Comparison of Music Lyric Generation Algorithms: Markov Chains and LSTM RNN

Kerem Enhos [*1] **Muralikrishna Shanmugasundaram** [*1] **William Varner** [*1] **Yunus Cem Yılmaz** [*2]

Abstract

In this paper, the top 100 pop songs in Genius website is used to extract song features and generate new songs. Information on number and repetition of sections, number of lines and words per line are extracted. Then k-means clustering is used to identify different song structures. These song types are generated using Markov chains and LSTM RNN. We also test whether a conversion into the phonetic alphabet improves results. The performances of these alternate models are compared through generated samples and it is found that Markov chains lead to worse performance in terms of coherence, but perform better in terms of musicality, especially with the use of the phonetic alphabet conversion.

1. Introduction

Pop is a wide genre of music that many people enjoy, in all areas of the world. Pop music plays a large part in mainstream media, and some of the biggest pop songs have millions and millions of views on YouTube. It is clear the pop music is very influential in many cultures. The majority of music has come from the creativity of the human mind, but with advancements in machine learning in the last few decades, the world has seen a rise in lyric generators. Machines can do this by detecting patterns in words and lyric structures in different songs. Taking interest in this and inspired to integrate machine learning and music, we developed a lyric generator of our own.

Machines can implement lyric generators in a number of ways. Our approach can be divided into two parts: 1) song structure and 2) language. First, our generator samples many songs to collect information to build a skeleton structure. If a regular pattern can be detected in different songs, it is possible to create a new song based on the "averages" of different clusters. Then, the generators adds words to the "skeleton" structure based on the lyrics samples. Some of the most popular methods to generate language involve Markov chains and neural networks. Markov chains are stochastic models that use maximum-likelihood to predict the next word based on the previous word. These are useful because they are relatively simple and straightforward, while neural networks are a little more complicated. Using perceptions and multiple layers, neural networks can also work well, but training can consume lots of time. We decided to generate lyrics using both to see differences in lyrics.

In this paper, we collect information from Genius's Top 100 Pop Songs (Genius, 2019) to create our own hit pop song. The main objective of this is to construct coherent songs, that emulate famous pop hits in language and song structure, but do not repeat them. There are many ways lyric generators can be implemented. Our project was an experiment to use clustering and probability-based sentence creation to generate different sets of lyrics. It was our goal to apply machine learning concepts to a new topic, and there were many things we learned in our process. In the second section, we detail our approach to making a lyric generator, explaining the process and providing justification for our decisions. Next, we present our results for the project, as well as an analysis of the results. Then, we discuss different aspects of the lyrics generator that could benefit from further work and improvement. Finally, we conclude with a summary of our results and what we learned from the project.

2. Building the Lyrics Generator

In order to build the lyric generator firstly the generation of the dataset is completed. After constructing the appropriate dataset clustering algorithm is applied in order to construct a suitable format for songs by clustering the structures in the dataset. Finally, two different methods are applied for generation of lyrics. Firstly, Markov chain based algorithm

[*]Equal contribution [1]Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, USA [2]Department of Economics, Northeastern University, Boston, MA, USA. Correspondence to: Kerem Enhos <enhos.k@husky.neu.edu>, Muralikrishna Shanmugasundaram <shanmugasundaram.m@husky.neu.edu>, William Varner <varner.wi@husky.neu.edu>, Yunus Cem Yılmaz <yilmaz.y@husky.neu.edu>.

Introduction to Machine Learning and Pattern Recognition Project - EECE5644, Northeastern University, Fall 2019. Copyright 2019 by the author(s).

is used with two different alphabet inputs of dataset; English and phonetic (IPA) alphabets. Then, neural network based generation is applied to English alphabet for comparison of two algorithms.

