

Final Year Project - Rap Battle Lyric Generator

By Alex Smith

ID: 1998833

Programme Name: MSci Computer Science FT

Supervisor: Eric Finster

Word Count: 9217 (includes references and appendices)

Abstract:

Use of AI to replicate forms of human art such as song lyrics is a recent but growing trend, with rapid improvements seen in the generation of images and long-form text from user prompts. I present an end-to-end AI model capable of generating a rap battle verse of a specified length from a user prompt, which enforces rap-battle textual features using a combination of dataset pre-processing and rule-based constraints during text generation. I find that while this approach leads to tonally correct rap-battle lyric generation, a more sophisticated model trained on a larger dataset would be required to ensure higher-level grammatical and structural features of English are replicated.

Table of Contents:

1.	Introduction	4
2.	Literature Review	5
2.1.	General Text Generation Methods	5
2.2.	Approaches to Rap Lyric Generation	6
3.	Method	7
3.1.	Dataset Creation	7
3.2.	Model Design and Implementation	10
3.3.	Pronunciation and Rhyme Detection	12
3.4.	Text Generation	15
4.	Evaluation	20
4.1.	Approach to Evaluation	20
4.2.	Evaluating Tone	20
4.3.	Evaluating Technical Quality: Rhyme Density	21
4.4.	User Survey: Rationale and Approach	22
4.5.	User Survey: Results and Discussion	23
5.	Discussion	26
6.	Conclusion	27
7.	References	27
8.	Appendices	29
8.1.	Example rap battle lyrics generated by GPT-3.5	29
8.2.	Full results of User Survey	30
8.3.	Sample Model Outputs	32

Section 1. Introduction:

Rap battles are a form of musical performance whereby two or more competitors (MCs) recite their verses, which incorporate insults, boasting, and clever wordplay. The battles are often performed live in front of an audience, although ‘battle rap’ can also be recorded in a studio like other genres of music (Edwards, 2009, pp. 25–27).

Rap (and more generally, song) lyric generation using machine learning techniques is an active research area within the more general field of text generation. The problem of generating lyrics presents interesting challenges - unlike most text, lyrics are intended to be spoken or sung aloud, meaning that pronunciation, rhyme and rhythm are important considerations, alongside the substance of the text that is generated. Rap battle lyrics, as a subset of rap lyrics, have additional traits: in particular, the insulting tone and the fact that the lyrics are usually directed at an opposing MC.

A variety of approaches to rap lyric generation exist. (Malmi *et al.*, 2015) have viewed rap lyric generation as a search problem, selecting a next line from a list of candidate lines. (Wu and Addanki, 2015) have used a bilingual neural network to generate a ‘response line’ to a rap battle challenge line. I could not, however, find any examples of research generating a complete set of rap battle lyrics.

The rap battle lyric generation problem specifically has an extra issue in the form of training data. Due to the niche nature of the genre, and the fact that rap battle lyrics are usually improvised and therefore not written down, there is a lack of easily accessible datasets of rap battle lyrics, or even platforms which host easily-accessible transcriptions of individual verses. Any machine learning approach will need to solve this data problem if it is to succeed.

In this project, I attempt to create a model capable of generating a complete set of rap battle lyrics by taking an unusual approach. Firstly, I solve the dataset issue by using lexical and sentiment analysis methods to curate a broader dataset of rap lyrics into a dataset of lines which mirror the properties of rap battle lyrics. And secondly, I decline to teach the model to recognise spoken elements of lyrics such as rhyme and syllables, instead using hard-coded rules and helper classes in the post-training text generation process in order to ensure the output of satisfactory lyrics.

Section 2. Literature Review and Related Work

Text Generation is a very large problem area with a lot of existing literature and research, ranging from general text-generation methods to specific problems such as generating rap lyrics. In this section I will provide an overview of existing literature, products and methods that may be relevant or even useful to the project.

Section 2.1. Generalised Text Generation Methods

Before addressing research into rap-lyric generation as its own problem, I will outline existing research into generating text from a user prompt. Although it is a much more general class of problems, lyrics are ultimately just text and so this research is still relevant to my project.

An early method for generating text in this manner is the recurrent neural network (RNN). It is difficult to find who to credit for their creation, but the work of Rumelhart et al, in particular *Learning Internal Representations By error Propagation* (Rumelhart, Hinton and Williams, 1985), and *Learning Representations by Back-Propagating Errors* (Rumelhart, Hinton and Williams, 1986) are often cited by authors when referencing RNNs. An RNN is intended to operate on sequential data, for example a sequence of words. At any given timestep, the state of the RNN is determined by a function of the current input, and the *hidden state* from the previous timestep. In this way, information from each word in the input sequence is incorporated into the final output (see the below diagram). (Goodfellow, 2016, chap. 10) This preservation of information is useful for the lyric generation task, since the next word to be generated depends on the context of all the prior words, not just the word's immediate predecessor.

In the context of text generation, the input sequence would be a sequential series of words, and the model would predict the next word.

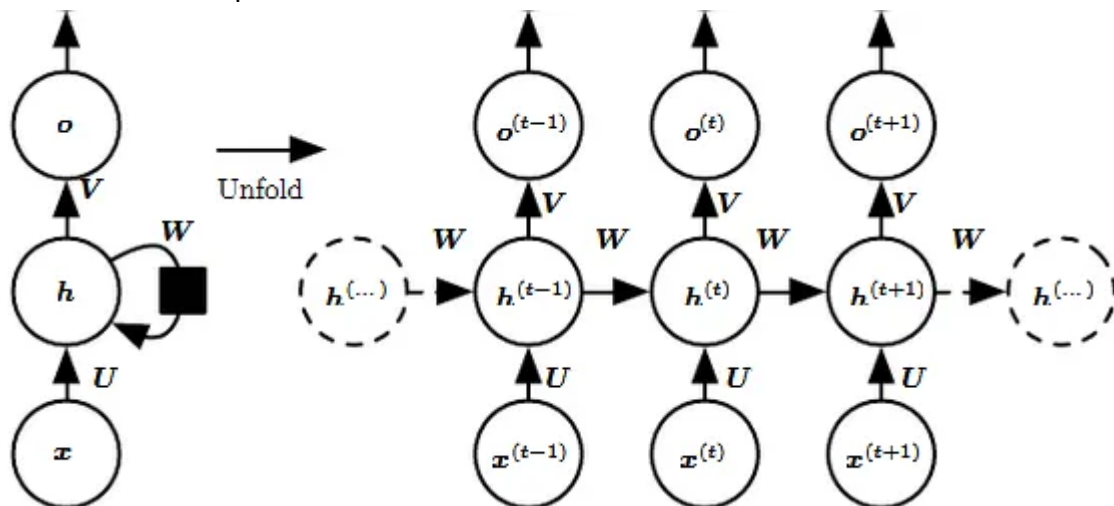


Image taken from *Deep Learning* (Goodfellow, 2016, fig. 10.3)

A further development of the recurrent neural network is the *Long Short Term Memory* (Hochreiter and Schmidhuber, 1997). This type of RNN solves the exploding/vanishing gradient problem (Bengio, Simard and Frasconi, 1994), an issue with traditional RNNs where they struggle to learn dependencies over a large number of timesteps. This allows LSTMs to be trained to learn more long-term dependencies between sequential data (e.g. words at the start and end of a sentence). This is invaluable for song and rap lyrics, which are frequently

clever, intricate, and have long term structural features such as rhyming with prior words.

Existing general-purpose text generation models

On an applied level, recent progress in powerful general-purpose text generation models and chatbots (such as *GPT-3.5*, *GPT-4* and Google's *Bard*) are worth discussing. Although these models are proprietary, and details of more recent models and their implementations are not available, I do know that GPT-3 has 175 billion parameters and was trained on a dataset of 300 billion tokens (Brown *et al.*, 2020). These kinds of general-purpose models are of limited use for a research project because I simply do not have access to the kind of hardware required to train a model of that size. However, they bear mentioning because their outputs are (personal opinion only) very convincing. (See appendix 8.1 for an example set of rap battle lyrics generated by GPT-3.5). As larger and more powerful general-purpose models are developed in the future, it seems fair to assume that performance on tasks across the board (including rap battle lyric generation) will also improve.

Section 2.2. Existing Research in the Field of Rap Lyric and Rap Battle Lyric Generation

There is limited existing research into the problem of generating rap battle lyrics, or even generating rap lyrics more generally.

(Malmi *et al.*, 2015) contribute an information retrieval approach to generating rap lyrics by representing the lyric generation problem as a search problem, seeking to select the next line of a rap from a list of candidate next lines. They also contribute a useful quantitative metric for evaluating the technical quality of rap lyrics, known as *rhyme density*. Further, they demonstrate the validity of this metric, by comparing scores using the metric to technical scores assigned to lyrics by a professional rapper. The resulting hypothesis test shows that rhyme density 'adequately captures the technical quality of rap lyrics'.

