Rap Lyric Generation: A Phoneme-Based LSTM Approach

Advaith Anand, Aneesh Anand, Jitesh Maiyuran, Michael Shum

{advaith, aanand, jitesh, mshum} @mit.edu

ABSTRACT

Rap lyric generation is a task that natural language processing has yet to solve. Due to constraints such as rhyme and meter, a traditional bag of words or n-gram model does not model the necessary dependencies. We instead propose a *Long Short Term Memory* (LSTM) recurrent neural net (RNN) trained on a corpus of rap lyrics, where words are decomposed into phonemes. By training the RNN on a phoneme level, we find that rhymes are more easily learned. We also explore different padding techniques when training in order to adhere more closely to the length and meter of the training data. Using metrics that assess rhyme and flow, we find that though unable to match actual lyrics, our model was competitive with existing methods.

1. INTRODUCTION/MOTIVATION

We propose a natural language processing system to generate coherent rap lyrics that adhere to general standards of rhyme and *flow*. We define flow as the ability of the lyrics to adhere to rhythm and meter. We expect that by training our model on rappers with differing flows and vocabularies, we will be able to produce raps of differing natures.

The architecture of our language model is a character-level Recurrent Neural Network (RNN), which has seen success in replicating the styles of distinct corpuses[1][2]. A token-level model passes in a token as a prompt, and samples from the resulting distribution over next tokens. We build on this by translating words from our lyric corpus into a sequence of *phonemes*. Phonemes are the building blocks of the English language. We reason that deconstructing a word into phonemes as opposed to characters would allow the model to more easily learn rhyme and flow. In addition, we use bucketing and padding to augment the training process.

We also work with Generative Adversarial Networks (GANs)[3]. GANs train a generative model through a discriminative model that receives lyrics from the generative model and real lyrics from our data set. The discriminator attempts to distinguish between real and generated verses. By backpropagating the discriminator's output through the lyric generator, our expectation is that the generator's output would be more similar to real lyrics.

For evaluation we measure rhyme through vowel phoneme similarity; we evaluate rhyming within each line and at the end of lines. We also examine line lengths to measure flow. These metrics are similar to those used by Malmi et al., which we will expand on in the next section as well as section 4.

2. RELATED WORK

The use of RNNs for language modeling is well documented[2][5][6][7]. The papers mentioned use RNNs to generate text character by character. Both Graves and Karpathy have demonstrated success with the character-level LSTM architecture in learning correct spelling of a

variety of English words as well as grammatical and punctuation rules. They also demonstrate success in various domains, such as Shakespeare texts, XML, and Linux source code. LSTMs are able to reproduce text that follows syntactic rules in each of these domains (e.g. semicolons at ends of lines in code) with remarkable success.

Language modeling in the context of poetry and song lyrics has been explored mainly using large corpuses and heavy constraints[8][9][10]. In the context of rap lyrics, Wu et al. (2013) have had success modeling the problem of rap lyric generation as a machine translation problem, learning a system that generates rhyming response lines given an input line. Malmi et al. (2015) generate 16-line hip hop verses by using a prediction model to select the best next line from a sample of candidate lines from existing lyrics. Both of these approaches differ from ours in that they generate lyrics at the line level, while our model attempts to generate text from more granular phonetic building blocks. However, Malmi et al. have developed a useful metric of *rhyme density*, which we also use in the evaluation of the quality of our generated lyrics.

Our approach is similar to that of Potash et al. 2015, which uses a character-level LSTM to generate rap lyrics. We use an implementation of Potash's model as our baseline.

As far as we are aware, our approach is the first to use phonemes as inputs in language modeling. While phonemes have been used in the context of rhyme detection[13], we are unable to find any literature that generates text encoded as phonemes.

3. APPROACH

3.1. DATA

Our corpus of rap lyric data is pulled from Genius.com. Hoping to capture an artist's specific style, we choose to use the corpus of only one rapper's lyrics to train our model. For examples shown, the model is trained on Eminem's song lyrics data which results in a corpus of 58,405 lines of text. In order to improve training of rhymes we preprocess the input by only including series of rhyming couplets. Preprocessing also involves removing choruses (as these are present in rap songs but are not in the style of rap) and lines with less than two words. This results in our final training corpus of Eminem lyrics being 11,648 lines, or 228,318 tokens.