2.1. Extracting Data from Lyrics

The first step in the process was to define relevant data that could be extracted from a set of lyrics, in order to recreate a new song based on specific parameters. We took a mathematical approach to modeling songs, by representing each song in a 6-dimensional vector. The six data points per song includes:

- Number of sections
- Average number of lines per section
- Standard deviation in lines per section
- Average words per line
- Standard deviation of words per line
- Repetition score

The sections were separated by carriage return characters. Repetition was detected by comparing the first line of each section.

The idea was to use these data fields to detect trends in length and variation of lyrics. Some songs are highly repetitive and have short lyrics, while other songs have no repetition, and have really long lines. Some songs have regular verses and choruses, and some songs just have long raps. Similar songs are classified in the same clusters, which have "averages" for each of these data points. This "engineering" approach is a good estimate to how songs are built. It provides a rough skeleton for the foundation of a song. While it was a little limited in how to construct a song, it gave us enough information to build a structure based on other popular pop songs. Our dataset consisted of Genius's top 100 pop songs. The lyrics were extracted and pasted into 100 separate text files. We used a simple python program to extract the data from all the songs, and export the 100x6 data matrix, which was the clustering input. Each dimension was normalized to values between 0 and 1.

2.2. Clustering Algorithm

The foremost step to begin with the clustering is to know the optimal number of groups into which our data must be clustered. We employ the **Silhouette Method**, a visualization method that helps to determine the optimal number of clusters.

The silhouette of a data instance is a measure of how closely it is matched to data within its cluster and how loosely it is matched to data of the neighboring cluster. The range of the $silhouette\ value$ is between +1 and -1. A high value is desirable and indicates that the point is placed in the correct cluster. If many points have a negative silhouette value, it may indicate that we have created too many or too few clusters.

The silhouette value $s(i)$ for each data point i is defined as follows:

$$s(i) = \frac{(b(i) - a(i))}{(max\{a(i), b(i)\})}, if |c_i| > 1 \tag{1}$$

and

$$s(i) = 0, if |c_i| = 1 \tag{2}$$

where $s(i)$ is defined to be equal to zero if i is the only point in the cluster. This is to prevent the number of clusters from increasing significantly with many single-point clusters. Here, $a(i)$ is the measure of similarity of the point i to its own cluster. It is measured as the average distance of i from other points in the cluster.

For each data points $i \in C_i$ (data point i in the cluster C_i), let

$$a(i) = \frac{1}{|C_i| - 1} \sum_{j \in C_i, i \neq j} d(i, j) \tag{3}$$

Similarly, $b(i)$ is the measure of dissimilarity of i from points in other clusters. For each data point $i \in C_i$, we now define

$$b(i) = \min_{i \neq j} \frac{1}{|C_i|} \sum_{j \in C_i} d(i, j) \tag{4}$$

where $d(i, j)$ is the distance between points i and j. Generally, $Euclidean\ Distance$ is used as the distance metric. The algorithm is computed as follow:

- Compute clustering algorithm (k-means clustering) for different values of k, by varying k from 1 to 10 clusters.
- For each k, calculate the average silhouette of observations. The Silhouette Values can be easily calculated in $MATLAB$ using $evalclusters()$ function.
- Plot the curve of $silhouette\ values$ vs number of clusters k
- The value of k corresponding to the global maximum is considered as the optimal number of clusters for the dataset

In Figure 1, the peak is observed at k = 2. Hence the optimal number of clusters is chosen as 2 for our case. Now