(Wu and Addanki, 2015) contribute a method for generating a rap battle response line for a given challenge line in *Learning to Rap Battle with Bilingual Recursive Neural Networks*. They train a *Bilingual* recursive neural network, which aims to 'translate' a challenge line into a response line. Notably, this is the only paper I could find concerning rap *battle* lyric generation in particular. Even here they are solving a different problem: generating a single response line to a human-supplied challenge line, as opposed to a verse of lyrics from a prompt. However, the generated lines rhyme and are fluent compared to past approaches of a similar nature, features I would like to emulate. While their solution may not be particularly applicable to my problem, it felt appropriate to include the rap-battle specific paper in the literature review.

A multitude of approaches to the more general task of song lyric generation exist, such as using markov models (Navarro-Cáceres *et al.*, 2020), and the creation of semi-automatic songwriting tools (Pudaruth, Amourdon and Anseline, 2014). None of these approaches are really useful as they are too different to the methods I intend to use. They are nevertheless worth mentioning to demonstrate the breadth of possible approaches when addressing this type of problem.

Section 3. Method

Problem Definition and Overview of Main Challenges

The aim of this project was to create an AI model capable of generating a verse of rap battle lyrics from a user-input prompt. The main problems I encountered are detailed below. These problems (and their solutions) will also form the main subsections of the method section:

- **Data:** How to either find or create a dataset of rap battle lyrics in order to train a model.
- **Model:** How to decide on the model's architecture and hyperparameters.
- **Rhyme and Pronunciation:** How to allow the model to pronounce words and detect rhyme, in order for it to generate rhyming text.
- **Text Generation:** How to generate a verse of text using a model which predicts the most likely next word.

Section 3.1 Dataset Creation

In an ideal world, there would be a ready-made dataset of rap battle lyrics freely available on the internet for me to download and get to work with. However, after some searching, I realised there were no suitable datasets available. This meant I would have to either manually create my own, or come up with a computational solution.

I decided that manually creating a dataset would take too long given the timescale of the overall project. Instead, I decided to utilise natural language processing methods to computationally curate a rap battle lyric dataset from a larger dataset of general rap lyrics.

Starting Dataset

I found a dataset containing lyrics from various US artists (Paupier, 2018), which totalled 172,800 rap lines. (This dataset will be referred to as the 'rap lyrics dataset', or simply the 'rap dataset'.) The rap lyrics dataset consisted of text documents, each one containing the lyrics from all the songs by a particular artist. The overarching idea was to filter the lines in this dataset, so that only the lines which matched the properties of rap battle lyrics were included. To do this, I first needed to explicitly define the criteria for being a 'rap battle line', and then find an approach to check if each line in the dataset met these criteria.

Defining Rap Battle Lyrics

As stated in *How to Rap* (Edwards, 2009), rap battle lyrics incorporate insults, boasting, and clever wordplay. While clever wordplay is quite subjective to try to determine in a quantitative manner, insults and boasts are more straightforward. Therefore, I chose to filter the rap dataset to include lines which were either insults or boasts.

Defining Insults and Boasts

I defined 'insults and boasts' myself with the dataset and sentiment analysis in mind, rather than using an academic definition. I made this choice on the basis that the curated dataset does not need to be perfect, as long as the majority of the lines in it meet these criteria. Additionally, I did not want to deviate too far from the main goal of the project by researching the linguistic criteria of what constitutes an insult or a boast. Time was also a consideration here - I did not want to spend too long creating a dataset, given that the rest of the project

needed to be completed by the hand-in date.

With that in mind, I defined insults and boasts as follows:

- Insults: lines containing **second person pronouns** which also have a **negative sentiment**
- Boasts: lines containing **first person pronouns** which also have a **positive sentiment**

I chose these criteria because they are easy to check for - I can check if a string contains any of a list of pronouns straightforwardly, while sentiment analysis models which check for positive/negative sentiment are included in mainstream natural language processing libraries (see below).

Use of Pre-existing Library: NLTK.sentiment.vader

Natural Language Toolkit (NLTK) is a widely-used collection of python libraries which perform a variety of NLP-based functions. For the purposes of evaluating the sentiment of each line in the rap lyrics dataset, I will be utilising the `SentimentIntensityAnalyzer` class from the module `NLTK.sentiment.vader`.

Vader (Hutto and Gilbert, 2014) is a model which performs general sentiment analysis on text. It is rules based rather than using machine learning methods - each word in a sentence is assigned a predetermined numerical score (between 4 and -4), and the overall sentiment of a sentence or line is the sum of all the individual word scores in that line, which is then normalised to lie within the interval [-1,1]. This score is also subjected to modifiers depending on use of punctuation, use of key words and bi/tri-grams, and use of all caps. A score of -1 is the maximum negative score, while 1 is the maximum positive score.

I chose to use this class because it was free to use, easy to access and add to my project directories, and because using the class in the code was also straightforward - you simply pass a string to the function `SentimentIntensityAnalyzer.polarity_scores()`, and it returns the sentiment value to you. Since I only want to check each line for a positive/negative sentiment, the module is sufficient for this task.

Programming the Dataset Helper Class

I programmed a helper class named *TextDocTokeniser* in order to help in processing the dataset. The class has attributes for arrays of first and second-person pronouns (although NLP libraries such as NLTK support part-of-speech (POS) tagging, they do not distinguish between the first and second person).

Text from a text document is stored in a 'text' attribute, which is an array of strings, each one representing a line from the current rap lyrics document. The class performs a variety of un-remarkable text file operations and string manipulation methods over the array of lines, including but not limited to:

- Opening a text file and reading each line into the text attribute array
- Converting between a one-dimensional array of lines and a two-dimensional array of words
- Removing punctuation, trailing/leading whitespace, blank lines and unrecognised characters

- Writing the contents of the text attribute to a file with specified path

There are some methods related to pre-processing the complete works of Shakespeare, which (as a collection of plays) contained stage directions and text indicating who speaks which line. These are not used as the Shakespeare dataset was not ultimately used.

The main interesting methods in this class are the ones which actually filter the text based on sentiment. The pseudocode for this method is below (assuming the current text document to be processed is stored in the text attribute):

```
first_person_lines = [line for line in text where line contains first person pronouns]
second_person_lines = [line for line in text where line contains second person pronouns]

boasts = [line for line in first_person_lines where sentiment > positive_threshold]
Insults = [line for line in second_person_lines where sentiment < negative_threshold]

output_lines = insults + boasts
for line in output_lines:
    write line to rap_battle_dataset
```

The square brackets denote list comprehensions - in Python I implement these using Python's filter function, which takes an iterable and a boolean function, and returns the entries of the iterable which the boolean function maps to True. There are helper methods I define within the class to make the list comprehensions tidier:

- Pronoun checking methods, which search a given sentence for each pronoun in an array of pronouns and return True if one is found, False otherwise.
- Sentiment checking methods, which call the SentimentIntensityAnalyzer's polarity_scores method, and returns True if the compound score is above the positive/negative threshold for a boast/insult respectively.
 - Compound is the sum of the positive, negative and neutral polarity scores, which is then normalised to be between -1 and 1. It is used to get the overall sentiment of the sentence in a single value.

Building the Dataset

I created a new python file, *Dataset_Builder.py*, which iterated through the path names of each text document in the rap lyrics dataset, creating an instance of the TextDocTokenizer class, filtering the lines from that text document and adding the accepted lines to an (initially empty) text document named insults_boasts_final.txt

The resulting text document was roughly 8000 lines long, which was smaller than ideal but enough to train a model on. Due to the time constraints of the project I felt it was time to move on to designing the model itself.

Section 3.2. Model Design and Implementation

With the dataset complete, the next step was to design a model to train with the dataset. While many approaches exist (for example the IR approach (Malmi *et al.*, 2015)), I was quite limited by the scope of my data - 8000 unordered, unlabelled lines of text. These lines did not rhyme with each other or flow into one another particularly, since they were non-adjacent lines in the original unfiltered dataset. For this reason, I decided to attempt to keep the model architecture simple as far as possible, and enforce certain features during the text generation process instead (as opposed to training the model to automatically replicate those features, using a dataset that by and large did not contain them).

Below I will outline how I implemented the model in Python. I will also describe the architecture of the model, and discuss the model's training and hyperparameters.

Use of External Libraries - Keras

The model was implemented in Python using Keras, a high-level deep learning API which runs on top of TensorFlow, a very well known and widely used set of machine-learning libraries and tools.

The main reason to use Keras was convenience - as a widely used library it is very well documented online, and it is compatible with Python. Additionally, as a high-level API to Python, it allows for fast construction of models - this was important since my approach meant there was lots still to do after the model was trained. Due to the simplicity of my model, I did not have to deviate very far from textbook use cases of Keras classes.

The two main relevant classes within Keras are Layer and Model. A Layer in keras roughly corresponds to a 'layer' in a neural network. Each Layer has a method which maps an input tensor (TensorFlow's multi-dimensional array class) to an output tensor. The Layer will also maintain an internal state determined by object variables, which represent the weights in a neural network. There are a plethora of built-in Layer classes included in the Keras.layers package, although it is possible to create custom layers also.

The Model class allows for the grouping of Layer objects into a single object, which has built in functions for training and making predictions. I use a Sequential model, which is a Keras class used when each Layer has one input tensor and one output tensor.