3.2. LSTM

While we describe our model as an RNN, the model is actually a *Long Short Term Memory* (LSTM) model. In practice, vanilla RNNs often fail to model long-term dependencies[15], giving rise to LSTMs which maintain the recurrent property of RNNs, but use multiple linear weight combinations and activations to determine state and output.

Originally proposed by Hochreiter et al., the LSTM cell has a state monitored by 3 gates (forget gate, input gate, and output gate), where each gate passes a concatenation of the input and previous state through a series of sigmoids and activation functions to determine what state to forget, what input to consider, and what state to output.

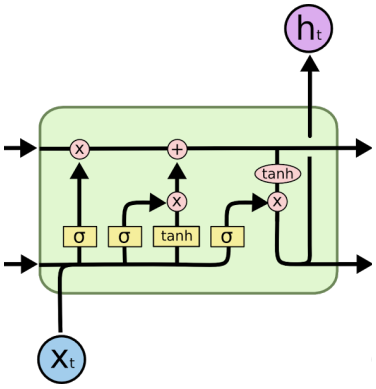

Fig 1. Each LSTM cell has an input, output, and forget gate, which are controlled by weights, biases and a sigmoid function. Training each of these values separately allows for longer term dependencies to be modeled/

In figure 1[16], we observe that the state evolves via many non-linear transformations, which allow an LSTM to model longer-term dependencies than a traditional RNN. For example, we start by combining the previous state with an activation of the previous state and input, f_t:

$$f_t = \sigma(W_t[h_{t-1}, x_t] + b_t)$$

A similar transformation happens at all three gates, where both the weights and the bias are trained via backpropagation. This complexity allows LSTMs to model longer dependencies compared to traditional RNNs.

3.2.1. BASELINE

Our baseline model is an implementation of Andrej Karpathy's character level RNN as described above. Due to the success of Karpathy's model on various data sets we expect to generate sentences with proper spelling and grammatical structure when trained on a corpus of rap lyrics. Potash et. al. apply a similar model with an LSTM for rap lyric generation, also based off Karpathy's work. We view Karpathy's work as a valid baseline as it is a successful text generation model but is not particularly tailored for rap lyric generation, especially rhyme generation.

3.2.2. PHONEME MODIFICATIONS

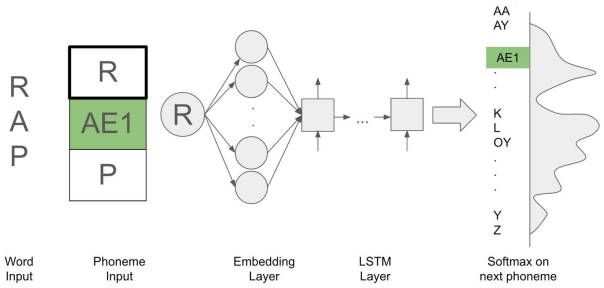

Fig 2. Architecture takes a phoneme, retrieves the embedding, passes through LSTM, then computes softmax. LSTM and embedding values are trained via backpropagation.

While we expect our baseline character-level RNN to produce lyrics that are semantically meaningful over short sequences of characters (one to three words), we expect little to no rhyme. We explain this behavior by looking at phonetic irregularity in the English language. If we consider the following line from Kanye West's *Ultralight Beam*:

You can feel the lyrics, the spirit coming in **braille**
Tubman of the underground, come and follow the **trail**

we see that 'braille' and 'trail' are dissimilar on a character level, even though they rhyme.

To resolve this issue, we propose the novel approach of a *phoneme-level* RNN. A phoneme is a unit of speech that comprises the fundamental sound of language. Using the CMU Pronunciation Dictionary, which operates on 83 phonemes and maps 133,000 words to sequences of phonemes, we decompose our lyric corpus entirely into phonemes. Using this model, when we look at the phonemes for the same words:

BRAILLE: B-R-EY1-L
TRAIL: T-R-EY1-L

we see similarity on a phoneme basis. Therefore, by decomposing words into phonemes, we are ideally able to bypass the idiosyncrasies of written language since a model would consistently see rhyme on a phoneme level. Rhymes are observed through phoneme-vowel similarity -- when two words' are identical starting at the *stressed* vowel (EY1) to the end of the word (EY1-L).

