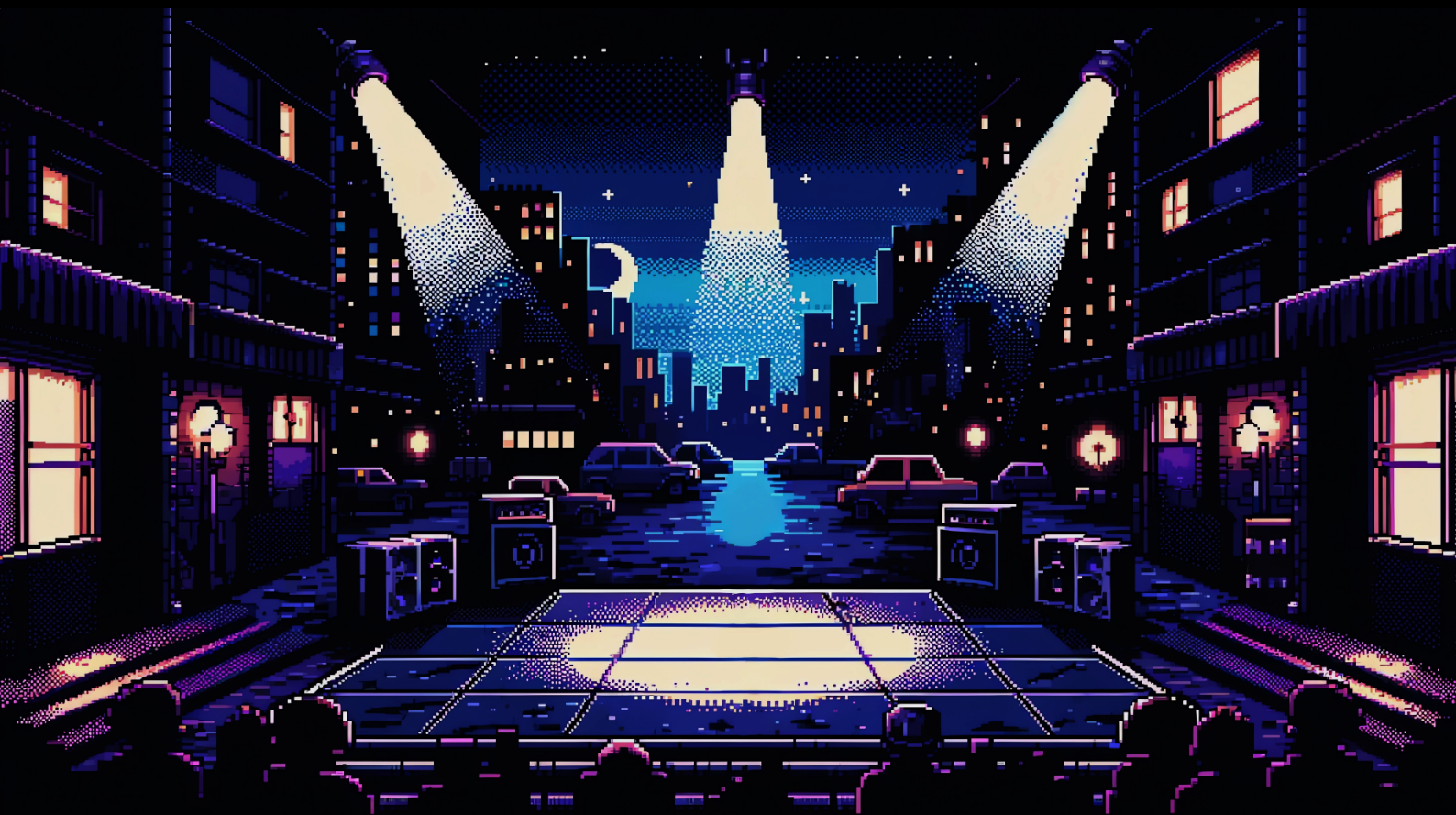
RAP BATTLE WITH NLP

TELECOM Paris

IA717 - Natural Language Processing

Pierre Billaud | Maxime Ledieu | Jaime Montealegre



Summary

Introduction

In our project, we ventured into the realm of natural language processing with a specific goal: to create a model capable of generating rap battle verses. Our approach was hands-on and focused on the practical aspects of NLP. We started by scraping rap battle data from the web, which involved collecting and cleaning large volumes of text to ensure our model had a solid foundation of quality data to learn from.

The heart of our project was training a model using Long Short-Term Memory (LSTM) networks, a type of recurrent neural network well-suited for understanding and generating sequences of text. Initially, we developed a basic LSTM model to get a baseline for performance.

As we progressed, our focus shifted to refining the model and optimizing it. We experimented with various hyperparameters, tweaking aspects like the number of layers, learning rate, and batch size. This process of playing with hyperparameters was crucial in improving the model's ability to generate more coherent and stylistically appropriate rap verses.

Our journey was as much about learning the intricacies of LSTM models and hyperparameter optimization as it was about generating rap battles. In essence, we merged the art of rap with the science of NLP to create something consistent.

Project choice

In this project, we chose to develop an NLP model focused on the generation of rap lyrics, thus departing from the approach initially proposed, which was the generation of poems. The idea grew out of our mutual love of rap linguistics and stylistics, and in particular of the rap battle format, which presents interesting challenges in terms of rhythm, rhyme and interaction between two speakers.

Using data from rappad.com, we designed and trained a model that generates texts that are intended to respond to each other, like a rap battle where a Challenger and a Defender face each other and battle it out in sentences. This report describes our work process, from data collection to text generation, focusing on the specific adaptations needed to successfully recreate - via a model - how is working a rap battle.

Group functioning

We first agreed on the project we wanted to make. After various discussions, we concluded that we wanted to make a model generating rap battles in English. We didn't have a precise format in mind; the initial idea was to make a model that could give 3, 4 or 5 phrases in a row as a challenger, then 3, 4 or 5 phrases in return for the defender. These phases had to be repeatable.

We divided up the tasks as follows :

Table 1 Work's repartition

People	**Tasks**
Jaime	• Data Gathering • Basic Model training
Maxime	• Data cleaning • Model optimization
Pierre	• Web Scrapping • Data cleaning • Optional: Unity Demo

We also used the following tools to work together:

- *Google Drive* to share files
- *Google Collab Pro* to create a notebook and training models with GPU A100
- *GitHub* to handle file versioning and to store our notebooks somewhere

I Data pre-processing

1 Data source

We thought about the different possible sources for data on rap battles, as this kind of performance is often done on video rather than in written format, which makes it complicated to find usable data for training purposes. We considered all the following solutions:

1. Read automatic subtitles for rap battles videos (on YouTube and similar websites)
2. Use comments under rap battles videos with all the lyrics and punchlines.
3. Use lyrics directly from rap songs
    a. A battle uses the same vocabulary as rap music, but is rap music usable data for what we want to do?
4. Using written rap battles

After many unsuccessful attempts we found the RapPad.co site, which, although it doesn't have texts structured in a well-defined number of paragraphs, does have a very large text base. RapPad.co (https://www.rappad.co/battles) is an online platform dedicated to writing and sharing rap lyrics. It offers users the tools to write lyrics, take part in online rap battles, and receive feedback from the community. What's special about this tool is that all users (not just professionals) can compete in a wide variety of formats. Users write their text, then record their voice directly afterwards. A vote is then taken to see who has won the duel.

2 Rap battle format on *RapPad*

The format of a rap battle can vary, as indicated above. Generally speaking, the format of a rap battle on the RapPad.co website is as follows:

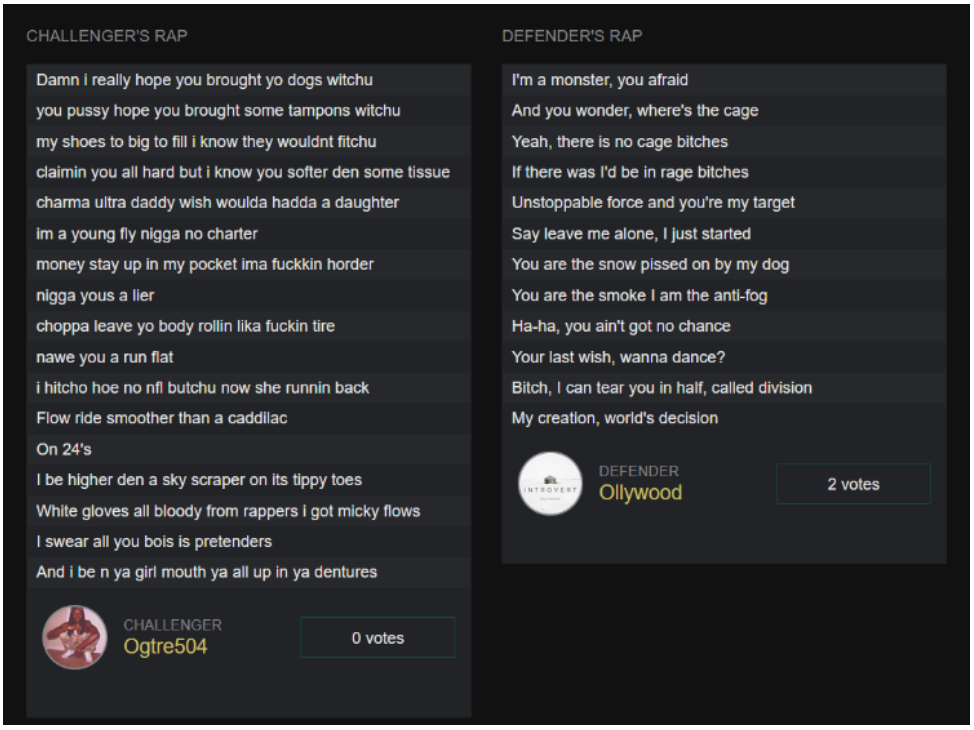


Figure 1 Rap battle versus

It features two rappers successively answering each other in two short rap texts that don't necessarily have the same number of lines, nor the same number of total syllables, but they do rhyme and have a certain rhythm. Other battles are longer or shorter and include several phases of this type. This extract can be considered usable. We have collected around 19,000 battles in this format.