Model Architecture

In this section I will outline the model's architecture and any relevant decisions regarding hyper-parameters.

Diagram of a model pass, assuming batch size of 1 (as it is during text generation)

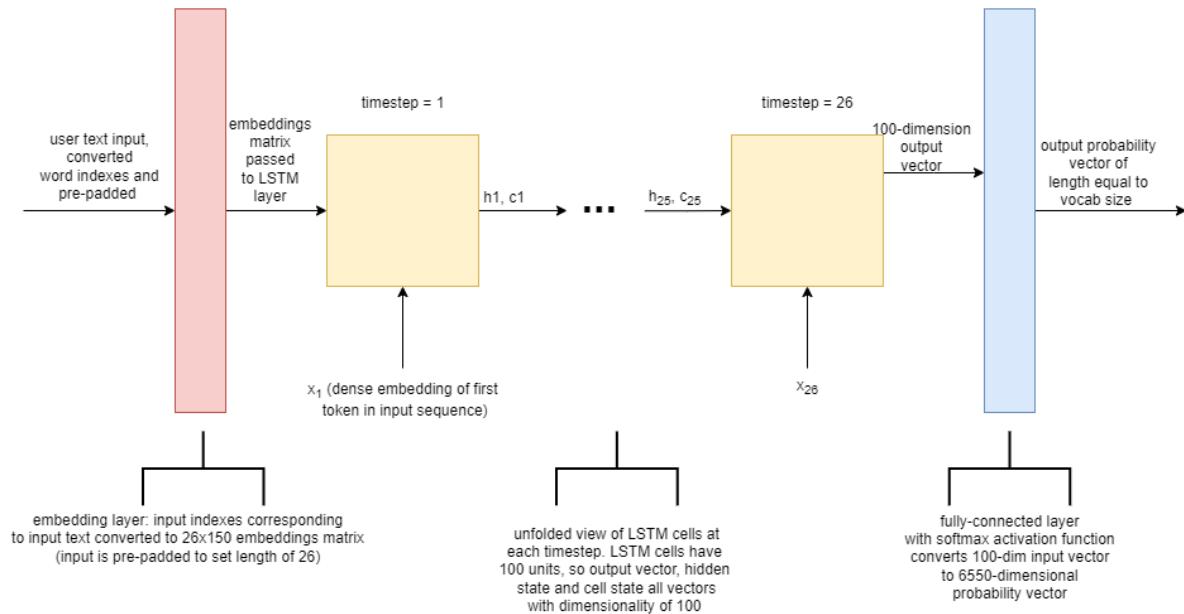


Figure: visual overview of model architecture

As a sequential model, the output of each Keras Layer feeds into the input of the next one. I will now detail each layer, in order.

- **Embedding layer:** Converts word indexes into dense vectors of a specified dimension. The input shape is (batch_size, input_length). The input length is determined by the length of the longest line in the dataset (all other lines are pre-padded to become the same length). The longest line in the insults_boasts dataset has 27 words, so the input length is 26 - since during training, the last word in the sentence is not input as it is the label, and the rest of the line is the feature. The output from this layer is a 3-dimensional tensor, with shape (batch_size, 26, 100)
- **LSTM layer:** a Keras implementation of the LSTM unit (Hochreiter and Schmidhuber, 1997) detailed in the literature review. At each timestep the LSTM layer will receive a dense embedding for the current word, and after all timesteps the LSTM will output a vector with dimensions (batch_size, 100). 100 is the size of the hidden and cell state vectors in each LSTM cell - I found this led to reduced loss compared to lower dimensionalities.
- **Dropout layer:** I add this after the LSTM layer in order to prevent overfitting. This layer is only active during training - input units are randomly set to 0 with given frequency during each training step. I found that a dropout rate of 0.2 led to the lowest loss with other parameters kept constant.
- **Dense layer:** a fully connected layer, which takes my 100-dimensional output vector from the LSTM layer and outputs a vector with dimensionality equal to the number of

unique words in the vocabulary (6550). The activation function used is the softmax activation function, which ensures that the sum of the output vector is 1 (each individual vector element is normalised to ensure this). This output vector represents the probability of each word in the vocabulary being the 'next word' given the input sequence.

Model Training

The next step after creating the model was to train it. The samples were each line in the text dataset **except for the last word in the line**. I converted them to word indexes using Keras' Tokenizer class and pre-padded them to a uniform length of 26. The labels were the last word of each line, which were again converted to word indexes, and then converted from indexes to a binary class matrix for compatibility with the chosen loss function.

I used the Keras Model.fit method to perform training with the following parameters:

- I chose categorical cross entropy loss for the loss function, since I was performing multi-class classification, and since the model returns a probability for each class (word in the vocabulary).
- I used Adam (Kingma and Ba, 2014) as the optimiser, since it has been demonstrated to outperform other comparable optimisers.
- I trained for 100 epochs - this was determined through trial and error, and was the number of epochs around which the loss stabilised and began to oscillate above and below a constant value.
- I opted for a batch size of 32. This is a smaller batch size than usual, but since I had a smaller dataset I needed to update the gradient after fewer samples, since there were fewer samples in the dataset overall.

Section 3.3 Learning Pronunciation and Enforcing Rhyme

At this point, the model had no ability to make lyrics rhyme, and had absolutely no concept of pronunciation. As stated earlier, I aimed to introduce these features using a rules-based approach during text generation. In order to do this, I needed to program a class capable of performing pronunciation-based tasks such as pronouncing strings and checking if words rhyme.

The CMU Rhyming Dictionary

The CMU rhyming dictionary (Carnegie Mellon University, 2023) is an open-source pronunciation dictionary for American English words. There is a Python wrapper class, *cmudict*, which can be installed through pip. I only actually used one function from it, the *.dict()* function, which imports the whole rhyming dictionary into a Python dictionary, such that:

- The key is the American English word, in lowercase.
- The value is the phonetic pronunciation, as a 2d array of phonic strings.

This dictionary is stored as an attribute of the Pronouncer class (see below) upon instantiation. Once this has been done, the dictionary can be interacted with as a standard Python dictionary.