One issue we anticipate is that a sequence of phonemes outputted may not map to a word, or perhaps unintentionally map to an obscure word. However, analogous to character level RNNs producing words, we find that with sufficient training, almost all generated phoneme sequences map to phoneme sequences seen in training (i.e. relevant words). We again use the CMU Pronunciation Dictionary to map back to words.

A critical aspect of the phoneme sequence-to-word mapping is dealing with overloaded phoneme sequences i.e. homophones. For example, consider two English words:

YOU: Y-UW1
YUE: Y-UW1

In this case, 'you' is much more common that 'yue.' We resolve these collisions by using a list of the 20,000 most common words[20] to prefer the most common one. In the case that no translation is within the 20,000 most common words, we randomly select one of the suggested ones.

Finally, we accommodate for slang in rap lyrics, completing words ending with "-in" with "-ing" (i.e. "fallin'" and "falling").

3.2.3. CHARACTER GENERATION

When sampling a character from the softmax output of an LSTM, many sampling techniques exist. Selecting the highest probability token produces repetitive output while sampling from the distribution can be occasionally nonsensical. To resolve this, we select the highest probability token with some probability *P* and sample from the distribution with probability *1-P*

When training the LSTM, sequence length was also important. Sequence length is the number of phonemes we backpropagate to, denoting the range of previous phonemes a next phoneme learns from. When generating output, we begin with a prompt character and use the sampling method described above over some number of time steps i.e. 500 tokens. We then take these phoneme sequences and map them back to words using the CMU Pronunciation Dictionary.

3.2.4. OTHER MODIFICATIONS

Because meter and rhythm are essential for proper rap lyrics, we experiment with different techniques that work with length of lines. In particular, we evaluate implementations of padding/bucketing, backward input, and input splicing.

Padding the input consists of selecting a maximum line length (including phonemes and spaces), and adding **<PAD>** tokens until all lines reach the maximum length. The benefit of padding is that a sequence that takes in twice the maximum line length would include exactly one couplet. Without padding, arbitrary sections

of couplets are trained at once, which could cause us to potentially miss the end of line rhyming relationship between rhymes and the newline character. The newline character is essential because it always follows a rhyming word. One pitfall of padding is the disparity between the longest line in a corpus and the median line length, in which case padding is excessive. To resolve this, we implement bucketing, which places lines of similar length in the same batch to standardize the padding over one gradient descent step. Padding and bucketing do not offer any significant advantages, most likely because of the model's inability to learn long-term dependencies over multiple newline characters.

Noting the relationship between final syllables and newline characters, we also train our model on a reversed corpus and expect the model to learn the immediate dependence between the newline and the preceding rhyme. We find that this method improves inter-line rhyme and end-of-line rhyme.

Finally, we slice our inputs to fixed-size inputs. We find that slicing each input line *n* tokens from the last (newline) token improves our performance by making the relationship between rhyming phonemes more consistent.

3.3. GAN

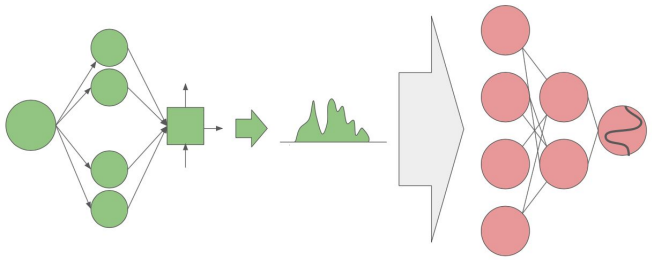

Fig 3. Architecture for GAN in which a softmax distribution is passed from a generator to a discriminator.

Having learned about Goodfellow's success with Generative Adversarial Networks (GANs) in vision[3], we hoped to utilize a similar architecture to learn semantic and syntactic features of an artist's style.

