music based on the stereotype that the genre features only a couple unifying themes (trucks, the South, rural lifestyle, etc.). If this is the case, then it might lead to generation of more semantically cohesive songs overall given the commonality in subject matter. We predict that applying the probabilistic language model will yield interesting results-- providing either an interesting take on the tropes of modern music genres or, if that angle fails, some comedy gold.

**B. Language Choice**

We are using Python for both the web crawler and the AI system proper; while technically less efficient than other potential choices, being an interpreted language, it has a sprawling module library that would serve to make the task of programming this system more intuitive and efficient-- for example, BeautifulSoup is a Python library that is integral to the formation of the web crawler that will build the database of the AI system, as it provides an API that allows for extremely efficient, precise, and compact Document Object Model (DOM) tree parsing.

**C. Data Sources**

We initially used Songlyrics.com’s premade top-100 country songs list to obtain data, as the webpage was organized very conveniently-- a table, with a column dedicated to hyperlinks leading to song lyrics. This made it extremely easy to parse and thus an ideal data source for us to use when implementing this system. However, we quickly discovered that several hyperlinks were broken or missing; We now intend to source songs from several different databases in order to form a more comprehensive dataset. Our new training set includes data from Professor Dave Tompkins Music Database, as well as other sources.

**D. Algorithms**

The algorithms are using in this program include:

1) Data Retrieval: We are using the Python BeautifulSoup extension as both a web crawler and a data-parser. Beautiful Soup’s implementation is discussed in greater detail in Section IV.

2) N-Gram: In order to produce output, our artificial intelligence needs to analyze word relations and frequencies. The n-gram probabilistic language model is one designed specifically for calculating the probability of the next word in a sequence by analyzing a contiguous sequence of *n* words in a corpus, and as such is the perfect model to use to generate our lyrics. In order to calculate when a line break should occur, we can utilize the n-gram concept. We will also be using an exponential function to increase the probability of a line break as lines get longer and longer.

**III. WORKS/ LITERATURE REVIEW**

***DopeLearning: A Computational Approach to Rap Lyrics Generation***

Our proposed model is very similar to a model produced by five researchers in Finland.

Their paper, *Dopelearning*, describes an artificial intelligence that generates rap lyrics by analyzing the rhyme scheme, “beat”, and lyrical density of sample lyrics. This model takes sentence structure and vocabulary into consideration, using“a dataset containing half a million lines from lyrics of 104 rap titles” (Malmi, Takala, Toivonen, Raiko, Gionis, 2015). We will be using this model as an example, but ours will lack this complexity. Since country songs are not restricted by structure or rhyme, our model does not need to consider those facets. It will simply pick up keywords such as “trucks” and “roads ”. Based on these terms, their relationship, and their frequency, it will generate a song that emulates the general formatting of the genre. Even so, the researchers desired output is very similar to ours, and as such will be (and has been) a large assistance in our task. Furthermore, this paper was was also extremely helpful in our determination of an appropriate size for our sample data; the programmers who worked on Dope Learning report satisfaction with their results. While this is, of course, partially attributed to the algorithm that they developed, it is also an indicator of the quality of the sample data-- something we aim to emulate. Finally, this research guided us to understand how to structure out our project. It helped us set milestones, so we would know exactly what steps to take and how to determine if the current phase of work is productive or detrimental to the final task.

***Deep Learning-based Poetry Generation Given Visual Input***

Another model that is similar to our project is one made by two Norwegian researchers. This model comes up with grammatically correct and meaningful poetry by taking an image as an input. While the aim of this project differs greatly from ours-- they aim to understand deep neural networks-- the approach to the problem is very similar. We will specifically focus on their implementation of BeautifulSoup and the Long-Short-Memory Lasting network. The authors have used BeautifulSoup to collect their dataset from mldb.com/. Through it, they found the number of tokens, vocabulary words, and unique words. Another tool that these programmers utilized was the Long-Short-Memory Lasting network, which they used to make word sequences. This network is given an input in the form of a vector space model of a vocabulary, and it predicts the next words in the sequence for each given input. Even after using this difficult algorithm, the researchers admitted that they would not be able to produce a consistent structure. This, however, is perfectly reliable for us because our dataset-- country music-- also does not have a consistent structure. Their use of BeautifulSoup partially inspired ours, as well as assisted in our implementation. Their rigorous vetting of their dataset has inspired us to more fiercely analyze our sample set.