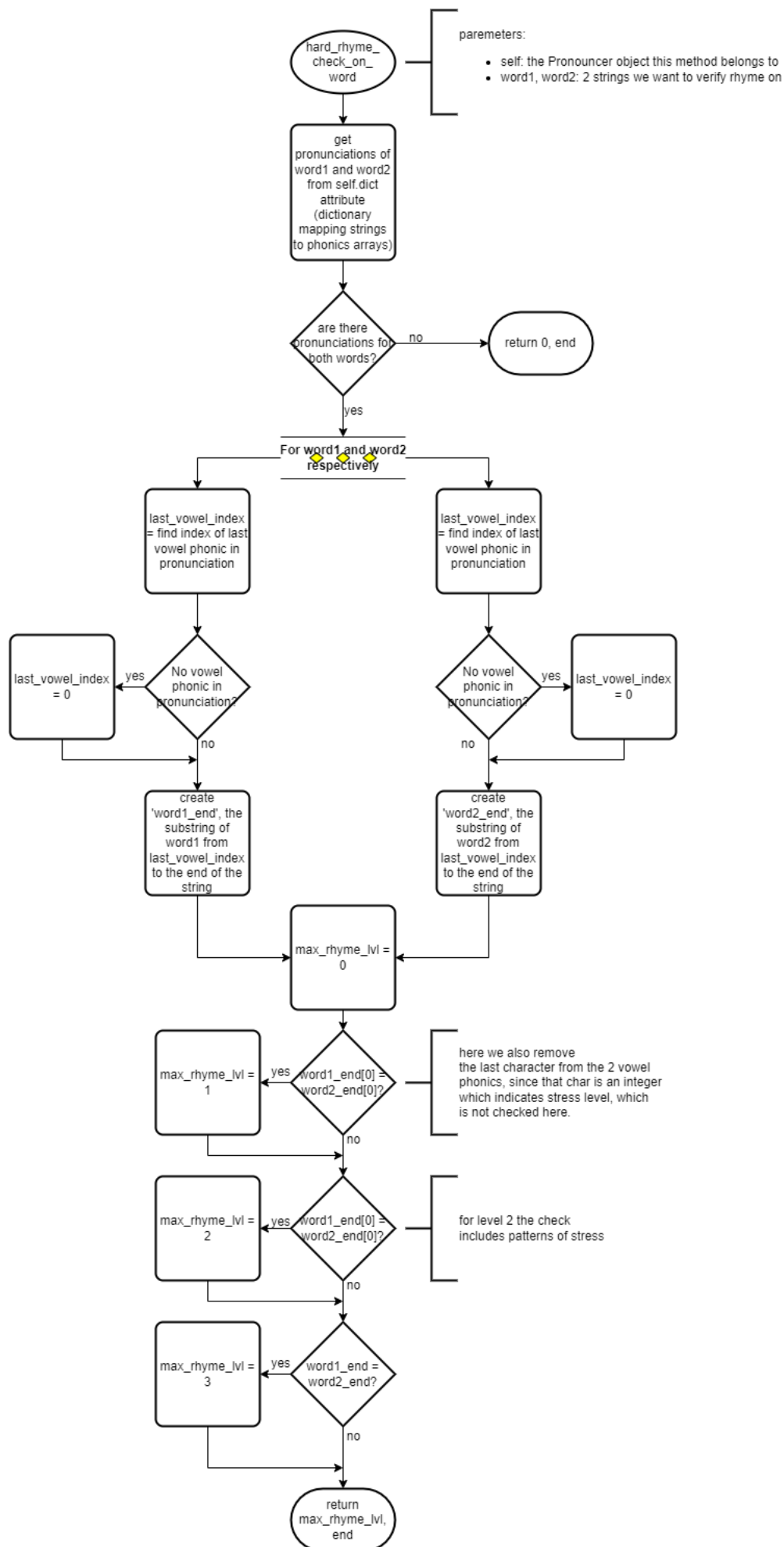
The Pronouncer Class

I programmed a class, named Pronouncer, which handles methods related to the pronunciations of various strings. This class would then be invoked by text generation methods when needed. The class only has one attribute, *dict*, which is a Python dictionary mapping lower-case strings of words to their pronunciations. In the CMU dictionary, some words have multiple pronunciations (for example, you could say 'again' as 'a-genn' or 'a-gain'). The dictionary returns both of these pronunciations (this is why the dict returns a 2d array of phonics and pronunciations). I chose to keep this feature in our pronunciation method - you assume rappers want their lyrics to rhyme and can therefore choose a pronunciation which benefits the performance.

The Pronouncer class has mainly unremarkable methods:

- Get the pronunciation of a word - returns 2d array of pronunciations, or an empty 2d array if word not in dictionary
- Get the pronunciation of a line - returns a 3d array of pronunciations - an array of the outputs of the word pronunciation method for each word in the line
- Count the syllables in a word - returns an integer equal to number of syllables (defined here as the number of vowel phonics)
- Count the syllables in a line - sums the number of syllables in each word over a line

The main noteworthy method in the class is the `hard_rhyme_check_on_word` method, which checks whether two words rhyme. I define two words as rhyming if the last vowel syllable in both words are the same. Below is a flowchart detailing the implementation of the method:



The Method assigns an integer level to the pair of words, which indicates ‘how much’ they rhyme (so that ‘better’ rhymes can be privileged over ‘worse’ ones):

- 0: does not rhyme.
- 1: The last vowel syllable in both words is the same.
- 2: The last vowel syllable in both words is the same and also has the same stress pattern on it.
- 3: The last vowel syllable in both words is the same and all subsequent consonant syllables are also the same.

The method in Python is slightly more involved because of the fact that some words have multiple pronunciations. I compared each combination of pronunciations of the 2 words. The highest rhyme level from amongst these combinations is the value returned by the method. There were also edge cases where words had no vowel syllables in them. The number of these cases were low, so as a fix I listed the last consonant as the last vowel syllable.

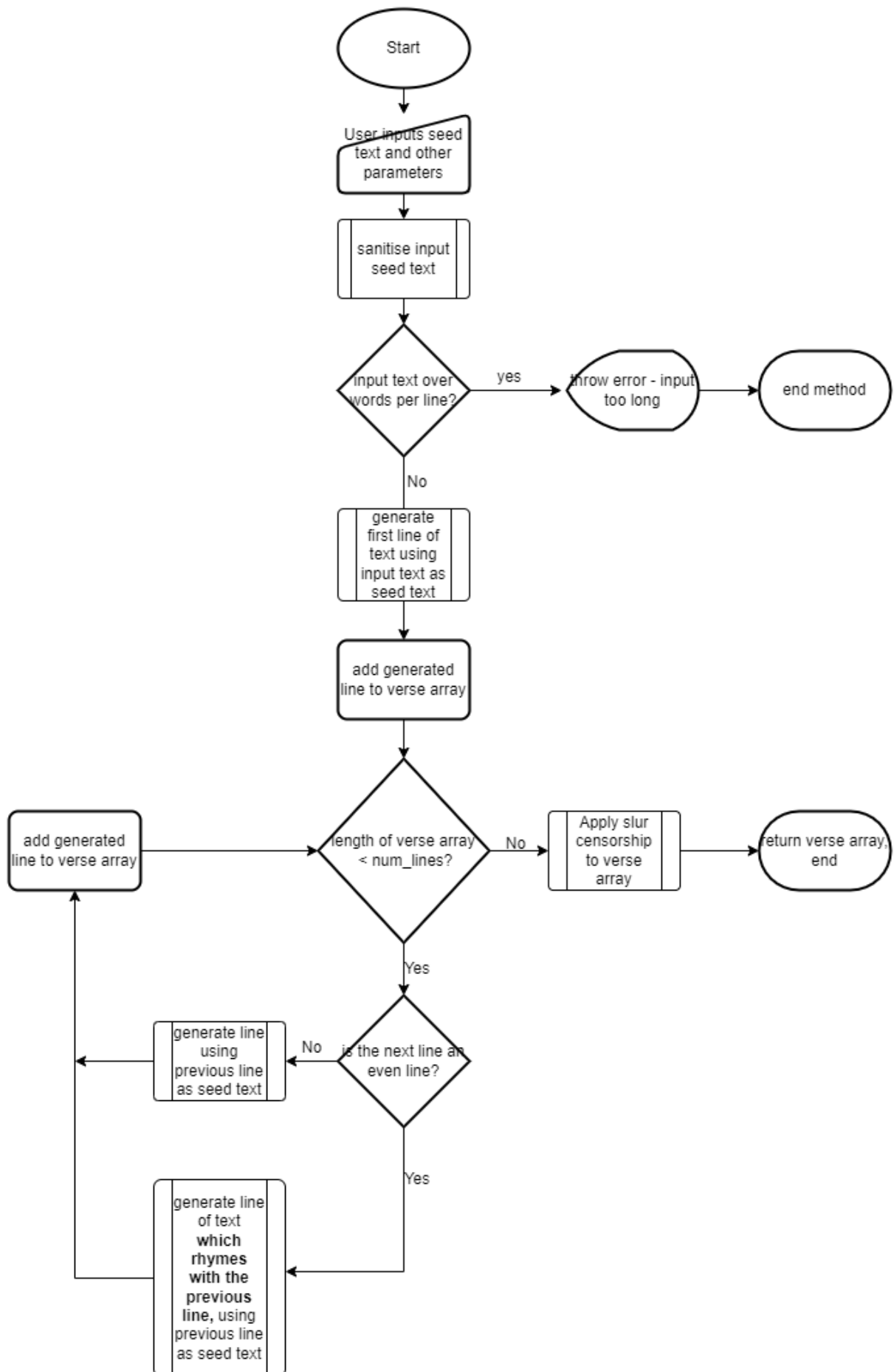
Section 3.4 Text Generation

Now I had a class capable of checking rhyme and pronunciation, I could use it alongside the model to generate text.

When a word index sequence is input into the model, it outputs a probability vector, where the highest probability is the index of the predicted word. From this, I created a method which generates a verse. I utilised a line generation method from within the verse generation method to handle the generation of each individual line.

The verse generation method has 2 modes, word mode and syllable mode. The mode is determined by a boolean flag (which is a parameter to the verse generation method) which determines whether each line has a set number of words, or a set number of syllables.

Below is a flowchart outlining the function of the verse generation method. All the flowcharts in this report were created before the methods were capable of generating lines of a set number of syllables. However, the 2 modes differ only on the level of line generation. These differences will be explained when I explain line generation methods.



No text is actually generated within the verse generation method. Instead, it calls the line generation method with parameters passed down from itself, in particular:

- Whether the next line should rhyme or not
- How many syllables/words the next line should have

Each generated line is added to a verse array. The method returns an array of strings, where each string is a line of the generated lyrics.

Slur Censorship

The verse generation method filters the generated verse for slurs and replaces them with a censored version. I felt it was important to implement this feature given the nature of the training dataset. It is not necessarily a criticism of the original artists but I did not feel it was appropriate to program a model which outputs these words - I would not say them out loud and I do not feel comfortable making a program which says them either.