3 Web Scraping

Once we had correctly identified the data source we were going to use, we used web scraping to harvest all 19,000 rap battles. As a reminder, web scraping consists in extracting data from the web, and for this purpose there are tailor-made libraries in Python.

Without going into the details of the code (which is provided in the notebook), here's how we went about it:

- Identify a library capable of scrapping the web
    - We chose BeautifulSoup for several reasons:
        - Already used by a group member
        - One of the most famous
        - Useful documentation
- Targeting URLs for scrapping
    - url = f'https://www.rappad.co/battles/{battle_id}'
- Define a scrapping function
    - Iterating over urls to scrape
    - Define storage file(s)
    - Define how to explore web pages and extract data from HTML code

Once sorted, here is an example of the text file generated by our web scraping function:

URL: https://www.rappad.co/battles/46160
Winner: 80HD

CHALLENGER NAME:

QwertyContent

CHALLENGER LYRICS:

So little man with light bulb head can freestyle

Well, your head will shatter meanwhile

Using poetry books for your rhymes

When I see your raps, they got no chime

You are almost like an AI, botting your votes

You are a lag, sorry to give up your hopes

You are an MPC, repeating the same lines

Broken like your PC, lagging at the same time

Your bar resolution is 1000 pixels short

Last I checked, all your raps were in the morgue

DEFENDER NAME:

80HD

DEFENDER LYRICS:

I used to be the first one out

Fuck that shit i'm in ya face wit a nerf gun pow

Your up now

Time to search ya browse

Your in search of clout?

Go ahead plot your y and x's

Yet my index is quite expensive

And the side effect is dyin senseles

My intent is to have fun with high suspenses

As I suspected

---

4 Data Gathering

After the web scraping, and for an easier handling of data, all data retrieved is stored in one source. To be able to identify possible useful and related variables such as name of the challenger, name of the loser, text of the challenger, text of the loser, etc; a data base was used to store all rap battles. From this SQLite database, the final corpus was generated. This database is only 48.7 Mb but it is easy to create, insert and extract data.

Rap battle data are stored in a single table in the following way:

Id (int)	Url (text)	Winner (text)	Challenger_name (text)	Challenger_lyrics (text)	Defender_name (text)	Defender_lyrics (text)

5 Data cleaning and pre-processing

The data cleaning and pre-processing phase was both challenging and intricate, requiring careful consideration of linguistic characteristics, stylistic elements, and technical constraints. Our approach, tailored to the unique needs of rap battle generation, ensured that the final datasets were prepared for the different models, balancing cleanliness with authenticity.

Main Challenges and Difficulties

In our project focusing on the analysis of rap battles, we encountered several unique challenges that required careful consideration and innovative solutions:

1. Maintaining Authentic Rap Style: Rap battles have a distinct style characterized by slang, African American Vernacular English, and non-standard English. Preserving this while cleaning the data was a primary challenge.

2. Inconsistency in Content: The user-generated content from RapPad included varied spellings, typos, and grammatical inconsistencies, making standardization difficult.

3. Balancing Cleaning with Style Preservation: The key was to remove errors and irrelevant data without stripping away the stylistic and linguistic elements crucial to rap battles.

4. Large Volume of Data: Processing and cleaning approximately 19,000 rap battles required a systematic and efficient approach.

5. Identifying and Handling Personal References: Lyrics often contained personal references to self and opponents, which needed to be generalized for the model to learn effectively.

Cleaning Functions - Utilization and Challenges

In our project, we recognized the need for specialized techniques to process and analyze rap battle data effectively. These techniques were essential to clean the data and make it lighter. In this overview, we outline the key methods we employed to ensure both the integrity and relevance of our data, while also preserving the authentic style of battles.

Table 2 Overview of all the cleaning techniques

Techniques	Usage	Challenges
Standardizing Rapper Names	To correct varied spellings or typos in rapper names for consistent entity tracking.	Differentiating between creative representations and actual typos.
Handling Slang and Non-Standard Language	Usage of Specialized Lexicon: To preserve the distinctive linguistic features while cleaning.	
Usage of Specialized Lexicon	Employing regex to remove irrelevant characters like URLs and emojis.	Retaining stylistically important punctuation.
Spelling Correction	Correcting obvious typos while keeping intentional stylistic spellings.	Differentiating between intentional and unintentional spelling variations.
Custom Dictionary for Rap Vocabulary	Merging standard dictionaries with custom rap-specific ones.	Ensuring all relevant rap battle vocabulary was included.
Expanding Abbreviations and Handling Colloquial Language	Expanding common abbreviations and retaining colloquial terms.	Maintaining the conversational tone of rap battles.
Removal of Non-Standard Words	Removing words not found in the combined standard and custom dictionaries.	Filtering out irrelevant content without losing meaningful rap content.
Replacing Self and Opponent References	Replacing personal references with generic placeholders.	Maintaining context without personalizing the data.
Decision Against Certain Techniques	- Lemmatization: Not used to preserve the conjugations and stylistic variations. - Tokenization: Deferred to the modelling stage to capture linguistic nuances. - Corpus Structure Depending on the Model	

Cleaned Data Structure

In our endeavor to analyze rap battle data, structuring the corpus was a critical step. We adopted two distinct approaches to cater to different modeling needs:

1. Basic Model: General Corpus for Text Generation Model
    - Structure: Concatenated cleaned lines without distinguishing between the challenger and defender or the specific battles.
    - Rationale: This structure was suitable for the model to learn general patterns in rap battles.

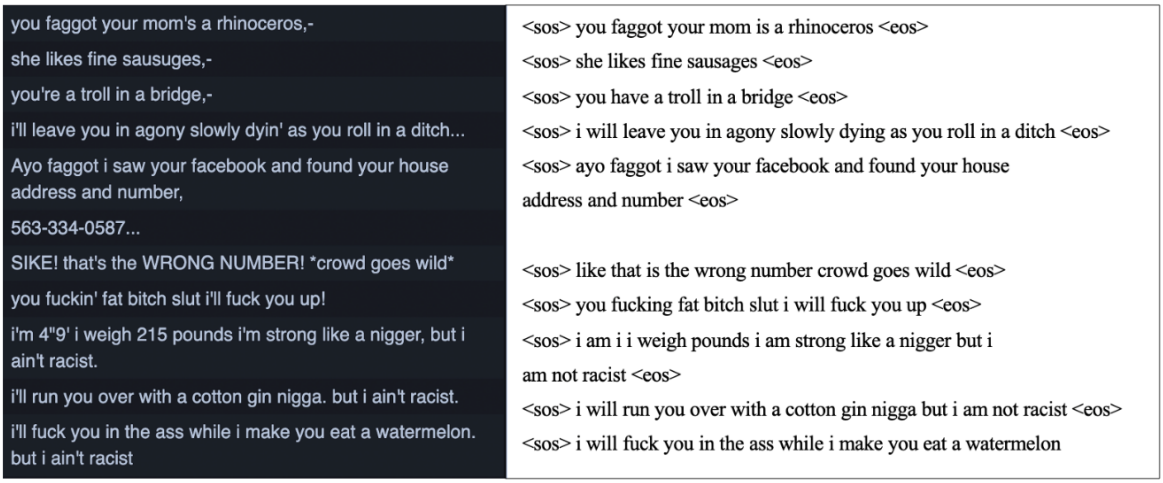


Figure 2 Before and after cleaning a text

2. Optimized Model: Optimized Corpus
    - Structure: Included markers indicating the start and end of each battle, the lyrics of the challenger and defender, and the battle winner.
    - Rationale: This format allowed the model to generate contextually relevant responses by understanding the flow and outcome of each battle.
    - Use of special Tokens: Incorporated to demarcate the beginning and end of each battle, and verse, aiding the model in generating coherent text structures.

<battle_start>
<challenger>
<sos> recovery is a pain <eos>
<sos> and a bitch but prison <eos>
<sos> so just listen <eos>
<defender>
<sos> you know that heard of that <eos>
<sos> tall ya girl to chest easy <eos>
<winner> challenger
<battle_end>

II Basic Model & Analysis

This section has been made for the deep comprehension of the basic LSTM model: text generation.

1 Training Corpus Analysis

To compare the train corpus with the future results of the model, we count the frequency of words and sequences of consecutive words. The output of the model might contain the most frequent sequence of words.

The next figure displays the count of the most frequent single words in the train corpus. Since the train corpus consists in rap battles, usually the participants talk about themselves in a first-person way. Therefore, the word “i” is the most frequent word. In the same way, the competitors talk about their opponents usually so the word “you” is the second most frequent word. For the rest, in general articles are very frequent (“the”, “a”, “to”, etc) and words which relate with “I” and “you” are also frequent such as “my” and “your”.