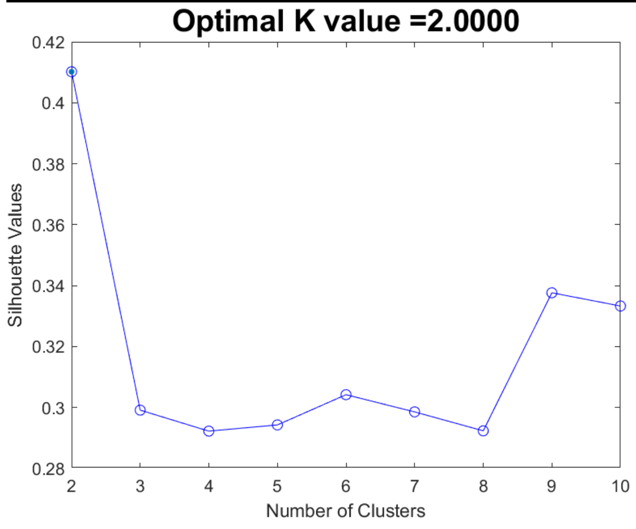

Figure 1. Silhouette values as a function of number of clusters, k, is given. According to the figure, maximum value is obtained at k=2, which represents the optimal value of clusters.

Table 1. Cluster means of normalized feature vectors. (L/S: Lines/-Section, W/L: Words/line, STD: Standard Deviation)

	CLUSTER 1	CLUSTER 2
# OF SECT.	0.27750939	0.18040621
MEAN # OF L/S	0.13302853	0.35859870
STD OF L/S	0.15713020	0.47500902
MEAN # OF W/L	0.48256413	0.49773274
STD OF W/L	0.45889511	0.35712891
REPETITION SCORE	0.56644800	0.28761600

using the appropriate k value, k-means clustering is performed with the songs dataset using the $MATLAB$ function $kmeans()$, which returns the cluster indices and cluster means of each data points. Cluster means of normalized and non-normalized feature vectors are given in Table 1 and Table 2 for cluster 1 and 2.

The cluster results given above will be used to decide on the lyric structure (different lyrics for each cluster will be generated). Line lengths, section lengths, number of sections and the repetition of sections will be decided on using the data above. When standard deviations are provided (number of words in a line and number of lines in a section fit this description), will be used and the variable means to draw from a normal distribution and choose the thresholds for the number of words in a line and number of lines in a section accordingly. These thresholds change between separate sections. For the repetition of sections, one of the first two lines will be selected as the chorus and repeat it according to the following rule: it will be drawn from a uniform distribution between 0 and 1 for every section. Then it will be compared to the drawn values with the repetition score and if the repetition score is larger, this section will be chosen as the chorus instead of an original section.

Table 2. Cluster means of non-normalized feature vectors.

	CLUSTER 1	CLUSTER 2
# OF SECT.	11.49275353	8.87096767
MEAN # OF L/S	5.327999275	9.27547725
STD OF L/S	2.13126903	5.56688943
MEAN # OF W/L	7.086869034	7.173174698
STD OF W/L	1.850334704	1.457065597
REPETITION SCORE	0.519244002	0.263648001

2.3. International Phonetic Alphabet (IPA) Conversion

Singing, song lyrics and concurrently poems are phonetically important concepts in artistic media. The rhyming phenomenon can be accomplished with similarly written words, which may have higher probability of resulting in homonym. This procedure can be called as mechanical rhyming. On the other hand, homophone words can be paired more easily when the similarity between words is assessed in International Phonetic Alphabet (IPA).

For this purpose, lyric generator with Markov chain algorithm is assessed with using IPA words that are generated through a dataset constructed in English language. IPA translation is completed by using a toolbox utilizes the Carnegie-Mellon University Pronouncing Dictionary to convert English text into the International Phonetic Alphabet (Phillips, 2019).

After translating whole dataset into IPA, Markov generation process is implemented with the translated dataset. As it will be explained in the next subsection, Markov generator algorithm assess the similarity between words that are end of each line, by using Levenshtein distance. Since this concept measures the similarity between two words using the number of character changes on a word that would be necessary to equalize to the word that is compared, using IPA instead of English alphabet will end up in more accurate and more complex rhyming results. So the same mechanical rhyming approach is applied to IPA, which will result in higher accuracy of rhyming. However, as it is explained in Section 3, the coherence in lines are more distorted compared to other lyric generation algorithms.