With that said, I will now describe the slur censorship method (named 'cleanse_slurs'). There is a text file in the project directories which is opened by the method and used to create a dictionary consisting of rows of items: (slur, censored version). The only parameter is the input lyrics array, which is an array of strings. The below pseudocode describes the method's implementation:

```
Initialise dictionary of banned words and their censored version
clean_lyrics = []
For line in lyrics array:
    temp_line = line
    For key, value in dictionary:
        If key in line:
            Replace occurrences of key in line with value
    Add temp_line to clean_lyrics
return clean_lyrics
```

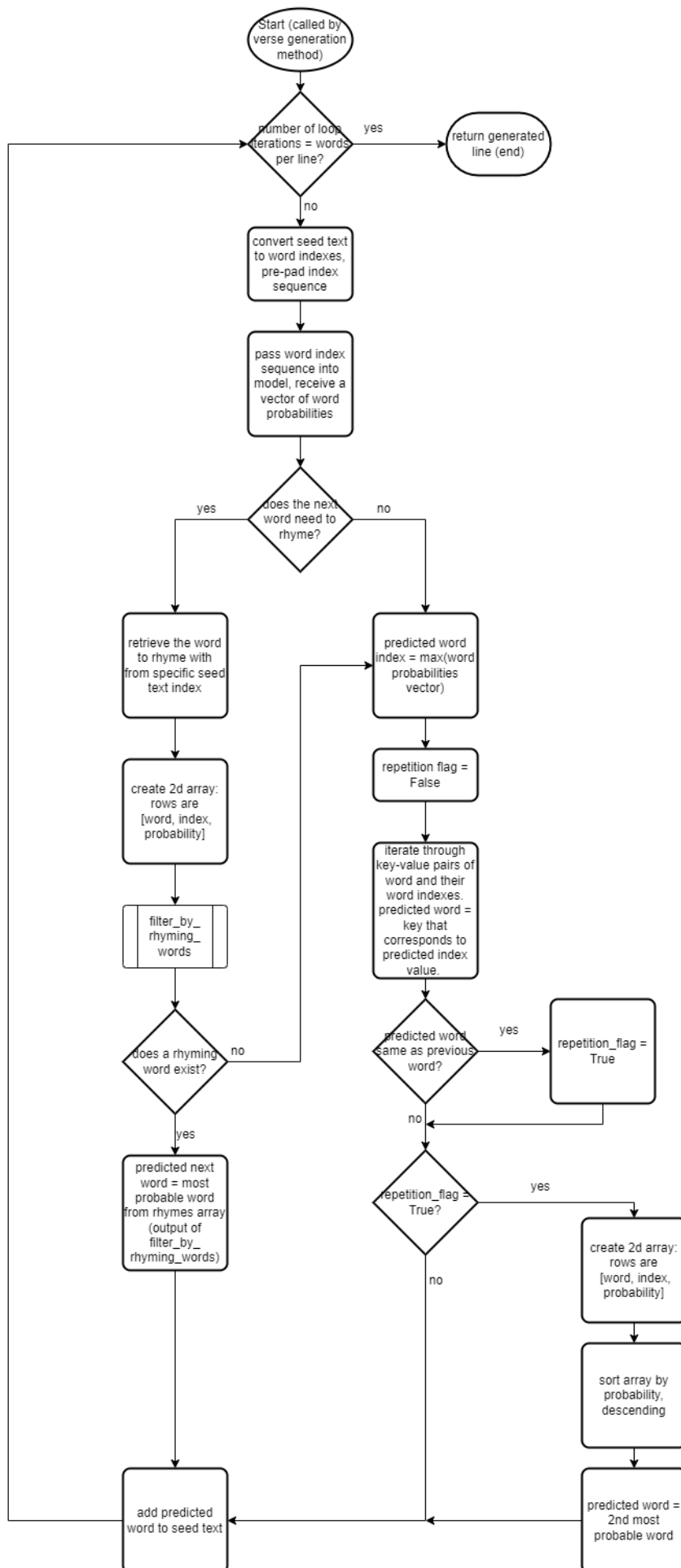
The censored equivalents have asterisks blocking out portions of the word. I felt that having the banned words stored in an external file was preferable to having them in the source code, where you (the marker) would have to look at them.

Line Generation

The majority of the work in generating text is performed by the generate_line method. There are actually two generate_line methods, one which generates a line with a set number of *words*, and one which generates a line with a set number of *syllables*. The two methods are almost identical, however there are some differences:

- generate_line_words predicts the next word within a for loop until the pre-defined number of words are generated, whereas generate_line_syllables predicts the next word in a while loop until the total number of syllables has exceeded or met the specified number
- generate_line_syllables utilises an additional method from the Pronouncer class, which counts the number of syllables in a string (by summing the number of vowel syllables in all of the constituent words in the string)
- generate_line_words includes the user's input text in the first line of the output lyrics, whereas generate_line_syllables does not.

Below is a flowchart detailing the flow of generate_line_words:



Explanation of the *generate_line* Method

The input text that the model uses to generate text is known as the seed text. For the first generated line in the verse the seed text is the user's input, and for all subsequent lines it is the previous generated line.

The generated words are added to seed text as they are generated, so that the next prediction can take into account the previous words of the current line. Despite this, the method only returns the newly-generated words as the current line (this is straightforwardly achieved by counting words).

For each word to be generated, the model converts the current seed text to word indexes and pre-pads them as necessary to be the desired length of 26, ready for passing into the model.

The model outputs a tensor of probabilities, where the index of each probability within the tensor corresponds to the word index of each word in the vocabulary.

In most cases, the predicted word is the word corresponding to the highest probability. This word is then added to the current seed text and the loop restarts. However, there are two special cases where a different word is selected:

1. Case of word repetition: the model had a tendency to predict the same word repeatedly in a row. To combat this, I explicitly disallowed it from predicting the same word more than once in a row. In cases where the most probable word is also the previous word, the method predicts the second most probable word instead.
2. Case of rhyming: when the current line being generated is an even line number, and when the next word to be generated is the last word of the current line (or in the case of *generate_line_syllables*, when there are 2 or fewer syllables remaining before the syllable target is reached), the next word predicted will be a rhyming word. The approach to this at a high level is to select the most probable word *from the subset of the vocabulary that rhymes with the last word of the previous line*. This is accomplished by the *filter_by_rhyming_words* subroutine, which I will explain below:

The *filter_by_rhyming_words* Method

This method takes a two-dimensional array consisting of rows of [word, index, probability], and the last word of the previous line (*word_to_rhyme*). It returns a filtered version of this array containing only the rows where word rhymes with *word_to_rhyme*. The array is also sorted in order of descending probability, so that the predicted word is in the first row of the array.

The rhyming is determined using the *hard_rhyme_check_on_word* method from the Pronouncer class.

When the array is returned to the relevant *generate_line* method, the word in the 0th index of the array is chosen as the predicted word. If the array is empty (i.e. there are no words which rhyme) then the method reverts to the default case, and the most probable word from the entire vocabulary is chosen.

Section 4. Evaluation

Section 4.1. General Approach to Evaluation

In this Section I will evaluate the successes and shortcomings of my approach. Due to the lack of available rap battle lyrics for comparison, I will evaluate components of the model's output individually, before finishing with a user survey to give a holistic summary of the approach's strengths and weaknesses.

The evaluation will focus on the quality of the generated text rather than functionality - I already know that the model *functions* in the sense that it outputs a set of lyrics. Instead I focus on assessing the quality of said lyrics and if the overall aims of the project have been achieved.

Section 4.2 Evaluating Tone using Mean Absolute Sentiment Intensity

To evaluate the tone of the generated lyrics, I calculate the **mean absolute sentiment intensity** (using the Sentiment_Intensity_Analyser (Hutto and Gilbert, 2014) module). This metric takes the absolute of the compound polarity score of each line in the generated lyrics, and calculates the mean. Taking the absolute of scores is necessary because negative sentiment is represented by a negative float score, while positive sentiment is a positive float.

I wanted the model to emulate the insults and boasts within rap battle lyrics, which are quite extreme in their sentiment - boasting is being overly positive about oneself, whereas insults are obviously negative. Therefore, a high score in this metric indicates that the tone of rap battle lyrics has been emulated by the generated lyrics.

Below is a table displaying the mean sentiment intensity over 10 runs of the model, using set numbers of words and syllables per line. Since Vader sums the sentiment scores of each word in a sentence, longer lines will have a stronger sentiment intensity on average. For each set of 10 tests, I use a variety of seed texts:

- Boastful seed text: 4 times
- Insulting seed text: 4 times
- Neutral seed text: 2 times

I evaluated sentiment intensity over a range of line lengths to get a broad view of the model's performance under differing parameters. In all cases I generated an 8-line verse, sufficient to minimise the impact of anomalous scores from individual lines.

For comparison, I included the scores for the metric over each line of the insult_boasts dataset, as well as the original rap lyrics dataset, in order to provide a baseline for comparison.

All results given to 4 significant figures

Seed Text	6 words per line	9 words per line	12 words per line	8 syllables per line	12 syllables per line	16 syllables per line	Insults boasts dataset	Rap lyrics dataset
Boast 1	0.3725	0.3787	0.5824	0.4208	0.5092	0.6003		
Boast 2	0.2731	0.4464	0.5579	0.4122	0.5579	0.5944		
Boast 3	0.4461	0.3577	0.4719	0.4537	0.5013	0.5853		
Boast 4	0.2568	0.4330	0.5853	0.3465	0.3612	0.6562		
Insult 1	0.3844	0.4923	0.6449	0.3557	0.4711	0.5093		
Insult 2	0.1535	0.3234	0.3889	0.3590	0.6125	0.6142		
Insult 3	0.3816	0.3742	0.5654	0.3252	0.5963	0.4835		
Insult 4	0.3958	0.3450	0.7690	0.3699	0.5975	0.6274		
neutral1	0.2576	0.3859	0.4577	0.4999	0.3600	0.4166		
neutral2	0.3098	0.3977	0.5306	0.5375	0.4729	0.7573		
average	0.32312	0.39343	0.5554	0.40804	0.50399	0.58445	0.6672	0.2442

Figure: table of mean absolute sentiment intensity

The table shows that longer lines of words/syllables leads to stronger sentiment intensity, as predicted. Statistically, I point out that:

- The forced selection of rhyming words at the end of lines will have a more significant impact on the content of shorter lines
- Lines with fewer words/syllables will be more vulnerable to being affected by a single word or an anomalous score.