**Lyrics-generation-using-Ngrams by Shubham Jain**

Mr.Jain’s program uses songs composed by Bob Dylan, which are very similar to one another. He uses n grams to train his model. Although he imported the Natural Language Toolkit Library, he never used it. He made his n grams completely from scratch. Although this is impressive, he trains his model character by character. This method is very ineffective because the model randomly selects a letter and randomly chooses a letter to come after it. This resulted in output that was entirely without form, function, structure, or sense. Even with his training data (which was 30,000 lines long), the output of the program was complete and utter gibberish, and did not improve with the substitution of our data. While this may seem like a poor example, it is always best to learn from the mistakes of others as to not repeat those same mistakes ourselves. This failure of a program ended up speeding up our development by steering us clear of difficult practices and some specific mistakes.

**Rap-Lyrics-Generator by Keerthana Sriranga from Github**

Ms. Sriranga’s program is almost identical in function to our final system. It takes in different text files containing song lyrics, creates a dictionary to store every word and what words follow it, and determines, based on the frequency of word-pairs in the input files, which words to place next in the output string. In other words, it is plainly a bi-gram Markov model, with the only significant deviations from our project being its multiple inputs (which were easy to condense into a singular one), the genre its author intended to use it on (rap), and it is more simplistic bi-gram nature. As our planned model was a trigram, the group felt that any results produced by this program would be worse than what would be output by our finished project. As such, if we could satisfy ourselves with our training data’s results on this model, we knew we would be satisfied with our results from our project - this proved to be a sensible and sound approach.

**IV. SETTING UP THE PROJECT**

We have divided the project into two parts. The first part involves extracting data (lyrics) from various online databases, using the BeautifulSoup Python extension. The second part is divided into further subparts, but the overarching goal is to utilize deep learning and train a model using the sample set, utilizing the newly-trained model to create song lyrics.

**BeautifulSoup**

To find country lyrics to train our model, we used BeautifulSoup. BeautifulSoup is a library in Python which facilitates the parsing of HTML documents. Once we found websites that we deemed appropriate for use as datasets, we used Chrome’s Inspect Element tool to identify the specific tags on the website which contained links to song lyrics. Then, we followed each of these links to their respective page, and from there identified which section of said page contained the lyrics. Finally, we extracted the lyrics and placed them all in a text file for use as a training set for our probabilistic model.

Afterwards, we took this text file and put two beginning of line tokens at the beginning of each line in the song, and one end of line token at the end of line. This will help our model detect that it is reading in a new line and/or it is at the end of line. This is so that it will learn what words will most likely come at the start of the sentence and when to end a line.

**Model to train AI**

In order to train the AI, we needed to first identify the distinct elements of the vocabulary. From there, we worked to establish probabilistic links between words: determining the likelihood that any given word will follow any other given word.

In order to start a lyrical output, we used two beginning of line characters as the “current” and “former” members of our tri-gram. This meant that all of our songs start with words that started off songs from our training set. Then, as described above, our model would select a random vocabulary element from a separately-created set of meaningful words.

From there, the AI system determines the probability of the next word in the sequence by utilizing out tri-dictionary, wherein both the current and previous word influence what the next will be. We continue this process until it reaches a user-defined line count. As each successive word should be chosen based on a probability, the outputs should be at least marginally unique, even on the rare occurrence where the starting word is identical.

**V. PRELIMINARY RESULTS**

Out initial plan was to test out program similarly to the way the authors of DopeLearning: A Computational Approach to Rap Lyrics Generation tested theirs - By feeding it the first word or words of a song, and by compare the intelligence generated lyrics to the actual lyrics. We believed our tests should be similar to those of actual researchers. This approach, however, did not last. We determined that it would be far more efficient to simply downloaded an already-completed program from the internet to use to test the quality of our data set.  
  
After going through several programs, we found two projects posted in Github. The first project is “Lyrics-generation-using-Ngrams” by Shubham Jain. The second project is “Rap-Lyrics-Generator” by Keerthana Sriranga. We decided to give our data as inputs to these projects, so we would understand how to design our own project. We were satisfied with the program's performance but unsatisfied with our input; we found another internet song database to use and those results were extremely promising. From there, we moved on to designing our own model.