In order to read out the generated lyrics more easily, IPA to English translator is also implemented. A dictionary with the words that are uniquely used in the dataset is constructed, and the IPA equivalent of these words are stored in a Python dictionary. By this way, IPA to English translation is completed. However, this approach also leads to the problem of translating homophone words incorrectly, such as the word $'I'$ can be mistranslated as $'eye'$. This problem also leads to higher degradation in the coherence of the lines of the generated lyrics.

2.4. Markov Generator

For the task of lyric generation, we have decided to turn on Markov chains, because they are a nice and simple way of getting random word sequences that are mostly original and have the potential to make sense. With Markov chains, we record the probability of a switch from one state to the other. In our case, a state is going to be a word (or a group of words), so that a Markov chain just predicts the transition from one word to another in a line. Using these transition probabilities, we can draw from a group of words that could possibly come after the word (state) that we are currently on, randomly.

One important feature of Markov chains is that they make it possible to track state changes both backwards and forwards. So, we can use the word before to predict the word after, but we can also use the word after to predict the word before.

Lyrics should rhyme most of the time because this helps it sound more rhythmical and easier to listen to. We will use the fact that Markov chains can predict the previous word using the next word to allow rhyming to occur. We will start from the final word of a line and make sure that it rhymes with the previous line once in every two lines (so our rhyming structure is of the form AABBCC). To do so, we will draw randomly from all the words that are in our dataset with frequency weights and we will discard the resulting word if the final two characters are not the same as the line above (this is where the phonetic alphabet conversion should help with the lyric quality). The other line that does not have to rhyme is just picked randomly from the weighted probabilities.

Our code will try this 2500 times, each time relaxing the constraint a bit more to a point where after the 2500 tries, the last two words could be completely different. The relaxation of the constraint is done using a concept called Levenshtein distance. This concept allows us to measure the similarity between two words (in our case it is just two characters) using the number of character changes that would be necessary to convert one into the other. For example, to turn "back" to "tuck", we need two character changes. In our model, we do our first 1250 trials while rhyming without allowing the code to accept any last two characters that are different. For the next 1250 trials, we allow one of the two characters to be different and finally after 2500 trials, we accept any word as the final word of the line. In case the last word is not made up of two characters either in the line before or the line that is being generated, we use the second word before the last (if this is the current line, we generate a word according to our rules) to ensure that rhyming still works with the same principle.

As we have mentioned, we use the backwards probabilities to construct every line of our lyrics. What this means is that we will be randomly drawing from the possible words that could come before the word that we are currently on, which is once again weighted by the frequency of that word occurring before our target word (the word we are currently on). We do this by drawing from a uniform distribution and adding all the probabilities of words occurring before it until we reach the threshold set by our uniform distribution draw. The word whose probability leads to the exceeding of the threshold is chosen.

Our code interprets each line as a separate entity while creating these probability dictionaries for each word (state), so the last words of every line are ignored in the creation of these probabilities (they are not succeeded by any word).

We have included a variation of the Markov chain generation models by allowing the code to assume that the current state is either the current word or the current and the next word while predicting the previous word. Using a single word as the current state allows for more randomly generated (original) lines, but it is less likely to be coherent. Using two words as the current state allows more sentence structure to seep in and creates more coherent lines, but it is also more likely to repeat a line in our dataset due to decreased probabilities of randomness. We are not sure which quality would be more important for our implementation, so we will decide to try both cases.

Also, in our model's implementation of the two word current state Markov chain, if the final two words in a line had never occurred before, the model reverts back to using the one word current state to ensure that lines are not prematurely ended. Similarly, if the current word has never occurred before in the dictionary (if it was only used once and it was at the end of a line for example) the next word is selected using the frequency weighted draw from all the words in the data set.