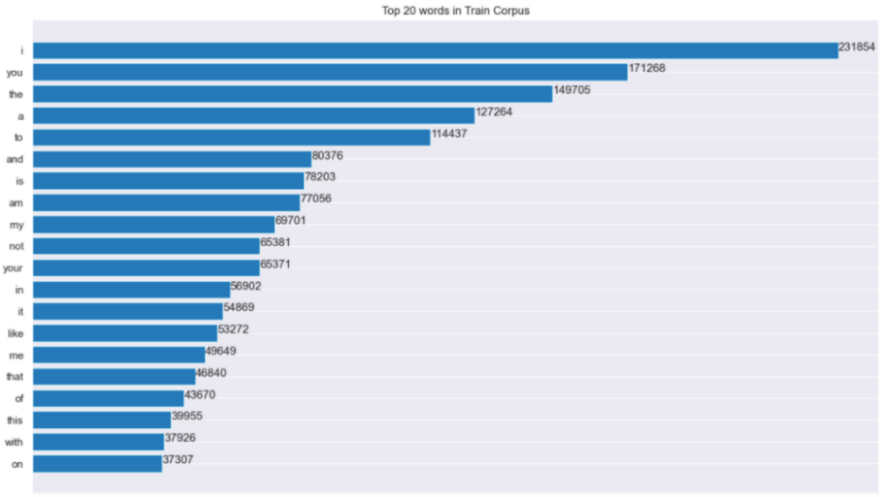


Figure 3 Most frequent single words

The following figure shows the most frequent 2 consecutive word combinations in the train text. In a similar way as in the previous case, the combination of “i” and another word is very frequent.

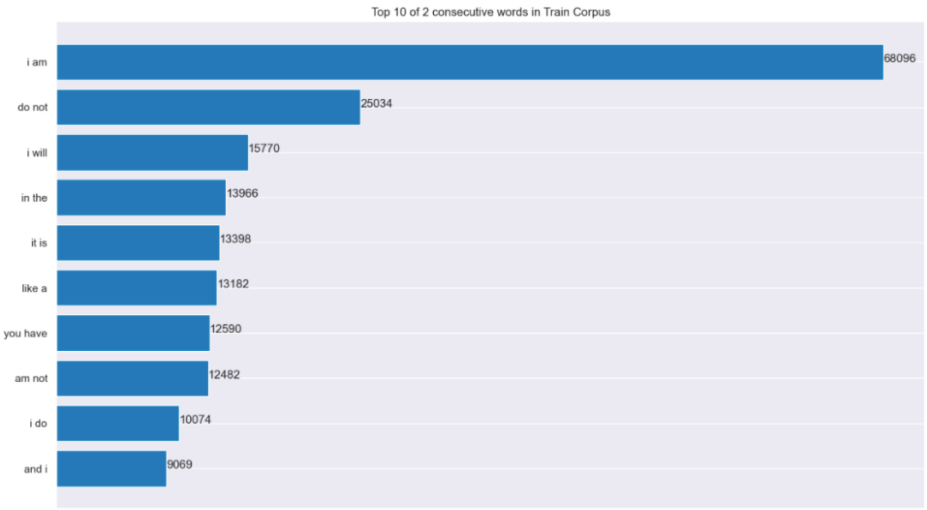


Figure 4 Most frequent 2 consecutive word combinations

Below, it's possible to see the most frequent 3 consecutive word combinations in the train corpus. The combination with "i" and other words are the most frequent cases. These combinations consist mainly in the two words "i am" and a third word ("i am not", "i am a", "i am the", etc). In the context of the rap battles, this means that the participants are usually more describing themselves rather than talking about their opponents (the combination "you do not" only appears at the 5th position and is the only one with a third person pronoun).

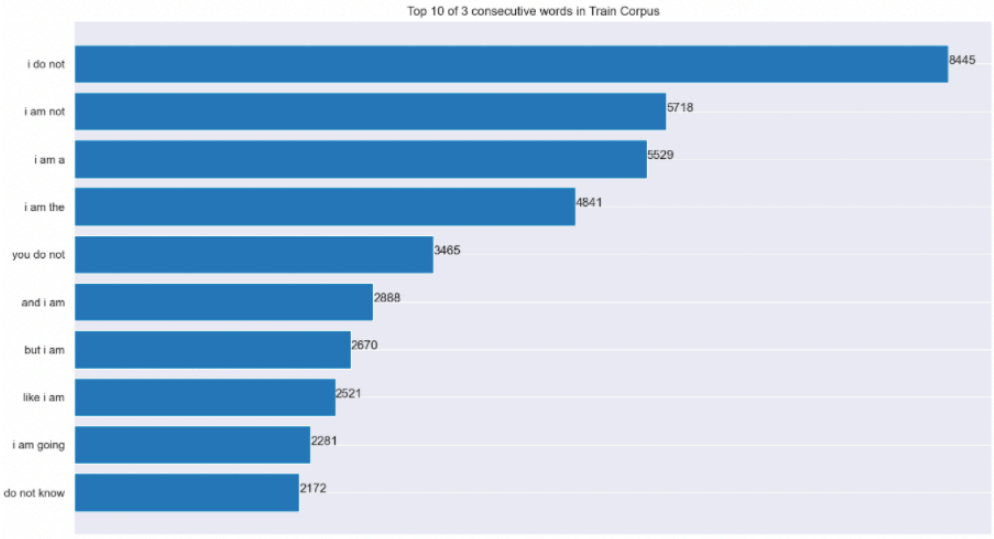


Figure 5 Most frequent 3 consecutive word combinations

The previous figures show the most frequent items in the train corpus. Since the model is trained with this corpus, the output of the model and the generate() function might have these combinations so it's expected that we see the above cases in the output.

2 Basic Model

A Long short-term memory (LSTM) network is used to train and to generate the rap text. These LSTM are recurrent neural networks (RNN) adapted to analyze sequential data such as time series or text. They deal with the vanishing gradient problem present in traditional RNNs, that is, they handle the issue where the gradient turns very small preventing the weight from changing the value in each iteration. The LSTMs have a short-term memory that can last thousands of timesteps.

To train a LSTM, data needs to suffer certain transformations before the input of the model. First, a dictionary with the tokens of the corpus is created. This dictionary helps as a reference to create the input vector of the model and it consists in a combination of token and index. For example:

"I" = 3

"you" = 45

"playing" = 423

“apple” = 922

This is a sequence of words (with no punctuation) and usually the index is created in function of the first time of appearance of the word. We decided to remove the punctuation because in the context of rap, punctuation is not important. Indeed, in the source data there are not too many punctuation characters; in general rap phrases consist in affirmations (no questions). Actually, the most important punctuation character is the apostrophe ‘ which, in English, helps to divide contractions such as “I’m”, “don’t”, etc. However, there were many cases in the source data which didn’t have this consideration i.e. “Im”, “dont”, “wont” which don’t seem incorrect and keep their grammatical meaning. So to ease the analysis, it’s better to remove the punctuation.

The split between train, test and valid sets is done in order and not randomly. The reason behind this is that we want the model to learn the relation between lines. In the source data, consecutive lines are ideas and even responses of the other opponent.

The next step would be to tokenize the train corpus. This consists on using the dictionary created previously to generate a vector with the index of the words. In this way, the tokenization is a translation between words to numbers. For example, if the dictionary has the following entries:

“down”=1, “the”=2, “block”=3, “sit”=4, “back”=5

The sequence “down the block” would be tokenized to “1, 2, 3” where the space is not considered in the tokenization process.

There are two special items which are added to the dictionary:

<eos> : which means “end of line”

<sos>: which means “start of line”

During tokenization, the character “\n” is tokenized to <eos>. So, for instance, the sequence:

down the block

sit back

Would be tokenized to:

1 2 3 <eos> 4 5

We don’t really use the token <sos> since after the token <eos> the next character would be automatically in the next line, so we don’t need another token to mark the start of a new sentence. This also reduce the length of the tokenized vector and, potentially, reduce time processing.

There is one step remaining before entering the data to the model. The tokenized vector created is a flat entity which is not very adapted for gpu processes. This vector needs then to have another shape so it can be treated more efficiently.

So, a vector in the form of:

[a b c d e f g h i j k l m n o p q r s t u v w x y z]

is transformed to:

$$\begin{bmatrix} a & g & m & s \\ b & h & n & t \\ c & i & o & u \\ d & j & p & v \\ e & k & q & w \\ f & l & r & x \end{bmatrix}$$

And this is the input vector to be inserted into the LSTM model.

3 Training

This section explains the results of the LSTM training. During this step, the cross-entropy loss is aimed to be decreased as much as possible. This loss value is calculated using the predicted probabilities and the true distribution of the classes, that is, the following words for a given sequence of words. The following graph displays the evolution of the loss value through each batch for several epochs.