The main takeaway here is that the model is very good at matching the tone and extremity of rap battle lyrics. A corpus of standard rap lyrics has a mean sentiment intensity of 0.24 (while having an average line length of 8.63 words), while a dataset specifically containing insulting and arrogant lines has a mean sentiment intensity of 0.6672 (with average length of 10.49 words). The table results demonstrate that, particularly at higher numbers of words or syllables, the model is capable of generating lyrics that match the tone of rap battle lyrics. I also note that in all columns the model outperforms a dataset of standard rap lyrics in this metric.

Section 4.3. Evaluating Technical Quality - Rhyme Density

I intended to evaluate the technical quality of the generated lyrics using the *rhyme density* metric contributed by (Malmi *et al.*, 2015). The metric works as follows, in the words of its creators:

1. Compute the phonetic transcription of the lyrics and remove all but vowel phonemes.
2. Scan the lyrics word by word.

3. *For each word, find the longest matching vowel sequence (multisyllabic assonance rhyme) in the proximity of the word.*
4. *Compute the rhyme density by averaging the lengths of the longest matching vowel sequences of all words*

However, I encountered an issue when attempting to implement the method, which I could not solve.

As previously mentioned, the CMU rhyming dictionary contains multiple pronunciations for lots of words in the dictionary. For the purpose of checking if 2 words rhyme, this is not an issue since I can check all combinations of pronunciations. It is beneficial in fact, since it allows more flexibility when selecting a rhyming word.

However, when comparing the pronunciations of more than 2 words, the number of pronunciation combinations increases rapidly, especially since some words can have more than 2 pronunciations. If I define 'proximity' as a line length either side of a word, and say a line is on average 8 words: If I say a generated word has on average 2 pronunciations, the number of pronunciation combinations among the 17 involved words is 2^{17} , or ~131,000.

This number is the number of pronunciations for a sequence of 17 words - I would have to iterate ~131,000 times over lists of phonic strings and perform the computations involved in step 3 of the rhyme density algorithm each time! I would then have to repeat this process for every word in the lyrics.

There is no 'usual/most common pronunciation' in the CMU dictionary, so there is no way to restrict pronunciations to one per word without randomly selecting one. Because of this, there was no way I could find to implement the metric without either implementing a whole other method of pronunciation (which there was not enough time for), randomly selecting a pronunciation for each word (which I felt was too random and unsatisfactory), or having the method take far too long to run. Because of this, I felt I had to abandon implementing the rhyme density metric.

Instead, I decided to evaluate lyrical quality through a user survey, which I will discuss in the next section.

Section 4.4. User Survey - Rationale and Approach

It is difficult to create metrics for evaluating higher-level properties of generated lyrics. A survey involving human evaluation allows us to evaluate the model in a more holistic and less empirical manner while still maintaining a level of rigour in the evaluative approach.

Before conducting the survey, each respondent was given instructions on how to use the model and what the parameters in the `generate_verse` method did, since this was the method users used to generate text. The respondents then were asked to run the model (at least) 10 times, changing any and all parameters including the seed text as they wished.

After finishing using the model the respondents were asked to fill in a google form reflecting on their experience of using the model and evaluating various aspects of its performance. I chose to utilise likert scales as much as possible rather than asking respondents to 'rate

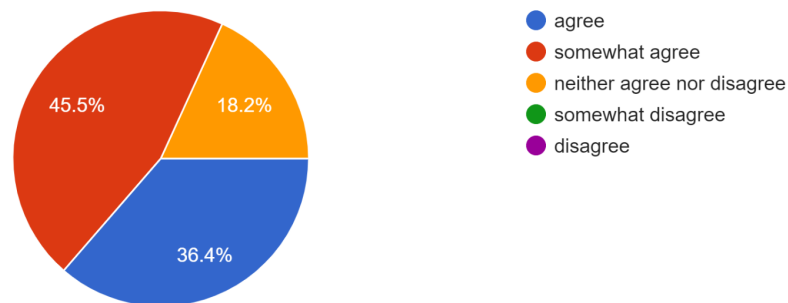
things out of 10', since I felt this would help reduce subjectivity in how respondents interpreted the questions.

Section 4.5. User Survey - Results and Discussion

I will now show the results of the survey and discuss the results for each question. As a disclaimer, the majority of respondents were friends and family. This is an inevitable consequence of having to ask people to perform your survey for free. All respondents were instructed to answer as objectively as they could but a certain amount of bias may be present in the results.

How much do you agree with the following statement: "the model generated text in a reasonable amount of time"?

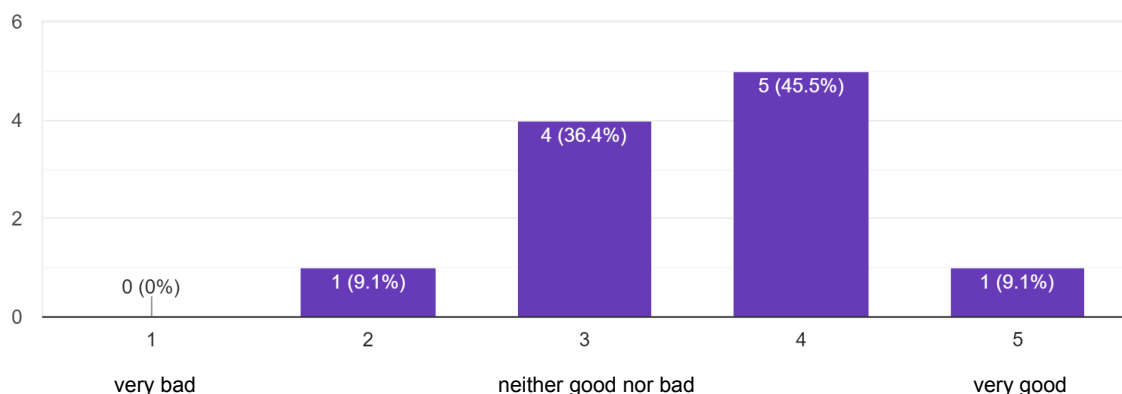
11 responses



82% of users felt that the model generated text in a reasonable amount of time. For reference, generating a typical verse (8 lines, 12 syllables per line) took 27 seconds. This indicates that users do not mind waiting for the model to work. The time taken scales linearly with the number of lines, and also linearly with the number of words/syllables per line. Time complexity was not something I concerned myself with throughout the majority of development, and it would appear users are not overly concerned as long as they do not have to wait for large amounts of time for an output.

How would you rate the model's ability to generate rhyming lyrics?

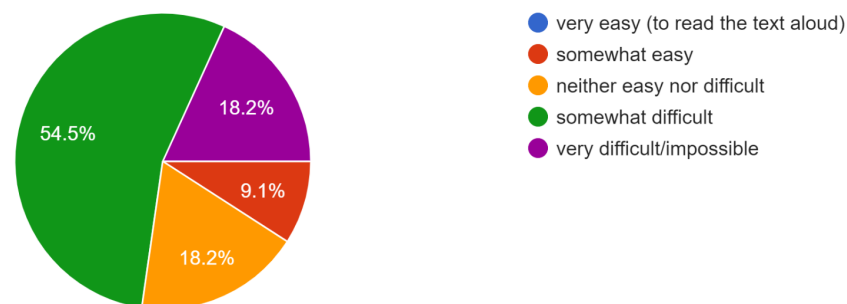
11 responses



The majority of respondents felt that the model was at least somewhat good at producing lyrics which rhymed. However, 5 out of 11 did not. This may be because the model's use of alternate pronunciations was quite liberal. (The model is forced to rhyme, so it realistically has to be my implementation of rhyme which is at fault.) Despite this, I still believe this set of responses could be described as majority positive, with only 1 negative answer out of 11.

Looking back at the text you generated, how would you rate the 'speakability' of the text? (i.e. how easy would it be to read out loud as lyrics)

11 responses



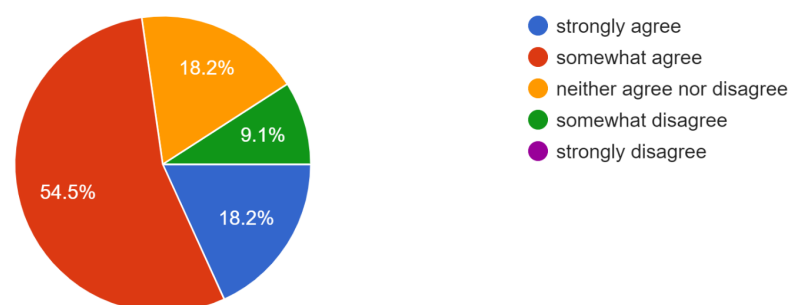
73% of respondents felt that the generated lyrics were difficult to read aloud. There are several reasons this may be the case:

- One respondent suggested the lack of punctuation made it difficult to know when to make pauses
- The censorship makes certain words hard to read out (although the alternative is arguably less preferable)
- The lyrics do not make total grammatical sense, which may throw off humans, who are used to reading text which follows the rules of English.