A final wrinkle that we introduced above and is also a part of this Markov model is the addition of the phonetic alphabet spelling of words. We will convert our original data into the phonetic alphabet and use the same code as above (one type of lyrics with the current word as the current state and the other type of lyrics with the current and the next word as the current state) to generate lyrics. We hope that doing so will help with the rhyming of lyrics and increase the quality of these lyrics, but we also know that we may lose some coherence due to the process of switching back, which will not be able to discern words that sound the same but are written differently. Since we don't know which of these effects will prevail, we will also include these models in our analysis.

2.5. Long Short-Term Memory (LSTM) Bidirectional Recurrent Neural Network Generator

We will also explore neural networks because they perform well with non-linear logic, which is usually the case with language and lyric building. More specifically we would like to look at recurrent neural network models that can use its own output as an input along with the input from the model (or the previous hidden layer), (Schuster & Paliwal, 1997) which would lead to further reinforcement learning using a smaller sample size.

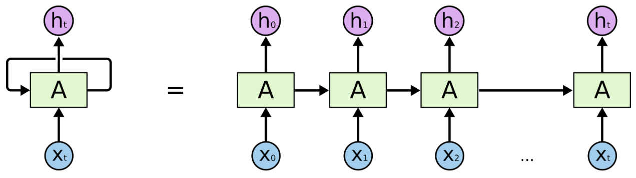

Figure 2. RNN algorithm representation. (Olah, 2015)

We will then use a smaller subset of RNNs which are long short-term memory models, which have a more complicated structure with nodes that decide which information from the previous states to keep and which ones to discard. The second set of nodes then update our beliefs using the current data and the data that we did not choose to forget. A final node then decides which of these features of the data to send as the output and which ones to remember. These features are complemented with a longer term memory, as pictured below at the top line which means that certain features will not be forgotten due to vanishing gradients.(Hochreiter & Schmidhuber, 1997) What all this means for us is that, this model could use information from many states before our current state to fix issues with our current state (tell us about the gender of the person that was mentioned 5 words before as an example). This should lead to better results than our Markov chain model, but it may struggle with creating original lines, especially due to our smaller sample size.

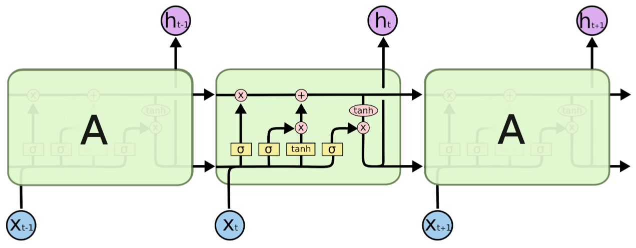

Figure 3. LSTM algorithm representation. (Olah, 2015)

We will be following the most common text generation architecture for LSTMs in our model as well, because it also makes sense with our own application. We want lines that are original and make more sense than the outputs of the Markov chain models.

The common text generator uses the categorical cross-entropy loss function (which is negative log-likelihood maximization after one-hot encoding the output matrix, much like what we did for question 1 in the second exam). The function looks like the following (where y is the actual output and $\hat{y}$ is the predicted output):

$$L(y, \hat{y}) = -\sum_{j=0}^{M}\sum_{i=0}^{N}(y_{ij} \times log(\hat{y_{ij}})) \tag{5}$$

We have also used an optimization algorithm that closely follows the gradient descent model that we learned about in class. This algorithm called root mean square propagation converges faster than gradient descent and we had limited computer power, so we chose to use it over gradient descent. It follows the following formula (where g is gradient descent, C is the loss function and beta is the moving average effect):

$$E\left[g^2\right]_t = \beta E\left[g^2\right]_{t-1} + (1-\beta)\left(\frac{\delta C}{\delta w}\right)^2 \tag{6}$$

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{E\left[g^2\right]_t}}\left(\frac{\delta C}{\delta w}\right) \tag{7}$$

Finally, the hidden layers have a sigmoid activation function (along with a hyperbolic tangent function in the second set of nodes) and the output layer has a softmax activation function, just like in the first question in the exam. The final transfer function leads to results between 0 and 1, which helps the model decide which output (previous or next state) is most likely. The math behind these are as follows:
Sigmoid:

$$f(x) = \frac{1}{1+e^{-x}}$$

TanH:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Softmax:

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}}$$

Since we wanted this to be a fair comparison between the two models (Markov chains and neural networks), we initially tried to use the same technique as the Markov chains of starting with a random word (or two words) and then using the previous word (two words) as the input to guess the previous word. We have constructed such models (with the transfer functions, optimization algorithms and loss functions described above), but this did not take advantage of the advantages of LSTMs, and led to the repetition of one to two words throughout the model. This was due to the small amount of information being portrayed in such models, which when combined with a smaller sample size and a large number of epochs led to these results (it is also important to note that the outputs of neural networks are not generated randomly like Markov chains). So, we had to scrap this idea (the codes are still in the data files) and make use of the advantages of LSTMs.

So we used a bidirectional (while training, the output is found using the states or words before and after it) LSTM RNN which used the nearest 10 words on each side to predict the output in our paper. While doing so, we used a package called textgenrnn (Woolf, 2017). This package allowed us to generate random lines because the package fed random seeds from the data into the model (note that the transfer functions, the optimization algorithm and the loss functions used did not change with this package). We could not integrate rhyming though, because we could not generate a random seed at every line (because seeds are supposed to be longer) and we could not think of a loss function that could ease this problem.

We generated our samples through a 5 hidden layer, 250 perceptron model with 20 epochs. For every epoch, the model used 20% of the sample to validate the results it originally obtained. We have chosen these model specifications in order to account for the non-linearity of language within songs, but we kept the number of epochs low and the validation sample size high to minimize overfitting issues.

We then used the random outputs that are generated via random seeds to form our songs using our clustering results presented above.

3. Results and Analysis

We have attached a sample section from each model to more accurately visualise the strengths and weaknesses of each model. A Markov chain model without a phonetic alphabet conversion (this sample comes from the model that uses the next two words to generate the output) looks like the following:

```
Begins we watched the ritual
They test me my love the real
Cities and I will never go tell
Seat of style is you seen a small
Looks like tomorrow when family's all
To his gift keeps on them ill
```

As we can see, the model generates less than coherent lines, but the generated lines are original. The rhyming also only works sometimes since the model just repeats the last two characters, which may not sound the same in English.

Then we will look at a sample section generated by a Markov chain model with a phonetic alphabet conversion (this sample comes from the model that uses the next two words to generate the output) and looks like the following:

Table 3. Normalized average ratings that are obtained subjectively by the authors and objectively according to the criterion explained earlier.

		SUBJECTIVE	OBJECTIVE
MARKOV	WORD 1	0.411	0.039
	WORD 2	0.906	0.587
MARKOV IPA	WORD 1	0	0
	WORD 2	0.386	0.288
LSTM		1	1

```
Of we are well I seem a bottle
You my body rock is it up double
And I know that clap clap clap along
Almost a few things that thong
A nation pretty but I'm in line do what
In the other like happiness I do what
I always in love me up all
```

The samples generated with this method end up rhyming better and more complex ("bottle" and "double" here) than our previous model, but this also leads to a further loss in coherence (even though it is a minute change) in the lyrics due to issues with conversion. We can't really observe this in this sample, though words like "night" end up becoming "knight" instead.

Finally, we will look at a sample section generated by a LSTM RNN model:

```
That's why look in the known and when
We go crashing down we
Come back every time just cause
everybody is your mind
But I love it to marriage
But I'll only stay with you one more
```

Here there is more coherence both within and between the lines, but due to our modeling restrictions, rhyming does not occur. The lines are also sometimes the same as lines in the songs in our database. This points to an overfitting issue which could be solved by less epochs, more validation data and more data overall.