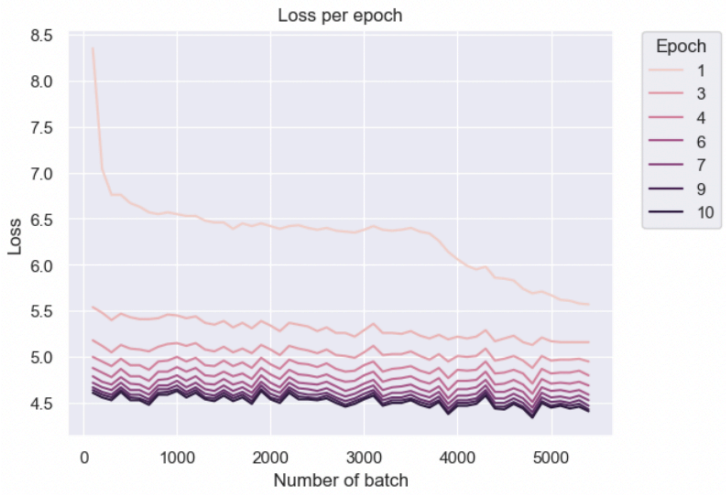


Figure 6 Evolution of the loss value through each batch for several epochs

At the beginning of training, the model has pre-initialized weights, and the initial loss is relatively high as the model's predictions are unlikely to match the true labels. As the model processes more batches and completes epochs, it starts adjusting its weights based on the training data. The loss typically decreases, indicating that the model is improving its ability to make predictions on the training set. Through epochs, it becomes more accurate on the training data and the rate of improvement slows down.

In some batches such as around 3000, for several epochs there is a big fluctuation of the loss value. This can be explained through the variations of the training data and the stochastic nature of optimization algorithms. At the end of each epoch, the loss value tends to stabilize, and no further training needs to be performed to significantly reduce the loss value.

Like the behaviour of loss, perplexity decreases through batches and epochs. The perplexity value explains how well a probability distribution, or a predictive model predicts a sample. At the epoch 1 and batch 1, the initial perplexity id relatively high indicating the uncertainty of the model. As the

model processes more batches and completes epochs, it starts learning patterns and dependencies in the language.

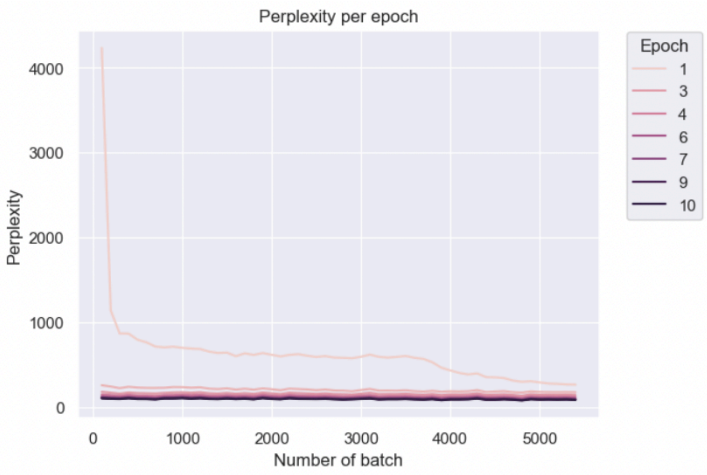


Figure 7 Evolution of the perplexity value through each batch for several epochs

The next image shows the evolution of loss and perplexity through each epoch. In this case, the best (minimal) value of loss and perplexity for each epoch has been considered. Perplexity tends to slightly converge quicklier than loss.

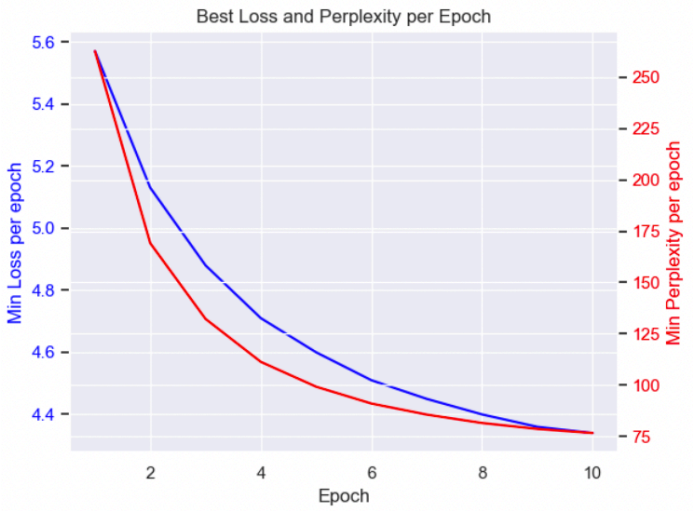


Figure 8 Evolution of the best loss and perplexity values by epochs

The following table shows some models (not all models tested are being displayed) and hyperparameters tested. The loss and perplexity values are calculated with the validation set.

Table 3 Parameters of several trained models

Model	Valid Loss	Valid Perplexity	LR	OPTIMIZER	WDECAY	CLIP	Time (min)
1	10.59	39588.24	0.0005	adam	0	0.25	32.4

2	5.84	345.39	0.0001	adam	0.0001	0.25	36.85
3	4.79	120.39	10	sgd	0	0.25	31
4	4.87	130.22	5	sgd	0	0.25	30.94
5	4.78	118.88	15	sgd	0	0.25	30.81
6	4.85	128.28	20	sgd	0	0.25	47.5

(Model 6 has 5 layers)

In average, all trainings take 30 minutes to complete in a A100GPU powered Collab server. The model 6 takes longer since it was tested with more layers. The model 2 also takes longer since the parameter LR (learning rate) was very small, so the weights of the network are updated by only a small amount during each iteration of the optimization algorithm.

The following image shows the loss per epoch for different models. The model 2, which has a small LR, converges quickly but with no optimal solution. Model 1 has the best initial loss among all models and, in each epoch, approaches to the optimal solution.

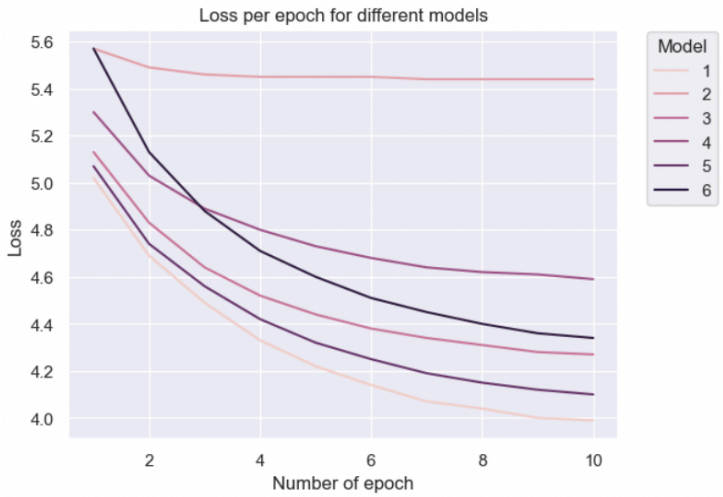


Figure 9 Evolution of loss per epochs for several trained models

In a similar way, the perplexity in different epochs is presented in the next image. Model 2, again, seems to converge quickly but with no optimal solution.

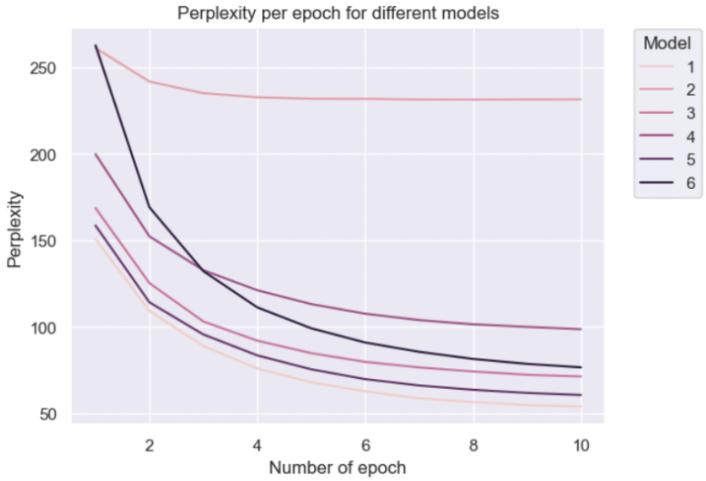


Figure 10 Evolution of perplexity per epochs for several trained models

4 Generated Corpus Analysis

To analyse de performance of the model, a corpus of 200.000 lines has been generated. This corpus uses the model 5 presented in the previous section, considered the best one. A similar analysis to the train corpus has been performed to compare both texts. The most frequent word is "i" which reflects how well the model adopted the nature of the train text where, usually, the participants talk about themselves, so they use a lot the word "i". The word "you" is the second most frequent word, reflecting how the model normally tries to talk about the other participant (as seen in the train corpus.)