A GAN reframes the training process as a game, pitting a generative and a discriminative network against each other. In Goodfellow's original usage to generate images similar to the real world, the generative model produces an image, while the discriminative model tries to classify if inputs are from the true original distribution or the distribution from the generated image. The discriminative model adjusts both its and the generative model's weights in order to have the generator mimic the true data distribution, such that the generative model produces similar images to the dataset.

We hypothesize that a discriminative network accepting one real and one generated couplet with appropriate labels would help guide a generative network trained on a specific artist to produce similar lyrics. We plan an architecture using our modified LSTM for our generative model, and a convolutional neural network for text classification as our discriminative model.

We leverage the work of Amos[17] to help connect our existing LSTM to our discriminator and have our generator output a couplet instead of an image. Phonemes are generated by sampling from the softmax distribution of the next predicted phoneme, and appending until we find two new line tokens.

Unfortunately, we discover that the act of generating characters eliminates the ability to backpropagate the predictions from the discriminative network as discrete samples of a distribution broke the ability to evaluate gradients. Passing the distribution from the generative model is also not helpful, as the distribution is the likelihoods of single characters

and would not guide learning of the structure of couplets.

4. RESULTS

4.1. SAMPLE LYRICS

dee as your get i booter he frocka front say i'm dee
treat all tram he's lesser got doc the then it
i get better brawl be sore afar
when denny feign rip
who shot peer cause who give odd a metal place
stall ain't down be mocked me
i lookin' alleys they anti it is you're how
i wanna d*cking one act the so why ain't purse you g**k is and
i crystal get din you wheat do
even shake i call a lose it the tried
someone goose it you falls i way but been lip why who
you go duggas you foo the you but walk do
white with gonna krok a yeah duddy your brains the means be
only shady some right chee

Fig 4. 7 couplets generated from a model trained with a batch size of 10 and sequence length of 100. Sampling done from a full distribution.

We find that qualitatively that not only do ends of lines rhyme, as “be” and “chee” do in the final two lines, but also a fair number of internal rhymes are apparent, such as “lesser” and “better” in lines 2 and 3. However, we note that semantic meaning is low, as a phoneme-level LSTM also suffers from the inability to track long-term dependencies.

4.2. EVALUATION METRICS

In order to quantify the effectiveness of our models we define multiple evaluation metrics. We develop the following four metrics to measure the technical quality of a generated verse.

4.2.1: Couplet Rhyme Percent (CRP):

$$\frac{\#rhyming\ couplets}{\#lines}$$

CRP is a metric to gauge the end-of-line rhyme scheme in a verse. An ideal ratio of 0.5 would signify that the entire verse is composed of rhyming couplets, with an AABBCC… structure.

4.2.2: Internal Density (ID):

$$\frac{\#\ syllables\ involved\ in\ rhymes}{\#\ syllables}$$

ID is a way to measure the presence of internal rhymes which showcase a lyricist’s overall rhyme frequency. An ideal ratio of 1 would indicate every syllable is involved in a rhyme.

4.2.3: Rhyme Density (RD):

$$\frac{avg.\ len.\ of\ best\ per\ word}{\#words}$$

RD is a metric originally devised by Malmi et. al. who use it as a way to measure the rhyme complexity of a verse[4]. For each word in the verse the max. number of vowel syllables involved in a rhyme is calculated and this is averaged over all words. When measured on real rap verses RDs of around 1 have been found so an RD of 1 is an ideal goal for generation.

4.2.4 Flow Irregularity (FI):

$$\sigma(\#\ words\ in\ line)$$

Raps need to have similar line lengths for them to be easily spoken (‘flow’) so this metric helps quantify that. A low FI would indicate a low standard deviation in the line lengths which would indicate nearly even line lengths throughout, the goal of generation.

4.3. DISCUSSION

In order to evaluate the quality of our lyrics, we sample outputs from our model with various parameters and compare the output's performance on the previously defined metrics to the performance of the baseline model's output as well as the performance of real rap lyrics. For each of these categories, we pick 4 text samples. For real rap lyrics, we pick random verses by Andre 3000, Eminem, Kanye West, and Kendrick Lamar. For the baseline model, we generate 500 tokens of text with sequence length parameters of 25, 50, 75, and 100. For our model, we generate 500 tokens of text with sequence length parameters of 20, 70, 90, and 100. The average performance of the 4 samples in each category on the metrics is depicted in Figure 4. We note that our model is competitive with both the real verses and baseline in CRP, ID, and RD, while outperforming real verses in FI.