We have generated 10 songs for each model and each song type and evaluated their performance using our subjective assessment of each song's overall quality (out of a hundred) and the grammatical correctness (given that they make sense) of lines. We then averaged our subjective assessments over songs and people to figure out the subjective assessment metric for each model and song type pair. We then average the grammatical sense metric of each song (grammatically correct lines over the total number of lines) to obtain the objective assessment metric for each model and song type pair. The table below shows this value for each of the model and song type pair. As we can see in objectively rated column of Table 3, the Markov models end

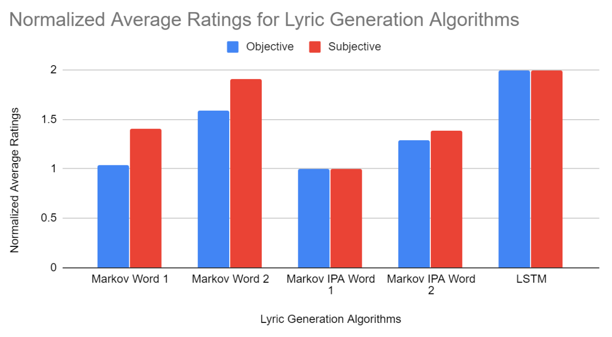

Figure 4. Normalized average ratings obtained by assessing 10 songs for each lyric generation algorithms (Markov chain with 1 word and 2 words, Markov chain with IPA dataset input with 1 word and 2 words, and LSTM algorithm) and each song type, subjectively and objectively.

up producing a very small number of lines that are grammatically correct (especially with a single word prediction system). This leads to lower subjective performance as well, because the lines end up making little sense, leading to little or no coherence within a section. The use of the phonetic alphabet leads to somewhat worse results; even though the rhyming is improved, the coherence drags the scores down (this also leads to a slight decrease in the objective score). The LSTM RNN model outputs do much better in terms of grammatical correctness, and they do exhibit a similar improvement in subjective scores even when the lack of rhyming and the duplicative nature of these songs (lines from original songs get repeated a lot) are accounted for. Given these results, the LSTM RNN model seems to be the best model to generate songs, but there is still a long way to go before we can create coherent, rhythmical and non-duplicative songs.

4. Future Work

The project could benefit from a few improvements. First, the method for representing song structure is a little rigid in its design. For example, we assumed that each section has around the same number of words per line, in order to emulate a regular sort of rhythm. Within a particular section, there would be no variation in line length, which isn't necessarily a requirement in songs.In addition, the repetition factor doesn't take full structure (ABAB, ABCB, etc.) into account. Similarly, repetition of certain lines may also be implemented (along with the chorus that we created) to create a much more realistic song. Also, the method of implementing the repetition in new songs was somewhat randomized. Having a more sophisticated, flexible way to detect/implement patterns in songs would make the lyric generator stronger. Rhyming is another issue that we could improve on. The addition of the phonetic alphabet conversion helps with rhyming, but the structure of the rhymes we create are quite rigid and only include the last two characters. In reality, there could be a more randomized structure for rhymes, and our model can benefit from the use of a more randomized structure to generate rhymes.

The lyric generator would also benefit from a much larger dataset. It would also be helpful if it were made up of songs of a narrower genre, since that would lead to more coherent lines. Genre selection can further benefit the model if we were to use genres where grammar and sentence structures are more coherent. Our models struggled with creating comprehensible and grammatically correct lines that were also not taken verbatim from a song because of these factors. It also hindered our ability to include more words in the predictions of the next (or previous) word. With a larger dataset a Markov chain with three words as a predictor could function without being too duplicative. Further data cleaning could also help the search for a more accurate model. Removing non-word interjections (sounds like 'ooh' or 'ah') could lead to better sounding songs.

Other improvements include additions to song structure. Word count is a blunt way of controlling rhythm and cohesion between lines. Another, more sophisticated method that would have been useful here is the use of the number of syllables instead of the number of words to control line length. The implementation would be much harder than counting words, but it would improve song quality significantly. Another important feature that could be added to these generators would be a loss function (or increased probabilities for the Markov chain) that would take alliteration into account. Repeating similar sounds (letters in this case) helps with the musicality of the song.