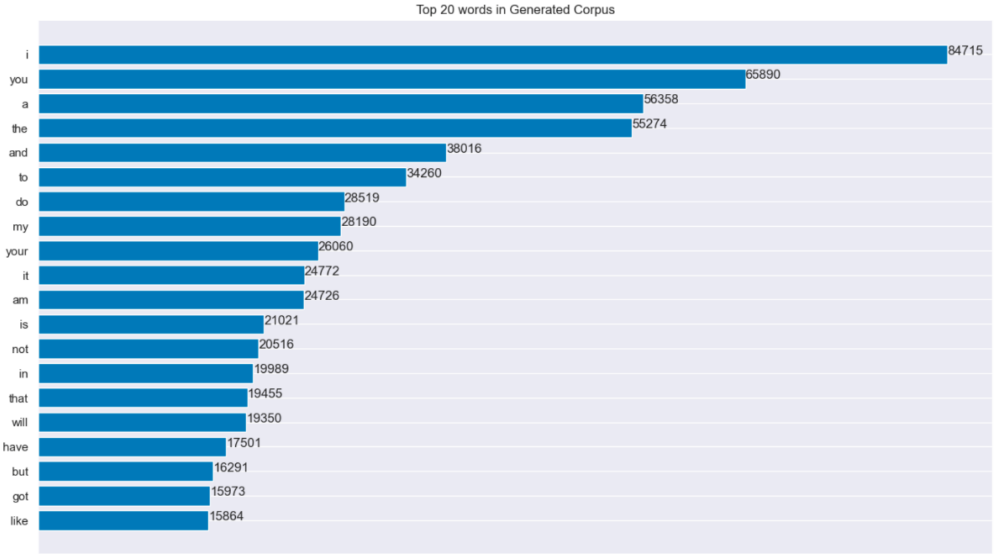


Figure 11 Most frequent words in generated corpus

Similarly, the most frequent 2 consecutive words are combinations of "i" and another verb/ auxiliar such as "i am, "i will" and "i do". This makes sense: the model learns, in the context of the rap battles, the nature of how contestants usually describe themselves ("am").

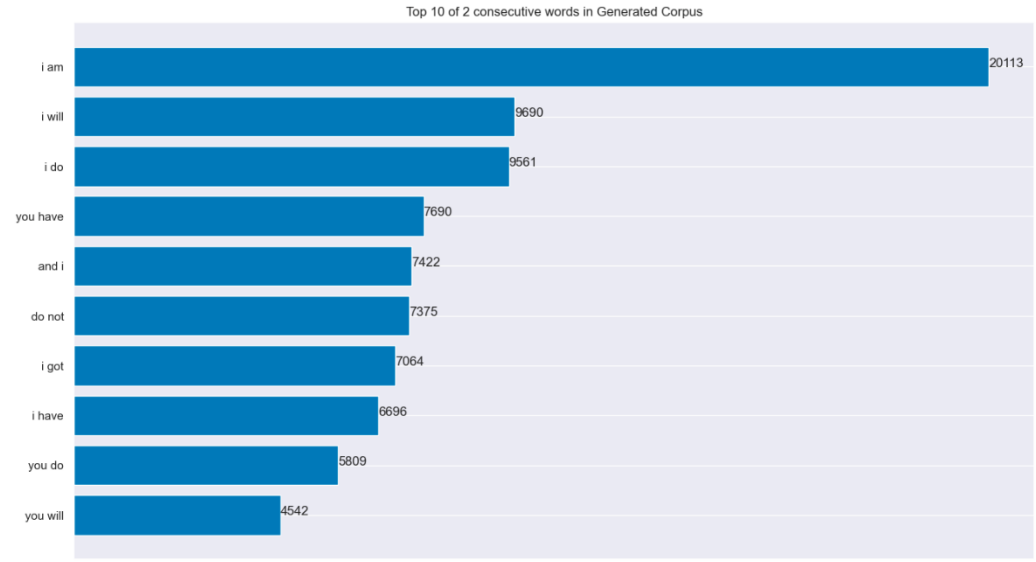


Figure 12 Most frequent 2 consecutive words in generated corpus

Finally, the most 3 consecutive words are also combinations of “i” and, generally, “am” and a third word. As in the previous graph, this behaviour represents the nature of description of the rap battles that are usually written in a first-person singular way.

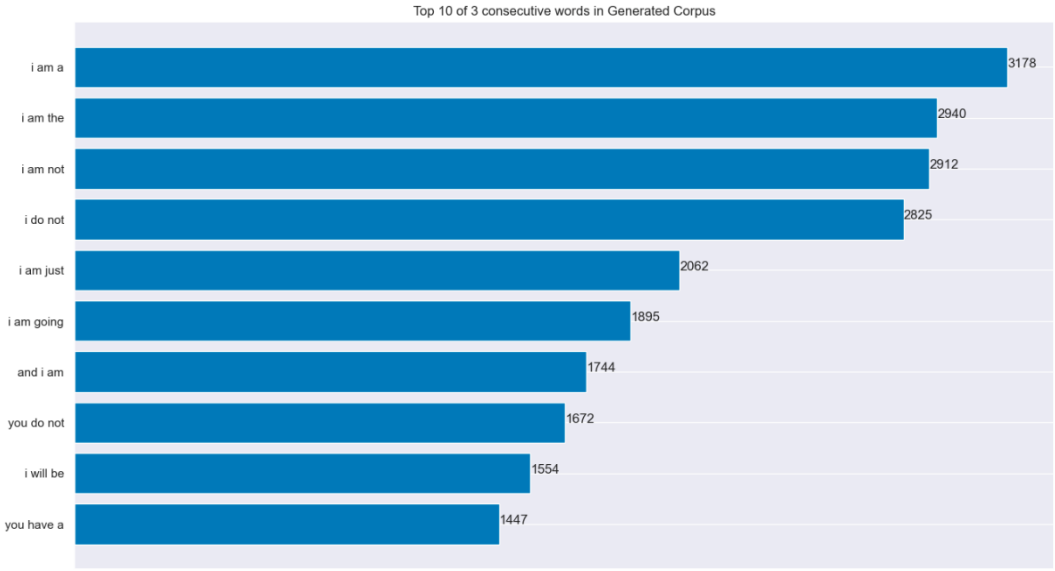


Figure 13 Most frequent 3 consecutive words in generated corpus

5 Zipf’s law Analysis

Zipf's law is an empirical law that describes the distribution of frequencies of elements in a dataset. It is named after linguist George Zipf, who first proposed it to explain the distribution of word frequencies in natural languages.

Zipf's law states that the frequency of any given element is inversely proportional to its rank. In other words, the most frequent element occurs approximately twice as often as the second most frequent element, three times as often as the third most frequent element, and so on. Mathematically, Zipf's law can be expressed as:

$$f(r) = \frac{1}{r^s}$$

where:

- *f(r)* is the frequency of the element at rank *r*,
- *s* is a parameter that characterizes the distribution, and
- *r* is the rank of the element.

This empirical law can describe the behavior of the train and generated corpus. The next table displays the frequencies of the most frequent words and the ratio between each position for the train corpus:

Table 4 Frequencies of words with ratio in trained corpus

Rank	Word	Frequency	Empirical Ratio from 1st position	Theoretical s	Theoretical value	Error (%)
1	i	231854	1	-	-	-
2	you	171268	1.353749679	0.4369609956	-	-
3	the	149705	1.548739187	0.3981761143	1.616156371	4.171451773
4	a	127264	1.821834926	0.4326961221	1.736704413	4.901842354
5	to	114437	2.026040529	0.4387142893	2.006511302	0.9732926284
6	and	80376	2.884617299	0.5912580663	2.19475513	31.43230692
7	is	78203	2.964771172	0.5585046431	3.159883222	6.17466014
8	am	77056	3.008902616	0.5297458048	3.194331255	5.804928286
9	my	69701	3.326408516	0.5470051677	3.20262409	3.865093829
10	not	65381	3.546198437	0.549763034	3.523750639	0.6370427362

Where:

$$\text{Empirical Ration from 1st position} = \frac{Frequency\ 1st\ position}{Frequency\ at\ i\ position}$$

$$\text{Theoretical s} = \frac{\log\left(Empirical\ Ratio\ i\right)}{\log\left(rank\ i\right)}$$

$$\text{Theoretical value} = \frac{1}{\frac{1}{rank\ i^{s\ at\ i-1}}}$$

$$\text{Error} = \frac{\text{abs}(\text{theoretical value} - \text{empirical ratio})}{theoretical\ value} \times 100$$

The following graph displays the evolution of the error through the ranks :

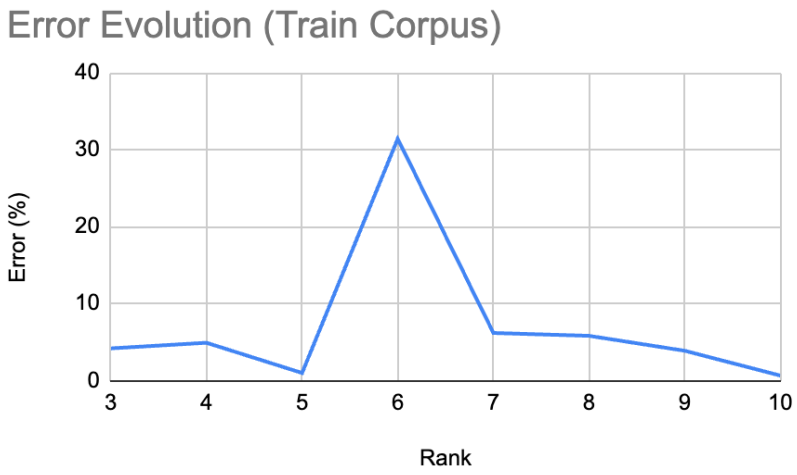


Figure 14 Zipf's low error evolution for train corpus.

Despite the pick at rank 6th, the behavior of the train text seems to be closed to the Zipf's low, this is, the frequency of the most frequent words have a natural (yet empirical) relationship. This represents a "natural" phenomenon in data.