This set of responses indicates that this is an area with a lot of room for improvement within the model - lyrics are meant to be spoken or sung aloud, so this is quite the fundamental flaw

How much do you agree with the following statement: "the generated text was insulting and arrogant"?

11 responses

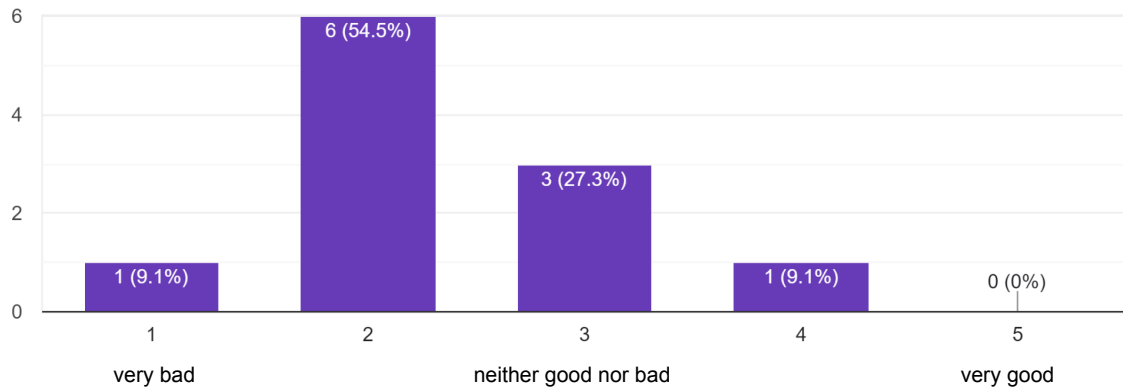


73% of respondents agreed that the generated text was insulting and arrogant. This is

supported by the results of testing using the mean absolute sentiment intensity metric, and shows that the model was successful in replicating the insulting and boastful content of rap battle lyrics.

Overall, how would you rate the quality of the output lyrics, as lyrics

11 responses



The majority of respondents did not think the quality of the output text as lyrics were good. This question is somewhat related to the speakability question, and one of the main reasons respondents felt this way may well be the difficulty in reading the lyrics aloud.

Summary of Results

The survey results demonstrate that the model has strengths and weaknesses. Although the tone of the model and implementation of rhyme were good, the speakability and overall lyrical quality were found lacking.

Section 5. Discussion:

I found that my approach had some benefits but also drawbacks.

The rules-based approach to generating rhyming lines worked well - this is backed by the survey results. In particular, the fact that the probabilities generated by the model are used when selecting a rhyming word means that the rhyming words still generally fit the context of the rest of the line, rather than appearing as if a random word which rhymes has been selected.

The method of creating the insults and boasts dataset from the rap lyrics dataset can also be considered a success in the sense that training the model on it resulted in a text output which was as extreme and insulting as desired, almost to the same extent as the training dataset itself, which consisted of lines from professional rappers.

One thing I overlooked throughout the project was the importance of speakability. One criticism agreed on by the survey respondents and myself is that the generated lyrics rarely made much sense grammatically and that this impacted the speakability and overall quality of the lyrics. Speakability was not a criterion I considered at all early in the project, as I instead chose to focus on features inherent to rap battle lyrics specifically. There was an implicit assumption that the generated text would be easy to speak aloud, and this was not the case.

I feel that a certain amount of grammatical correctness could be introduced with a larger dataset - given that mine had only around 8000 lines, the model may not have had enough text to implicitly learn the structure of a typical sentence and basic grammatical patterns. Models with large datasets (e.g. GPT-3.5) have a much better grasp of higher level features of language such as grammar and sentence structure. However, the fact that the model removes all punctuation and generates text with no punctuation except apostrophes is something that could be addressed. In hindsight, more effort could have been put into expanding the dataset before proceeding onto the design and implementation of the model.

The shortcomings of my implementation of pronunciation were also exposed at quite a late stage in the project - the complexity introduced by having multiple pronunciations for each word was not something I considered until it was too late. I think if I had more time, programming a second implementation of pronunciation with only one pronunciation per word would be useful.

If I had more time, I also would have liked to use a similar rule-based approach to ensure generated text follows a set pattern of metre in each line. This wouldn't require any major innovation, since metre can be reduced to words containing certain stress patterns and their position in a line. The CMU dictionary already includes stress on each vowel phonic (although the multiple pronunciations issue may persist). This problem is left open to further research.

One other thing I would say I have learned is that generating lyrics using AI is hard! Machine learning methods are powerful and their applications are widespread, but I would question whether the pursuit of generating song lyrics is a worthwhile use of these methods (since humans seem to be a lot better at it), especially given that training these models inevitably requires the use of human-written song lyrics anyway.

Having said all that, I still think the project has been worthwhile - I could not find any previous attempts to make a rap-battle lyric generator capable of generating a whole verse, and the project has revealed a lot of potential improvements and topics for further research.

Section 6. Conclusion

Rap Battle Lyric Generation is a niche and relatively unexplored problem within the field of text generation. It presents specific problems (in addition to the general problems surrounding text generation), those being the lack of available training data and the additional tonal and grammatical features such as use of first and second person pronouns, and insulting and boastful lines.

I contributed an approach to solving the dataset issue, as well as an approach for generating a set of rap battle lyrics from a user prompt. The model was capable of generating rhyming verses of a given number of lines, and with a given number of words or syllables per line.

I found that the model was good at replicating the insulting and arrogant tone of rap battle lyrics, and that the rules-based approach to enforcing rhyme was successful. However, my approach also led to shortcomings in overall lyrical quality and speakability.

In summary, the project has provided a solution to the problem of rap battle lyric generations while also revealing some areas for further research in the field, particularly with regards to introducing rhythm and metre, and possibly constructing a larger training dataset.

Section 7. References:

Bengio, Y., Simard, P. and Frasconi, P. (1994) 'Learning long-term dependencies with gradient descent is difficult', *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council*, 5(2), pp. 157–166.

Brown, T.B. *et al.* (2020) 'Language Models are Few-Shot Learners', *arXiv [cs.CL]*. Available at: <http://arxiv.org/abs/2005.14165>.

Carnegie Mellon University (2023) *CMU rhyming Dictionary Website*, *CMU dict website*. Available at: <http://www.speech.cs.cmu.edu/cgi-bin/cmudict> (Accessed: February 2023).

Edwards, P. (2009) *How to Rap*. Chicago Review Press.

Goodfellow, I. (2016) *Deep learning / Ian Goodfellow, Yoshua Bengio and Aaron Courville*. (Adaptive computation and machine learning).

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

- Hutto, C. and Gilbert, E. (2014) 'VADER: A Parsimonious Rule-Based Model for Sentiment Analysis of Social Media Text', *Proceedings of the International AAAI Conference on Web and Social Media*, 8(1), pp. 216–225.
- Kingma, D.P. and Ba, J. (2014) 'Adam: A Method for Stochastic Optimization', *arXiv [cs.LG]*. Available at: <http://arxiv.org/abs/1412.6980>.
- Malmi, E. *et al.* (2015) 'DopeLearning: A Computational Approach to Rap Lyrics Generation'. Available at: <https://doi.org/10.1145/2939672.2939679>.
- Navarro-Cáceres, M. *et al.* (2020) 'Integration of a music generator and a song lyrics generator to create Spanish popular songs', *Journal of ambient intelligence and humanized computing*, 11(11), pp. 4421–4437.
- Paupier, F. (2018) *RapLyrics-Scraper/lyrics_US/*, *GitHub*. Available at: https://github.com/fpaupier/RapLyrics-Scraper/tree/master/lyrics_US (Accessed: January 2023).
- Pudaruth, S., Amourdon, S. and Anseline, J. (2014) 'Automated generation of song lyrics using CFGs', in *2014 Seventh International Conference on Contemporary Computing (IC3)*, pp. 613–616.
- Rumelhart, D.E., Hinton, G.E. and Williams, R.J. (1985) *Learning internal representations by error propagation*. California Univ San Diego La Jolla Inst for Cognitive Science. Available at: <https://apps.dtic.mil/sti/citations/ADA164453>.
- Rumelhart, D.E., Hinton, G.E. and Williams, R.J. (1986) 'Learning representations by back-propagating errors', *Nature*, 323(6088), pp. 533–536.
- Wu, D. and Addanki, K. (2015) *Learning to Rap Battle with Bilingual Recursive Neural Networks*. (International Joint Conference on Artificial Intelligence).

Section 8: Appendices

Appendix 8.1. Lyrics generated by GPT-3.5

Chatbot available at <https://chat.openai.com/chat>. (you need an account, so I cannot link directly to the page displaying the generated text). Note that Chat-GPT uses GPT-3.5, unless you subscribe to chatGPT plus (I do not), in which case you have access to GPT-4.