Our model has an average couplet rhyme percentage of 12.6%, which improves on the 8.0% CRP of the baseline model, while falling short of the 22.6% CRP of the actual rap lyrics. Our model's internal density and rhyme density measures are also comparable to the baseline. We believe that our model performs well on the rhyming tasks because of the previously mentioned ability of phonemes to encode rhyming structure that characters cannot.

Our model also outperforms the baseline in flow irregularity- a lower value indicates that different lines have similar numbers of words and thus follow some sort of rhythmic structure. We believe this is due in part to the phonetic encoding, but also because of our practice of splicing inputs discussed in section 3.2.2. Overall, we are impressed with the ability of our model to both produce words through phonetic features and perform competitively in comparison to the baseline and real rap lyrics.

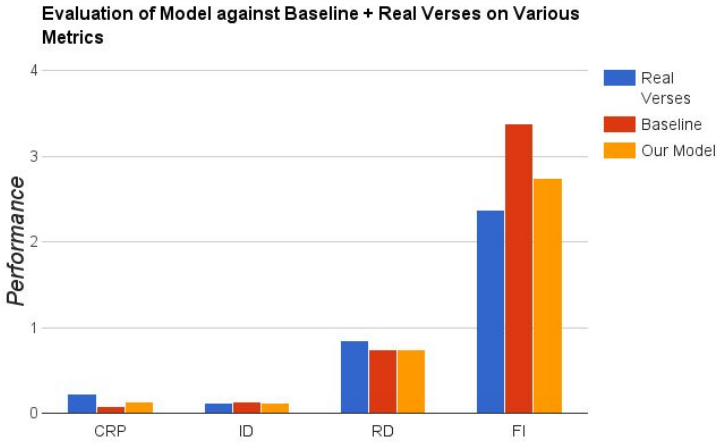

Fig 4. Our model's performance on the defined metrics in comparison to the baseline character-level model and real verses. Each bar represents an average of the metric on 4 sample verses. The blue bars represent average performance of real verses from Eminem, Kendrick Lamar, Kanye West, and Andre 3000. The red bars represent average performance of 4 versions of the baseline char-level LSTM (with sequence lengths of 25, 50, 75, and 100). The yellow bars represent the average performance of 4 versions of our model (with sequence lengths of 20, 70, 90, and 100).

5. MEMBER CONTRIBUTIONS

aanand: I worked mainly on refining the LSTM model and optimizing its performance. I ran through the training and sampling process with various parameters of batching, sequence length, and temperature. I also implemented the padding, bucketing, and masking of the inputs to allow our model to account for the variability of input sequences. I made the input clipping and reversal modifications in order to improve LSTM performance.

advaith: I spent my time working on the LSTM model and on pre-processing and evaluating the inputs and outputs, respectively. Most of the work that was spent on the LSTM was on trying to understand existing implementations and understanding which parameters could be tuned and how various current approaches could possibly be modified to introduce rhyming and structure into outputs. After working with the LSTMs we realized we needed to further preprocess our training data so I wrote a function to remove lines which weren't part of rhyming couplets to improve the rhyme metrics of our input. In addition I worked on developing the metrics to quantify the successes of different approaches. I wrote a function to do this which helped us understand where our model was successful and where it was less-so.

jitesh: I spent most of my time working on GANs and the word-to-phoneme tokenization for the RNN. Interfacing with the CMU Pronunciation Dictionary was straightforward, though due to the complications, I handled some edge cases related to slang in rap lyrics and homophone phonetics collisions by using word frequency. I also modified our character-level RNN to tokenize by phoneme instead of character. Working with GANs consisted of reading about existing success (mostly vision applications i.e. Amos), and using Tensorflow to define backpropagation between the discriminative and generative models. Determining an input to the GAN was also something we spent time researching, such as feature vectors vs direct text, as well as the architecture of the discriminator, which we eventually chose to be a convolutional neural network. Though the GAN was not part of our final implementation, it was still a significant part of our research.