However, the generated campus shows a different behavior. Similarly, to the previous case, the following table displays the frequency and relationships of the most frequent words:

Table 5 Frequencies of words relationships in trained corpus

Rank	Word	Frequency	Empirical Ratio from 1st position	Theoretical s	Theoretical value	Error (%)
1	i	84715	2.736870684	-	-	-
2	you	65890	3.518804067	1.815085185	-	-
3	a	56358	4.113950105	1.287427492	7.345404569	43.99287247

4	the	55274	4.194630387	1.034271846	5.958110925	29.59798097
5	and	38016	6.098853114	1.123436155	5.283539964	15.43119114
6	to	34260	6.767483946	1.067179721	7.485197234	9.588435224
7	do	28519	8.129808198	1.076893161	7.977585923	1.908124542
8	my	29190	7.942925659	0.9965568336	9.387099705	15.38466716
9	your	26060	8.896930161	0.9947578022	8.932168222	0.3945073562
10	it	24772	9.359518812	0.9712535215	9.880019511	5.268215296

Observing the Error value, it variates a lot and doesn't seem to be as stable as per in the train corpus:

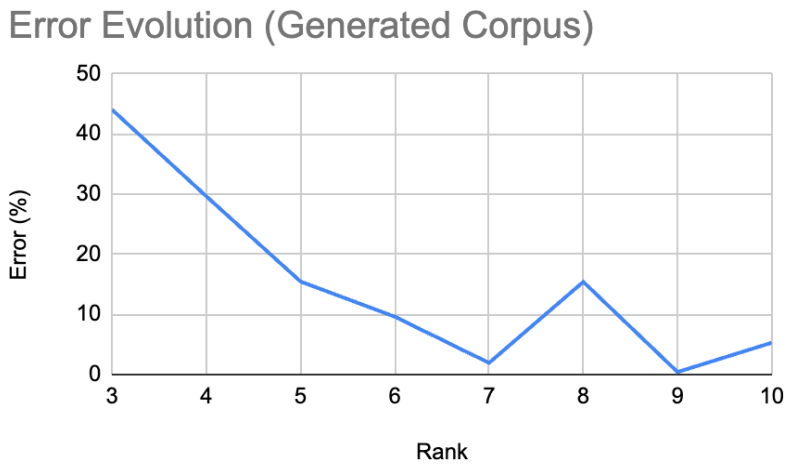


Figure 15 Zipf's low error evolution for generated corpus.

This behaviour doesn't follow the Zipf's law, showing an unnatural relationship between frequencies in words (yet again empirical). Thus, through this analysis, it's possible to see the "unnatural" side of the output model and compare it with the "real world", this is, the train data.

6 Parameters of generate() function

There are two important parameters of the generate() function (which is the one that uses the trained model to produce data): temperature and topk.

Temperature:

- Definition: The temperature parameter controls the randomness of the generated text. Higher values (e.g., 1.0) make the generated text more diverse and creative but might introduce more randomness. Lower values (e.g., 0.5) make the sampling more deterministic and focused.
- Effect: Higher temperature values lead to more diverse outputs, but they might also result in less focused or more random text. Lower temperature values make the generated text more deterministic and focused, but they can also make it more repetitive.

Topk:

- Definition: Top-k sampling involves sampling from the top-k most likely next words according to the model's predicted probabilities. This can help avoid sampling from less likely words and produce more focused and relevant outputs.
- Effect: Larger values of k (e.g., 5, 10, 20) may result in more diverse outputs but could introduce more randomness. Smaller values of k produce more deterministic outputs with a higher focus on the most probable words.

The next image displays several temperature values and their outputs (model 5 used):

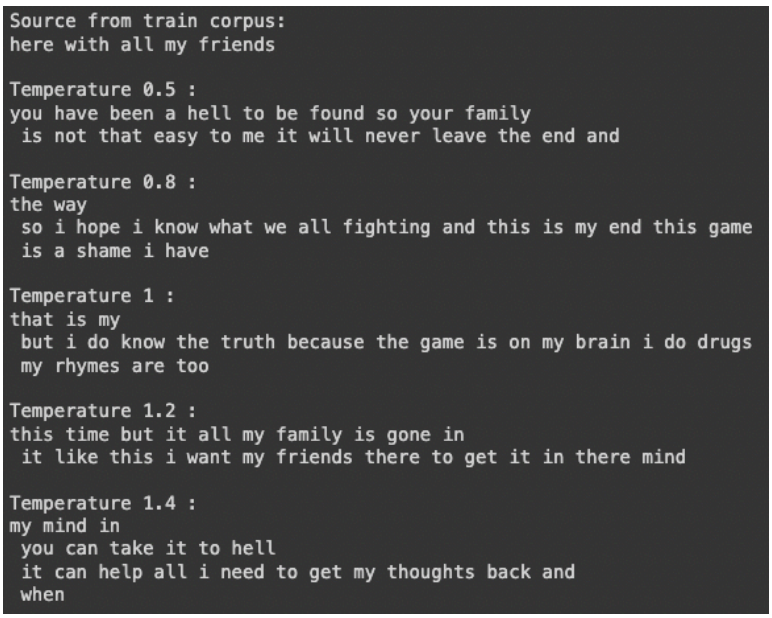


Figure 16 Output for several temperature values.

It's possible to see that increasing the temperature value leads to more randomness but not necessarily with more sense. Still, the word "i" appears often in the outputs, showing, again, the capacity of the model to learn the first-person characteristic of the rap battles.

The next image displays several topk values and their outputs (model 5 used):

```
Source from train corpus:
im not a rapper i am an artist a sorts been dabbling in music since before you were born

Topk 5 :

 i am the type of guy that will never be the same
 but i will just say you will get me through this

Topk 10 :
i had your girl on me like the other hand i am the king
 so you want me
 you know me want to

Topk 15 :
but my dick got no bars and your ass could take more and a hit
 i know when a white man came from that

Topk 20 :
you a little bitch i bet no cursing like some white dog like i would do that like some do not have bars that make

Topk 30 :
this had it your turn it off and get turned off your chest a lot more times in my body so there would i stay
```

Figure 17 Output for several topk values.

When increasing the topk value, the meaning of the phase gets lost. Low topk values seem to be better for the model since they display more congruent and grammatical correct outputs.

7 Results (not processed corpus)

To analyze how a good (or bad) pre-treatment could impact the performance of the model, an untreated corpus was used to train a model (with the same hyper-parameters as model 5.) In this way, the corpus used here didn't receive any processing (filtering, cleaning, lowering, etc.) The results are the following.

For several values of *temperature:*

```
Source from train corpus:
Fuck this shit man, people thinkin' I'm crap, well

Temperature 0.5 :
Bursts decerve mutual, parisite 'Hes (soul), loon #punk Aristocrat 'Hes loon Ralph, (soul), (soul), mutual, 20can mutual, #foil #foil 20can (soul), (soul), (soul), mutual, Aristocrat

Temperature 0.8 :
stronger. loon amd #punk Bursts 'Hes loon #punk amd #punk amd (soul), Bursts amd decerve #foil #punk mutual, Laboratory 'Hes loon #punk #foil (soul), #punk

Temperature 1 :
Laboratory loon loon amd #punk 20can amd Ralph, Bursts ANYDAY #punk Aristocrat ANYDAY amd Ralph, 'Hes Ralph, wissobi Ralph, Ralph, 'Hes parisite 'Hes wissobi amd

Temperature 1.2 :
Laboratory decerve 20can stronger. decerve amd Bursts Bursts mutual, 20can mutual, parisite #foil amd (soul), headstrong. chainsaws, amd decerve 'Hes statistic weeee (soul), mutual, 'Hes

Temperature 1.4 :
Aristocrat Ralph, stronger. ANYDAY Ralph, stronger. amd wissobi Bursts ANYDAY mutual, #foil loon decerve decerve amd supremacist/ (soul), #punk Aristocrat loon amd 20can 20can 'Hes
```

Figure 18 Output for several temperature values: no pre-processed corpus.

For several values of *topk:*

```
Source from train corpus:
Maybe it's not just America, but Earth is who we need to teach,

Topk 5 :
20can Ralph, 20can amd amd 20can (soul), 20can 20can Ralph, decerve amd decerve decerve decerve 20can amd amd decerve 20can (soul), 'Hes decerve Bursts (soul),

Topk 10 :
loon Bursts #punk 20can decerve Aristocrat #punk (soul), 20can Ralph, (soul), headstrong. Ralph, Aristocrat wissobi Bursts (soul), #punk mutual, 'Hes 'It Aristocrat 'Hes escalation, weeee

Topk 15 :
amd headstrong. ANYDAY #punk wissobi giftcard statistic Ralph, loon amd Ralph, amd decerve (soul), amd wissobi (soul), ANYDAY Ralph, Ralph, 20can (soul), parisite #punk Aristocrat

Topk 20 :
loon chainsaws, 'Hes mutual, statistic escalation, 'It 'Hes amd supremacist/ 20can decerve Aristocrat weeee 'Hes Ralph, stronger. mutual, (soul), #foil mutual, Bursts #betta Ralph, Aristocrat

Topk 30 :
Rhea, giftcard 20can loon Rhea, sharks. Bursts fufilling ANYDAY (soul), Laboratory #betta giftcard stronger. wissobi DISNEY Bursts weeee amd wissobi decerve Ralph, preach 20can stronger.
```

Figure 19 Output for several topk values: no pre-processed corpus.

The output for all parameters looks very random and senseless. It seems also that the results have the same words each time.