**VI. TESTING**

As stated in section five, initial results were, admittedly, poor. We had originally planned to base out algorithm off code obtained from Blazingking on Github, and as such we downloaded this program and gave it our input to test results. The program would generate a few words, but they would be lost in a sea of random characters. Considering this test program indeed chained characters instead of words, this result was not surprising to us. We also discovered that this test program was using very little actual learning and was instead randomly generating which character to place next. Our group decided to find a different program to use for data validation.

Results improved when we instead tested out training data with code obtained from Keerthana Sriranga. As this new program used Markov chains based off words, rather than returning the random array of characters, only words that were in the training data set were printed. While this was an improvement, we still had two primary issues; First, our training data was quite small (about 100 lines) – which led to our model having a limited vocabulary and repeating specific sequences of words multiple times (even during our limited early testing). Secondly, we came to realize that the text files we had for input were incredibly poorly formatted, containing multiple instances of “(chorus)” as opposed to repeating the chorus, “instrumental” for solos and the like, or other formatting text woven in with the lyrics. We set out to solve each issue independently.

Firstly, we spent a significant portion of time looking for a superior database. We eventually settled on Dave Tompkins Music Database, which had both a greater number of songs and generally better formatting than out previous source. Some issues persisted – such as the rare nebulous formatting issue – but overall, this website’s song formatting was far easier to work with. Furthermore, this website’s song catalog was at least ten times larger than that provided by SongLyrics.com, solidifying its place as our new dataset. This new training set led to superior results, although they were still low quality and made little sense.

To improve our training data further, we designed a small program that formats our file properly. Using regular expressions, we first remove all punctuation except for apostrophes. We made this determination as to keep words like “lovin’” formatted properly, and because we saw very little (if any) formatting using apostrophes in our training data. Accordingly, we would have kept hyphens-- however, several of our songs were formatted with lines of hyphens dividing them, and we determined that the value of making it easier to train our N-gram model outweighed the value of keeping words like “Honky-Tonk” hyphenated. It also makes every word lowercase, in order to ensure the model does not make mistakes like counting “love” and “Love” as different words. The final, and most important, task removing every occurrence of certain words--specifically “chorus”, “miscellaneous” and “instrumental”. These words were commonly used in formatting as opposed to being used as lyrics, and as such, removing them wholesale was a net gain on our project. This showed in our results, which were now almost entirely free of odd formatting mistakes - our primary problem, now, was creating sensible output.

From here, out group split into different tasks - some of our group members started work on the generative model for our program, while others began producing a more advanced dictionary for the model. The generative model was finished before the dictionary was, utilizing a simpler dictionary to produce output that the group deemed acceptable, even with limited input data to train from. Soon after, the more complex and theoretically superior dictionary was complete. To quickly summarize how this dictionary functioned: it used the “nltk” python library to make n-grams and tokenize the input file into words automatically. The keys in the dictionary contained bigrams, and the values were lists. Those lists contained tuples, while each tuple contained a word, and finally, the probability that word will follow the bigram. Unfortunately, we had trouble in the implementation process while using this algorithm, and so had to make the decision to not utilize it. In the end, we decided to simply use the original, simpler generative model.

**VII. THE GENERATIVE MODEL**

The generative model was trained on a line-by-line basis using the input data. First, it recorded the unigram, bigram, and trigram counts for each word sequence of length 1, 2, and 3 respectively. To accomplish this, the algorithm would first add 1 to the count of an individual word, then to the count of that word and its immediate preceding word, and then to the count of the word and its two immediate preceding words. The counts for this “multigram” model were stored in a 3-dimensional Python dictionary. To illustrate the function of this dictionary, consider the example phrase “I eat cake”. If our model reaches the word “cake”, it will first add 1 to that count using the word as a key in the dictionary. Then, it would add 1 to the count of “eat cake”, using “eat” as the key to the first layer, and then using “cake” as the key to the dictionary obtained from “eat”. Finally, it would add 1 to the count of “I eat cake” using “I”, “eat”, and “cake” in that order as keys to each subsequent layer of the dictionary.