One other improvement with our project that would make it much more exciting would be to make it more interactive. Assuming that we could get a larger dataset, it would be interesting to allow the users to select artists, genres, periods (like decades) that would be used to generate the songs. It would lead to songs that would be more coherent, but it would also lead to more interesting results.

One other issue that we can't improve upon unless we change the language is about coherence. Conversion to the phonetic alphabet causes certain words like "I" and "eye" to be interpreted as the same word. This leads to less coherent lines due to the words that sound the same but are written differently (but these differences can't get picked up in the conversion). We would suggest using a phonetic language like Turkish or Spanish for better results (where the conversion to the phonetic alphabet would not be necessary).

Further improvements can also be made in the song generation structure. The use of more words to predict the next (or previous) word could lead to more coherent lines, especially with Markov chains. We have used a LSTM RNN which does make use of the additional information that can be used in the previous (or next) words, but it was limited by the line that it was observing. Allowing the song generator to follow the whole song may lead to more word combinations and more coherence between lines (this would also require the program to differentiate between different songs). Coding our own LSTM RNN would solve these problems and would allow us to assign loss values associated with rhyming, leading to a more cohesive song.

5. Conclusion

In this paper, we generated and collected dataset to extract song features and generate new songs. We extracted information and then use k-means clustering to identify different song structures. We generated these song types using Markov chains and LSTM RNN. Assesment and comparison of generated lyrics are done both subjectively and objectively by defining specific criteria. We compared the performances of these alternate models through generated samples and found out that Markov chains lead to worse performance in terms of coherence, but perform better in terms of musicality (rhyming), especially with the use of the IPA conversion. On the other hand, with LSTM, rhyming could not be maintained but much more coherent text generation could be possible.

Acknowledgements

The code repository that is used in this work can be found here.The repository is formed in the following fashion: Extracted and Edited Lyrics file includes all the songs that were used. Data Extraction and Clustering Code file includes all the code that we used to clean and combine lyrics, extract features from these lyrics and k-means cluster these extracted features. Clustering Input and Results file contains the extracted features from the song along with the k-means clustering analysis results. Song Generation Code file includes all the IPA converter files, along with the code that we used to generate songs using these extracted lyrics and features with Markov Chains, RNN LSTM and LSTM. Sample Songs from Each Model file contains 10 sample songs generated from each model. Generated Song Analysis file contains all the analysis of the generated songs and comparisons between models.

We also would like to thank the following people for their open-source contribution: Max Woolf whose text generating neural network, Michael Phillips whose IPA converter we modified. Kemal Burcak Kaplan whose markov lyric generator and Mohammed Ma'amari whose LSTM model we drew from.

References

Genius. *Genius – Top 100 Pop Songs*. https://genius.com /Genius -top -100 -pop -songs -lyrics, 2019.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

Kaplan, K. B. *Serdar Ortac Generator*. https://github.com /kaplanbr /Serdar -Ortac -Lyrics -Generator, 2017.

Kassambara, A. *Determining The Optimal Number Of Clusters*. https://www.datanovia.com /en /lessons /determining -the -optimal -number -of -clusters -3 -must -know -methods/, 2019.

Ma'amari, M. *AI Generates Taylor Swift's Song Lyrics*. https://towardsdatascience.com /ai-generates -taylor-swifts -song-lyrics -6fd92a03ef7e, 2018.

Olah, C. *Understanding LSTM Networks*. http://colah .github .io /posts /2015-08- Understanding -LSTMs/, 2015.

Phillips, M. *English to IPA*. https://github.com /mphilli /English-to-IPA, 2019.

Schuster, M. and Paliwal, K. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45 (11):2673–2681, 1997.

Woolf, M. *textgenrnn*. https://github.com /minimaxir /textgenrnn, 2017.