The valid loss (12.57) and valid perplexity (288940.74) of this model, are very high. Indeed, the loss is almost three times higher than the average loss for all models and the perplexity is 2.5k times higher than the perplexity of model 5 (the best one).

However, looking at the loss evolution per epoch, it looks that the model has the usual behavior: the loss value gets decreased when the batched and epochs are increased:

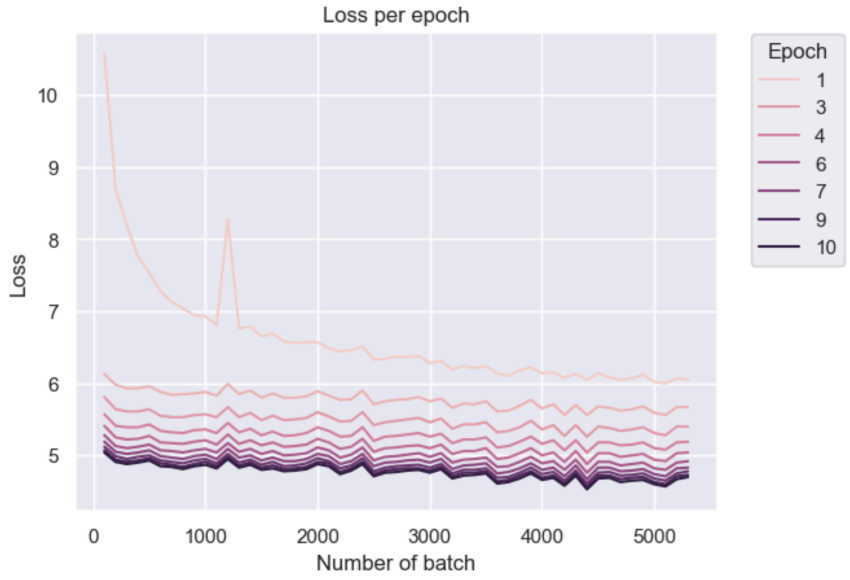


Figure 20 Loss evolution through epochs: no pre-processed corpus.

These results show how a bad processing (or null processing) can impact the performance of the model. Indeed, quantitatively, the valid loss and perplexity are the highest of all tested models; qualitatively, the output of the model, for all parameters, is senseless, repetitive and monotonous: none of the generate() parameters can improve the performance of the output of the model.

In this way, it's very important to have a consistent and congruent input data for the model, meaning, it's necessary to perform a good pre-treatement process to have good results.

III Optimization of the model

This section refers to the optimization of the basic LSTM model: sequence to sequence generation.

1 Potential ameliorations

In our project, we've developed a basic NLP model using an LSTM (Long Short-Term Memory) framework to generate rap battle text. Our model has successfully managed to create coherent text, which is a good start for a project of this nature.

However, we're aware that there's still a lot to improve. Currently, our model doesn't control syllable count, an important feature in rap battles known for their rhythmic patterns. This means our model's output doesn't always match the rhythmic and poetic feel typical in rap battles.

Another limitation is that our model can't yet generate battles with specific roles like a defined challenger and defender. This is a key aspect of rap battles, which often have a clear narrative structure and distinct roles. Without this, it's difficult for our model to produce rap battles with a clear start and finish, which are integral to creating engaging and realistic content.

2 Optimized Model for Generating Contextualized Defender Replies in Rap Battles: sequence-to-sequence

Our project has taken a significant leap forward with the development of an optimized model designed to generate responses in a rap battle context. This model is specifically tailored to act as a defender, responding to the lines thrown by a challenger.

The primary goal of this optimized model is to accurately simulate the role of a defender in a rap battle. This involves not just generating any rap text, but crafting replies that are contextually and stylistically aligned with the challenger's verses.

Model Structure and Training Corpus

In developing our LSTM model to simulate rap battles, we aimed to produce a system that could generate responses rich in context, especially from the Defender's perspective, in reaction to the Challenger's verse. This involved a blend of strategic corpus design, advanced technical implementation, and a dual approach to tokenization.

- Corpus Design and Training Methodology

The training corpus was carefully designed to reflect the real-world structure of rap battles. It included distinct segments for the battle's commencement, the Challenger's verses, the Defender's responses, and the battle's conclusion. This structure was crucial to train the model in a way that mimicked the natural flow and interaction of a rap battle.

The core of our training methodology involved pairing the verses of the Challenger with the corresponding replies from the Defender. This pairing was instrumental for the model to understand and generate responses that were not just linguistically accurate but also contextually relevant, maintaining the essence of a rap battle.

- Technical Implementation: Context Transmission and Data Processing

The technical implementation of the model was centered around ensuring the seamless transmission of context from the Challenger's verses to the Defender's responses. Each verse from the Challenger and the Defender was first tokenized, converting words into numerical tokens. We experimented with two tokenization methods: the straightforward word-level tokenization and the more efficient Byte Pair Encoding (BPE). While word-level tokenization provided a direct and simple mapping of words to tokens, BPE allowed us to efficiently handle the diverse vocabulary of the rap corpus by breaking down words into frequent subword units.

After tokenization, the verses underwent padding. This process ensured that all sequences were of uniform length, a necessary step for efficient batch processing in the model. The padding process, however, posed its challenges. We had to instruct the model not to train on the padding tokens, as they do not carry meaningful linguistic information. This was crucial to maintain the quality and relevance of the generated responses.

- Batch Processing and LSTM Layers

Batch processing presented another significant challenge. Due to the varying lengths of verses and the need to pair Challenger and Defender verses, we often faced issues with batch and output sizes. Adjusting batch sizes and managing these variations were critical steps in ensuring that the model could effectively process and learn from the data.

At the heart of the model were the LSTM layers, known for their ability to handle sequential data and inherent dependencies. As the LSTM processed the Challenger's verse, it updated its hidden state with each word, thereby accumulating a rich tapestry of information about the content, style, and context of the verse. This hidden state then became the foundation upon which the Defender's response was built, ensuring that the reply was not just a random collection of words but a contextually coherent and relevant counter to the Challenger's verse.

- Training Phases and Word Embeddings

Throughout the training phases, we exposed the model to various pairs of verses, enabling it to learn the diverse styles and nuances typical in rap battles. The model's parameters were continuously updated, guided by the training data, to enhance its ability to generate linguistically and contextually rich verses.

A crucial aspect of our model was the use of word embeddings, specifically employing Word2Vec on a non-pre-trained set. This approach allowed the model to develop its embeddings, tailored to the unique linguistic characteristics of the rap battle corpus, thereby enhancing its understanding of language nuances specific to rap battles.

In summary, we aimed our LSTM optimized model for simulating rap battles was a product of careful planning and execution in corpus design, context transmission, batch processing, and technical implementation. The model was supposed to capture the dynamism of rap battles, learning to generate responses that were not just accurate in language but also rich in context and creativity, closely mirroring the essence of real rap battles.

Challenges Encountered and Addressed

We encountered and addressed several notable challenges, essential for the integrity and functionality of our model. This section details these challenges, emphasizing the solutions and strategies we adopted to overcome them.

- Handling Verse Pairing and Resulting Sizing Issues

One of the most significant challenges we faced was related to the pairing of verses. Each rap battle consists of two parts: a verse from the Challenger and a response from the Defender. This pairing was crucial to maintain the context and flow of the battle. However, this introduced complexity in batch processing, as we had to ensure that each pair remained intact throughout the training process. The variability in verse lengths further complicated this issue, leading to difficulties in batching and inconsistencies in input and output sizes. These discrepancies often resulted in mismatches during training, validation, and testing phases, requiring extensive debugging and adjustments to ensure that the model's output aligned correctly with the target sequences.

- Padding and Its Impact on Training

To manage the variable lengths of verses, padding was implemented. However, this introduced the challenge of the model potentially learning from these padded sequences, which held no actual linguistic or contextual value. To mitigate this, we employed techniques to ensure the model recognized and ignored these paddings during training. This was crucial for maintaining the quality and relevance of the model's learning process, ensuring that it focused solely on meaningful content.

- GPU Constraints and Training Duration

Another significant challenge was the computational demand of training the LSTM model. Given the complexity of the model and the size of the data, the training process was GPU-intensive. On average, training a single model iteration took approximately 30 minutes. This duration imposed limitations on the number of experiments and adjustments we could feasibly conduct within a given timeframe. Managing these resource constraints required careful planning and optimization of our training processes.

To address these challenges, we adopted several strategies:

- Dynamic Padding: We implemented dynamic padding, ensuring that padding was applied just enough to make the sequences in a batch uniform in length, without adding excessive padding.
- Ignoring Padding in Loss Calculation: During the loss calculation phase, we modified our approach to ensure that the padding tokens did not contribute to the loss. This was crucial in guiding the model to learn from the actual content of the verses rather than the artificially added padding.
- Optimized Batch Processing: We refined our batching process to accommodate the paired nature of the verses and the variability in their lengths, ensuring efficient and contextually coherent training.
- Resource Management: Given the GPU constraints, we optimized our model's architecture and training parameters to balance between performance and computational feasibility.

These challenges and our approaches to addressing them were integral to developing a robust and effective LSTM model for rap battle simulation. The experience highlighted the importance of meticulous data management, strategic model design, and efficient resource utilization in the field of natural language processing and AI-driven content generation.

3 Results

We trained 6 different models (3 for each tokenization) with varying parameters to find the optimal configuration.