In order to generate a unigram, bigram, or trigram simulation of country lyrics, the model uses random integers to pick one word at a time from a set of probabilistically likely words. For the unigram model, it simply selects a word each time at random based on the percentage of times that word appears in the corpus. For the bigram model, it selects the word given the percentage of time it appears after a given previous word, and for the trigram model, it uses two previous words. After experimenting with unigram, bigram, and trigram models, we settled on using the trigram model alone as it maintained the highest degree of grammaticality while not overfitting the data, i.e. simply generating the existing data word for word. To begin a line, the trigram model uses two “beginning of line” tokens (represented as <BOL>) as the given preceding words, and a line is ended when the model generates an “end of line” token (represented as <EOL>). These tokens were initially added in the training process in order to expedite the later generative algorithm. The lines are completely independent of each other.

**VIII. RESULTS**

We tested our intelligence system several times on with several different song lengths; we found that it worked the best on shorter songs (5-10 lines), as those tended to hold consistent themes and have smaller chances of error. Our system is, however, still entirely capable of writing songs with longer formats; A few of our favorites are presented below.

**Song 1 (Twenty Lines):**

*cowgirls don’t cry for me*

*i hope you know i don’t know what to throw it away*

*ooh it’s off to the table*

*she climbed inside*

*i smell trouble*

*i’m feedin’ the dog the cat*

*life goes on*

*cause a stir*

*wellhey hey hey hey honey i’m home*

*well he must have thought i’d fall again so easily*

*this phase is gonna play its part*

*be happy all the time alone*

*i don't need to tell us*

*mama knows the highest hill and field on a roll in the barn the*

*barn is on the bottle break against the waves*

*and a big neon sign*

*it’s alright alright alright*

*we took turns at becky there was no chance*

*i heard her say i love you yet but i need your love amazes me that*

*it makes you sad*

*you said*

*la la*

**Song 2 (Nineteen Lines):**

*just a singer of simple songs  
there at the beach to have  
but i've still got a new kind of hurry i've been weak  
just one more chance to begin  
'cause every time i see you soar above the sky within my reach  
well there's no place to fall  
workin' in a trap  
so take a load off fanny  
i feel like rockin'  
leave us  
what condition my condition was in  
i thought i'd get red  
travis tritt  
maybe i'll do everything i need  
now i'm learning to see all the signs  
hear a little dust on my phone   
i'm mad about you iÃ«m i'm crazy bout you iÃ«m crazy bout you*

*looks like heartache*

*you don’t have to believe*

While our output clearly has some issues, such as more odd problem characters and the occasional line repeated entirely from a song in our input file (“take a load off, Fanny”), those appear to be rare, and given that our input corpus contains 1,000 songs, the occasional problem character is manageable. Other than that, our AI’s songs are somewhat sensible overall (Both songs are love songs, for example), while small snippets of them (such as lines 1-6 of song 2) actually make a good amount of sense on their own. With some additional adjustments, particularly, manual formatting, the output is capable of being rather palatable:

*Just a singer of simple songs,  
there at the beach to have  
But I've still got a new kind of hurry; I've been weak  
Just one more chance to begin  
'Cause every time I see you soar above the sky within my reach  
well, there's no place to fall...*

**IX. IMPROVEMENT DISCUSSION**

If we were to revisit this project, we may be able to improve the clarity of the songs written via increasing the layers in our N-gram; a quad-, penta-, or sextagram could produce far more sensible lyrics. However, there exists a drawback to this approach: a high enough ‘N’ could eventually result in an AI that merely reproduces the training set, as there would only be one possible sequence of words for most given lines it could use. We feel that a tri-gram was the best approach to create sensible output without overfitting the data - furthermore, for each layer you add to an N-gram, the time and space complexity is exponentially increased; A problem we avoid by utilizing a tri-gram.

Additionally, we would also create and save several different dictionaries produced from several different training sets, dividing them by genre, so that our AI can write songs of any given type. While this seems useful in theory, the amount of work required to accomplish that task is far too immense to justify the result. Furthermore, even without this restructuring, there is not any inherent design bent in our AI that prevents it from being trained on a set of non-country songs and producing songs reminiscent of those.