mshum: I worked primarily on GANs, data-gathering, and cross-evaluation of parameters for the LSTM. I read recent GAN papers related to vision and explored implementations from various blog posts (r2rt, Amos). From this testing I learned and explained how TensorFlow creates RNN models, graphs, and sessions, as well as how GANs connect their generative and discriminative models. I then implemented our proposed GAN architecture, modifying an existing one to use our generator and a discriminator for text classification. We spent time researching and testing inputs (word/phoneme pre-trained embeddings/one-hot vectors) to our generative model. Data gathering involved scraping webpages manually with BeautifulSoup due to a lack of an API for lyrics from genius.com. Finally, for cross-evaluation of parameters I composed bash scripts to generate phoneme-level lyrics, translate these into words, and finally run metric calculations on them.

6. REFERENCES

1. Graves, Alex. "Generating sequences with recurrent neural networks." arXiv preprint arXiv:1308.0850 (2013).
2. Karpathy, Andrej. "The Unreasonable Effectiveness of Recurrent Neural Networks." The Unreasonable Effectiveness of Recurrent Neural Networks. N.p., 21 May 2015. Web. 10 Dec. 2016.
3. Goodfellow, Ian, et al. "Generative adversarial nets." Advances in Neural Information Processing Systems. 2014.
4. Malmi, Eric, et al. "Dopelearning: A computational approach to rap lyrics generation." arXiv preprint arXiv:1505.04771 (2015).
5. Sutskever, Ilya, James Martens, and Geoffrey E. Hinton. "Generating text with recurrent neural networks." Proceedings of the 28th International Conference on Machine Learning (ICML-11). 2011.
6. Graves, Alex. "Generating sequences with recurrent neural networks." arXiv preprint arXiv:1308.0850 (2013).
7. Mikolov, Tomas, et al. "Efficient estimation of word representations in vector space." arXiv preprint arXiv:1301.3781 (2013).
8. Colton, Simon, Jacob Goodwin, and Tony Veale. "Full face poetry generation." Proceedings of the Third International Conference on Computational Creativity. 2012.
9. Toivanen, Jukka M., Matti Järvisalo, and Hannu Toivonen. "Harnessing constraint programming for poetry composition." Proceedings of the Fourth International Conference on Computational Creativity. 2013.
10. Das, Amitava, and Björn Gambäck. "Poetic machine: Computational creativity for automatic poetry generation in bengali." 5th international conference on computational creativity, ICCC. 2014.
11. Wu, Dekai, Karteek Addanki, and Markus Saers. "Modeling hip hop challenge-response lyrics as machine translation." 14th Machine Translation Summit (MT Summit XIV) (2013).
12. Potash, Peter, Alexey Romanov, and Anna Rumshisky. "GhostWriter: Using an LSTM for Automatic Rap Lyric Generation."
13. Hirjee, Hussein, and Daniel G. Brown. "Rhyme Analyzer: An Analysis Tool for Rap Lyrics." Proceedings of the 11th International Society for Music Information Retrieval Conference. 2010.
14. Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." Neural computation 9.8 (1997): 1735-1780.
15. Bengio, Yoshua, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult." IEEE transactions on neural networks 5.2 (1994): 157-166.
16. "Understanding LSTM Networks." Understanding LSTM Networks -- Colah's Blog. Chris Olah, n.d. Web. 14 Dec. 2016.
17. Amos, Brandon. "Image Completion with Deep Learning in TensorFlow." Image Completion with Deep Learning in TensorFlow. N.p., 9 Aug. 2016. Web. 10 Dec. 2016.
18. Britz, Denny. "Implementing a CNN for Text Classification in TensorFlow." WildML. N.p., 04 Feb. 2016. Web. 10 Dec. 2016.
19. u/MartianTomato "Recurrent Neural Networks in Tensorflow III - Variable Length Sequences - R2RT." R2RT Full Atom. N.p., 15 Nov. 2016. Web. 10 Dec. 2016.
20. https://github.com/first20hours/google-10000-english