Here the best result that we got with the following hyper-parameters:

- Embedding Size: 200
- Hidden Size: Varied across models (200, 256)
- Layers: 2
- Dropout: Varied (0.3 to 0.8)
- Epochs: Varied (3 to 15)
- Learning Rate: Varied (0.001 to 0.01)
- Weight Decay: Varied (1e-4 to 1e-3)

- Word tokenizer

INPUT: challenger_verse = "i can dunk i can rap how can you do this to me like that put on that cap"

OUTPUT: Defender's response with temperature 0.7 and topk 30:
not but but with to not your have am be be your i and with me you but for

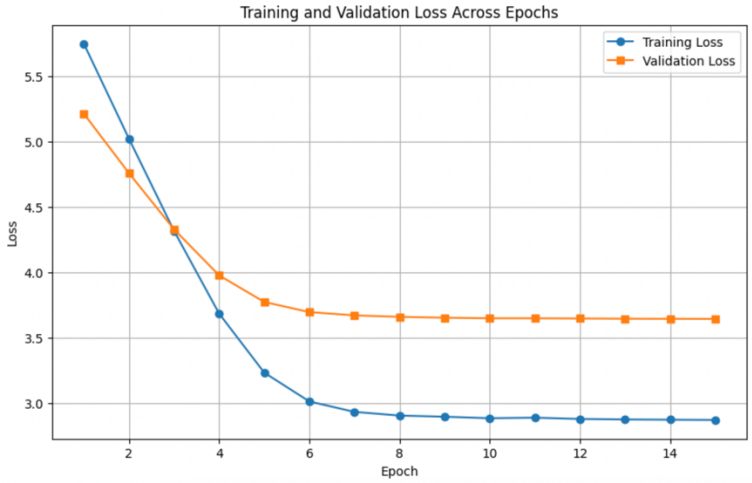


Figure 21 Training and Validation Loss vs. Epoch

- BPE Tokenizer

INPUT: challenger_verse = "i can dunk i can rap how can you do this to me like that put on that cap"

OUTPUT: Defender's response with temperature 1.0 and topk 20:
bullshit hurts <pad> sos fartface hered artic ali graz underneath otes
> omar confucian stucked eos academic minnie <pad> istry < sos tuned eos pos
sos < and zie tip thigh crowbar flo soldier kinesis yon hunt eyelash ris <pad>

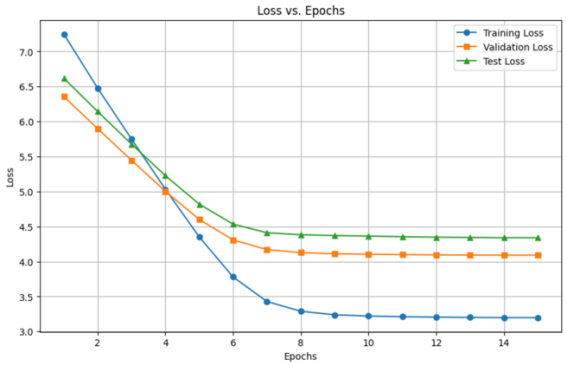


Figure 21 Training and Validation Loss vs. Epoch

General Performance:

The model demonstrated promising aspects, as indicated by the convergence of training, validation, and test loss, which suggests that it learned to some extent the underlying patterns in the data without overfitting.

However, despite the positive indications from the loss metrics, the actual output of the model revealed significant shortcomings. The responses generated often lacked coherence and did not consistently align with the context provided by the challenger's verse. This was exemplified by the output using both word-level and Byte Pair Encoding (BPE) tokenization methods, where the presence of placeholder tokens like '<pad>' and sometimes disjointed or incoherent sequences of text highlighted the need for further refinement of the model's ability to generate meaningful and contextually relevant text.

The discrepancies between the loss metrics and the quality of the generated text suggest that while the model can minimize error in a statistical sense, it struggles to produce outputs that are qualitatively aligned with the stylistic and contextual nuances characteristic of rap battles. The challenges encountered with batch size management, GPU constraints, and the handling of variable verse lengths contributed to these issues and underscore the complexity of creating a model that not only learns language patterns but also creatively emulates the dynamic interplay of a rap battle.

The model's current performance indicates that further enhancements are necessary to improve the quality of generated responses and ensure that they not only exhibit grammatical coherence but also embody the creativity and context awareness essential to the rap battle domain.

3 Future Enhancements and Considerations

In considering the future enhancements for our optimized LSTM model, a straightforward approach would be to make regular checkpoints during the training process to provide insights into the model's learning patterns, revealing when and why it may deviate from expected performance. This approach allows for precise tweaks to be made where they're needed most.

Context is crucial in language models, and thus, employing context-aware embeddings such as Google's word2vec can provide our model with a deeper understanding of language nuances. These sophisticated embeddings are trained on extensive datasets and are adept at capturing the subtle meanings of words based on their context.

Moreover, the cleanliness of the training corpus significantly influences model performance. An extra round of thorough cleaning to remove any noise or irrelevant data can improve the model's learning efficiency and output quality.

Finally, the model could benefit from extended training with a variety of parameters. Experimenting with different learning rates, batch sizes, and epochs can help find the optimal settings that yield the best performance. This process is essential for fine-tuning the model's ability to generate relevant and engaging rap battle verses.

Optional part

Integration inside a 3D engine

Figure 21 Introduction screen for our battle experience

In our recent project, we've taken the initiative of integrating our NLP model into the Unity 3D engine. To achieve this, we set up a local server embedded and directly launched within the Unity application. This server is responsible for loading our LSTM-based NLP model and waiting for input from the Unity interface.

When Unity needs to generate rap battle text, it encodes a string into bytes and sends this data to our Python-based server. On the server side, we decode this string, use it as input to our NLP model, and then generate a response. This generated string is then encoded back into bytes and sent to the Unity application, where it is displayed to the user.

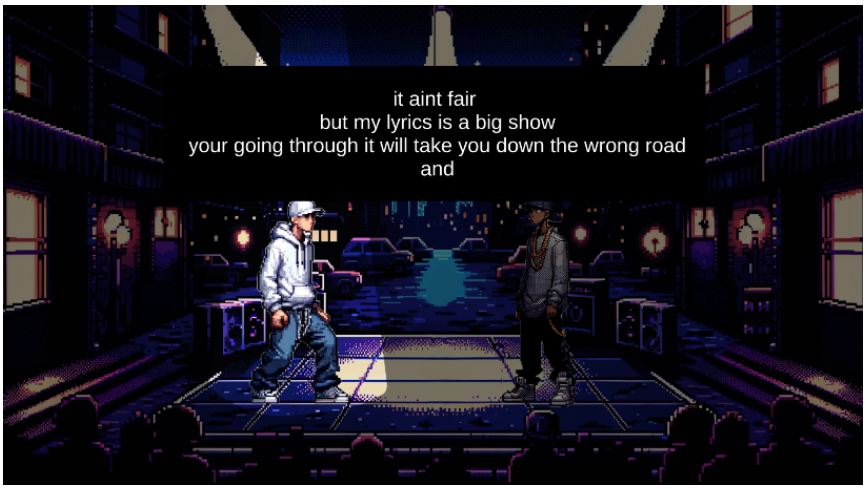


Figure 22 Screenshot of a rap battle sentence

This integration process was not without its challenges. Finding a way to make Python and Unity interact smoothly was a significant hurdle. We spent some time figuring out the best method to send and receive data between these two different environments.

One important aspect of our setup is that everything runs locally on a computer. While this ensures no dependence from Internet, it also means that the performance of our NLP model is tied to the local machine's resources. This can lead to slow response times, which is a key area we need to improve, especially for real-time experiences in gaming. Moreover, like described in the previous part, our model is limited, which can lead to strange punchlines inside the context of a game.

Despite these challenges, integrating our NLP model into Unity has been a rewarding experience. It demonstrates the potential of using advanced language models to enhance interactive experiences in games and other applications. This integration opens up exciting possibilities for future projects where NLP models can add a unique and interactive element to digital experiences.

Figure 23 Rap battle experience

Conclusion

Our project to build a natural language processing model for rap battles has been insightful. We approached the data organization with a focus on maintaining the storytelling of rap, organizing our datasets sequentially. This was crucial in keeping the conversational flow of the battles.

We made a point to save the model at each stage of training. This way, we could pick up where we left off without redoing work, saving time and energy. Tweaking the learning rate and other hyperparameters turned out to be significant, as these adjustments often meant the difference between a mediocre model and a high-performing one.

When it came to generating the raps, we tried out various methods. Some were about crafting an entire line in one go, while others built up the rap word by word. Adjusting the settings for text generation was a process of trial and error to find what worked best. After the model produced text, we also did some cleanup to make sure the final output was polished and flowed well.

Choosing the best model wasn't just about looking at the loss numbers; we had to read and judge the text itself. We had to be cautious about overfitting and ensure our model was learning effectively. Surprisingly, we found that the model's loss rates didn't always match the quality of the text it generated—low loss didn't necessarily mean good output.

In conclusion, this project has taught us that building an NLP model to generate creative content is as much about the technical details as it is about the artistry of language. It’s about finding the right balance to create a model that not only functions well but also produces text that is truly engaging.

Sources

1. **https://drive.google.com/drive/folders/1xLk7fSFD33m78KwZL0ae_YrY8Eu2DoqM**
2. **https://github.com/jaimeMontea/MS_IA_NLP**