Prompt: give me a rap battle verse

ChatGPT response:

Verse 1:

I'm the king of this battle, you can call me the rap boss
My rhymes hit harder than a hammer on a nail toss
You better step up your game if you wanna compete
My flow's so sick, it'll leave you incomplete

Verse 2:

You're no match for me, you're just a wanna-be rapper
Your rhymes are weak, they couldn't even make a crapper
I'm spitting fire, while you're just blowing smoke
You'll be begging for mercy after I deliver this stroke

Verse 3:

I'm the champ of the game, the rap god in the flesh
You're just a pretender, trying to put me to the test
I'll dismantle you piece by piece, line by line
And when I'm done, you'll be left with nothing but a rhyme crime

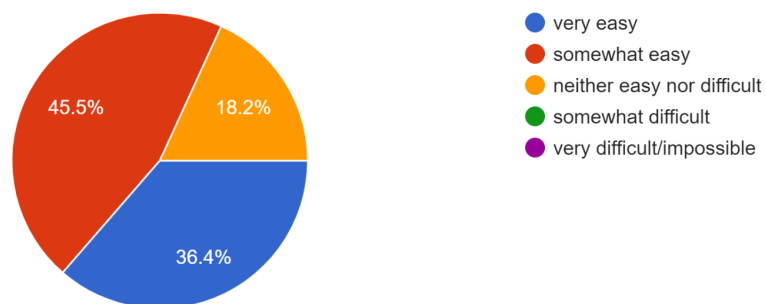
Verse 4:

You're a newbie in the scene, still trying to find your flow
While I'm already on top, and there's nowhere else to go
You can try to come at me with your best shot
But in the end, it's me who's gonna claim the crown, that's a fact, not a thought.

Appendix 8.2: Full Results of User Survey:

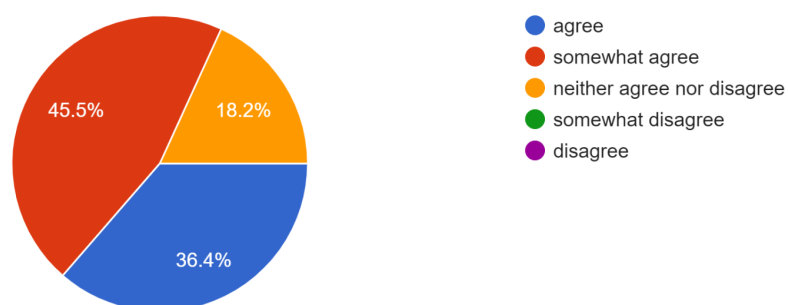
How easy did you find it to use the model?

11 responses



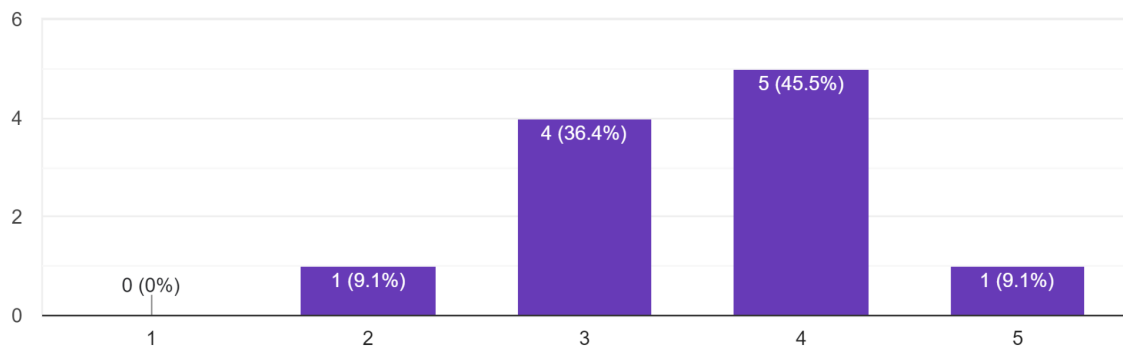
How much do you agree with the following statement: "the model generated text in a reasonable amount of time"?

11 responses



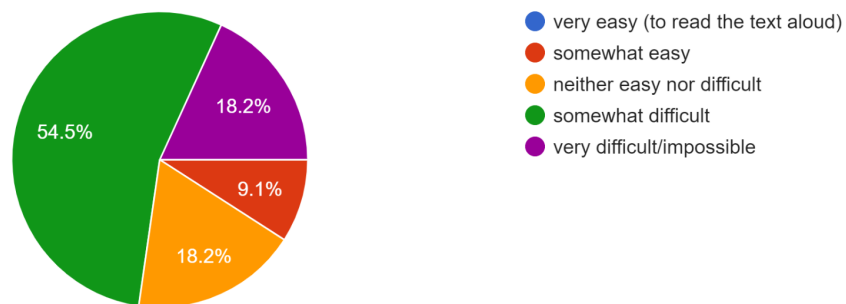
How would you rate the model's ability to generate rhyming lyrics?

11 responses



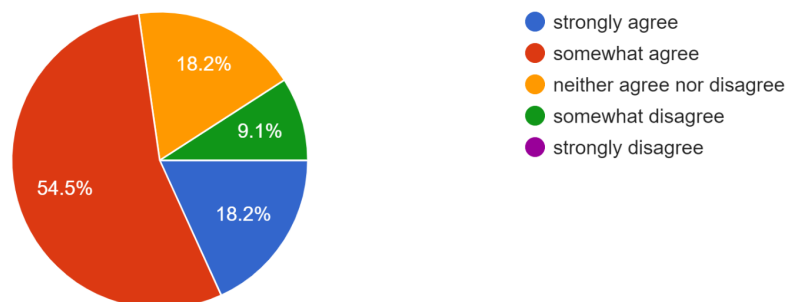
Looking back at the text you generated, how would you rate the 'speakability' of the text? (i.e. how easy would it be to read out loud as lyrics)

11 responses



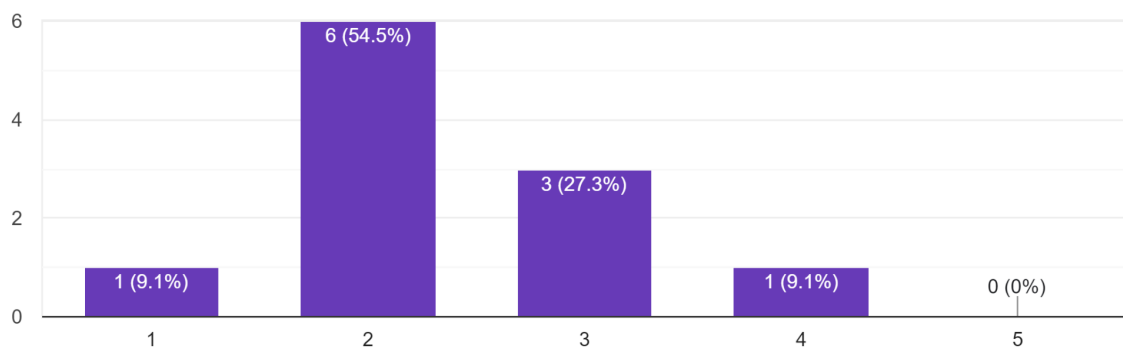
How much do you agree with the following statement: "the generated text was insulting and arrogant"?

11 responses



Overall, how would you rate the quality of the output lyrics, as lyrics

11 responses



If you have any other comments, please add them here. Thank you for participating in the survey! ::

6 responses

The text had the feel of a rap. I would find it easier to read if there was punctuation breaking it into sentences.

This was really fun to test!

no comment

a couple of times a line would be the same 2 words repeating

It made me chuckle a few times

Pedantic I know, but it is difficult to speak a word aloud if it has been censored with asterisks.

Appendix 8.3: sample model outputs

Verse one; prompt: 'you're a loser', 12 syllables per line, 8 lines

Output:

you're but i should call it mad source over of lost
one guys could sing go us in grab it like before
i express like a an used to be my french mouth
and your own shit ain't no shit or been here out
of you are sure you could fight no war or die still
it's through you stupid girl shit word you write get
a song this b*tch with them shits 'em that share that shit
that rhyme about it i have be here to if

Verse two, prompt: 'I've ran out of pepsi', 9 words per line, 8 lines

Output:

i've ran out of pepsi you n***as broke ho
but your n***as is it up mad at don't
you aint out my level of day do i
feel a love anecdote got a minute b*tch like
the last heart they told me they love me
to hit 'em with that crime how we be
like this like raps like me like a uhh
block i don't know shit she got me that

Verse three, prompt: 'who lives in a pineapple under the sea', 12 syllables per line. 12 lines

Output:

she know i know those enough to go heaven go
to get up like a kid who a baby no
big shorty with you posers they're alone they call
that shit she was hurt for me i would be on
the whole francisco days the life or is our hand
will set our little dreams here and feet motherfucking
which it is to the best behind it gets you go
wanna go hurt life she tryna take it know
that i'm good yet i deserve it for it to you
love the game and i can see it g so woo
i don't know it was so good to me yeah i love
him more like a kid how to make you real that