Another improvement we would make is that we would make our model learn the structure of the songs as well. Instead of giving one big text file that contains all the songs, we would give it command line arguments that lead to the path containing the directory of all of our input text files. Each text file would contain a singular song. In our project proper, when we formatted the text file used for input in our current iteration of the project, we would take out the lines that defined the structure of the song (such as [chorus] or \*instrumental break\*). Doing this allowed our model to be more easily trained because it would only have to worry about the word order, and furthermore did not have to consider the structure of the given song because it would read from a massive file of compiled songs stripped of their formatting tags. Thus, it only needed to learn about the semantics of the words. If we were to improve it next time, we would collect metadata. For example, we would collect variables, such as the number of lines or verses in each song, how many words each line has, et cetera. Our model would be trained using this metadata, so it would figure out on its own about how many lines and verses it should produce. Furthermore, it should take the chorus into consideration, and structure the song this way. The goal would be to train our model to make output songs better resemble those songs written by human songwriters.

The final improvement that we would consider making would be to design out data structure in a more memory-conscious way. Currently, we take no consideration to potentially running out of memory; our program is not space-efficient. We had plans to change it if issues arose, but none showed themselves, so we continued on. In a setting where we were designing this intelligence for commercial or government use, or in any other circumstance where failsafes were a necessity, it would be imperative to ensure our program can process the necessary amount of data without crashed or bugs.

**X. CONCLUSION**

It is our group’s firm belief that the AI we have created produces desirable output. While certainly not a billboards-smashing artist at any rate, the songs generated have evolved far from the gibberish-spewing nonsense of the system’s early days. While large blocks of text may not make much sense - the AI likes to frequently change what topic its song is about - small sections tend to be coherent in both basic semantics and overarching theme, even if larger sequences in the song may not be entirely meaningful. For what it’s worth, our AI system produces output far exceeding the models we initially based it off of, being both more coherent and cohesive.

All in all, while not perfect, our AI does what it says on the tin: it rights country songs imitative of those it is given as inputs. It doesn’t always make perfect sense, it doesn't always stay on topic, and it occasionally doesn’t even properly form sentences - but often, it does all of those things. As that is the nature of a random system, we don’t worry much about the times it has little hiccups - we much prefer to focus on the numerous times it does its job well.

**XI. ACKNOWLEDGMENT**

The authors would like to thank such influential figures as Noam Chomsky, for being a pioneer in the field of computational linguistics; Andrew Markov, for devising the Markov chain, and all computer scientists that have come before, as they have made coding so much easier for us.

We would likewise love to give our highest praise to Leonard Richardson, the creator of Beautiful Soup, as the time that he has put into working on that library has saved us incredible amounts of time and effort. We would like to also thank Steven Bird, Edward Loper, and Ewan Klein, the creators of Natural Language ToolKit (NLTK) for creating a library that can parse through large datasets to create any type of n grams.

**XII. REFERENCES**

Coroneos K (2015). Pop Country Lyrics Score at a 3rd Grade Level According to New Study.

Retrieved from

https://www.savingcountrymusic.com/pop-country-lyrics-score-at-a-3rd-grade-level-acco

rding-to-new-study/

Ferraro, F (2018). Probability and Language Modeling [PowerPoint Presentation]. Retrieved

from

csee.umbc.edu

Jain, Shubham. “Lyrics-generation-using-Ngrams”. GitHub, 27 Sep. 2017,

github.com/Blaziking/Lyrics-generation-using-Ngrams

Loller-Andersen, M., & Gambäck, B. Deep Learning-based Poetry Generation Given Visual

Input. Retrieved from

http://www.computationalcreativity.net/iccc2018/sites/default/files/papers/ICCC\_2018\_p

aper\_59.pdf

Malmi, E., Takala, P., Toivonen, H., Raiko, T., & Gionis, A. (2016, August). Dopelearning: A

computational approach to rap lyrics generation. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 195-204). ACM. Retrieved from

<https://dl.acm.org/citation.cfm?id=2939679>

Sriranga, Keerthana. “Keerthanasriranga/Rap-Lyrics-Generator.” GitHub, 28 Apr. 2017,

github.com/keerthanasriranga/Rap-Lyrics-Generator.

Link to Professor Dave Tompkins Music Database:

<https://www.cs.ubc.ca/~davet/music/list/Best2.html>