

1 Introduction

The field of Natural Language Processing has experienced a major transformation in recent times, mostly because of the rapid advancements made in deep learning architectures. This rapid advancement has not only improved NLP systems performance but also opened the door for innovative uses in a variety of fields. Due to the field’s continuous need of innovation, once-difficult tasks like text generation, sentiment analysis, and machine translation have now reached previously unreachable levels of accuracy and fluency.

These days, NLP technologies are clearly embedded in our daily routines. Via the use of natural language, these tools—which range from sophisticated chatbots and recommendation systems to virtual assistants like Alexa and Siri—have emerged, enabling smooth communication between people and machines. The link that exists between technology and people highlights how powerful NLP can be in influencing how we work, communicate, and engage with information.

Examining the development of computational creativity makes it interesting to revisit established discussions about the role of machines in the creative process. In the past, people have questioned whether algorithms and machines are capable of creativity and imagination. Even if early critics expressed doubts about the creative capacity of machines, current AI systems continue to put doubt on these ideas by generating outputs that frequently confuse content created by humans and machines.

The creation of song lyrics based on genre and artist names is one of the applications of NLP that we will be investigating in this paper. We seek to understand the complexity of computational creativity and demonstrate how AI can support and improve artistic expression by utilising the power of innovative neural networks and transfer learning techniques. By pushing the limits of what is possible in the field of NLP and beyond, we seek to make a contribution to the continuing conversation about the connection between technology and creativity.

2 Motivation

Natural Language Processing and the creative arts are two separate but related fields that are the driving forces behind this research. There is a strong chance to investigate the use of language technology in the field of artistic expression, particularly in the creation of song lyrics, as developments in NLP keep rising.

For many years, audiences have been fascinated with music as a universal means of human expression, and it has been used to communicate feelings, stories, and cultural identity. In the world of music, lyrics are essential for influencing the listener’s experience, promoting feelings, and delivering deeply personal messages. Lyric writing is an area typically connected with human creativity, requiring a special combination of linguistic skill, emotional intelligence, and creative intuition.

But even with songwriting and lyricism’s history, there are still obstacles and restrictions that prevent this art form from becoming more widely accessible and collaborative. Traditional lyric writing techniques frequently depend on the skill and imagination of talented lyricists, which restricts the range of voices and viewpoints that are represented in the music industry. Additionally, writing original lyrics can take a lot of time and resources, which can be a barrier for musicians and content producers.

In this context, reinventing the lyric-generation process and expanding access to creative expression through the application of NLP technologies offers a new potential. With the use of large-scale language models and state-of-the-art machine learning algorithms, we can enable songwriters, musicians, and other content producers to explore new creative directions.

Moreover, based on genre and artist names, the capacity to produce customised and contextually appropriate lyrics offers up new opportunities for improving listener participation, customising music

suggestions, and producing engaging storytelling experiences. By incorporating NLP approaches into the music industry, we may transform the way we listen to and engage with music, promoting greater diversity, variety, and originality in the musical landscape.

3 Literature Review

An end-of-term review of a bi-directional LSTM song generator for creating song choruses is presented in this study [?]. Although the creation of poetry has been studied for many years, stylistic output came to light in the mid-1990s when NLP, AI, and machine learning came together. But a lot of today's text creation algorithms don't really have a distinct author voice or style. The relevance of author authenticity is emphasised in the study, particularly for applications such as language studies and customer service systems. The authors recommend feeding a suitable corpus that mimics the intended author's style to the predictive model in order to enhance the quality of generated material. They also stress how crucial it is to include details about relationships in the text. To create unified choruses, their models integrate multiple NLP methods and are based on customised datasets. All things considered, the system was successful in linking several NLP tools and producing choruses that could be linked to a certain musician. But there's always room for development, especially when it comes to boosting style capture and strengthening the generated documents' readability and consistency. In later revisions of the system, the authors propose improvements including adding a Rhyme Analyzer component and striking a balance between lyrical flexibility and grammatical structure.

This research paper [?] explores the possibility of creating lyrics for particular musical genres using deep learning techniques. Although recurrent neural networks (RNNs) or gated recurrent units (GRUs) have been used in computational research on linguistics to generate lyrics, this study uses an LSTM network to generate lyrics that are particular to a given genre. In order to examine language similarities, differences, and the effectiveness of the LSTM network, the authors compare the generated lyrics to other genres and the training set, evaluating them using a variety of linguistic metrics. The findings show that the LSTM model produces both pop and rap lyrics with good accuracy, with average line length and word variance across songs and genres being closely matched. Possible areas for research include delving into more basic models such as Markov Models and examining the application of GRUs to speed up training. The authors suggest training models across a variety of genres in order to efficiently capture linguistic structure similarities. The research highlights the possibility of pre-trained models being able to adjust to various linguistic styles with increased processing power, even in spite of computational constraints. It inspires more research into the creation of musical lyrics and beyond, especially when it comes to modifying generic models to fit different genres and writing styles. Finally, the paper highlights the importance of capturing not only semantic but also stylistic differences in text generation tasks.

The paper [?] explores the relationship between machine learning and the production of creative content, with a specific emphasis on song lyrics. The introduction, which draws on Ada Lovelace's doubt over computers' ability to create unique content, sets the scene by reflecting on past discussions regarding the creative potential of algorithms and machines. Despite this mistrust, the study highlights the unexpected outcomes of machine learning algorithms, leading to a reconsideration of the place of computers in creative industries. The author wants to investigate how machine learning—more especially, the GPT-2 model—can be used to write original music lyrics. As part of the technique, a dataset with over 122,000 carefully chosen song lyrics and related metadata was cleaned up. Metrics such as repetition score and TF-IDF vectors are used to analyse the song lyrics' length, word usage, and musical genre. The steps involved in creating song lyrics with the GPT-2 model are explained in depth, emphasising how crucial the volume of training data is to the creation of meaningful and diverse lyrics. The absence of training data for specific musical genres presents difficulties and requires innovative methods of assembling datasets. In spite of these obstacles, the study offers suggestions for improving the training procedure to raise the calibre of lyrics that are produced. Following studies efforts may encompass expanding the range of song lyrics to encompass a greater variety of musical styles, enhancing metrics for contrasting lyrics produced by machines and humans, and integrating sentiment analysis to describe lyrics from other musical genres. In summary, the study highlights how machine learning might enhance creative processes and encourages more research in this area.

This work [?] investigates the possibility of creating poetry writings and song lyrics using GPT-2, an advanced neural network design. Although a lot has been accomplished in the field of Natural

Language Processing (NLP), especially in the areas of machine translation and text generation, there are still difficulties in producing texts that contain poetic devices like metaphors, metonymy, and paraphrases, especially in the musical genre. The performance of a GPT-2 model that has been refined using datasets of English and Portuguese lyrics is examined by the authors. They assess the produced writings according to their similarity to lyrics written by humans as well as their spelling, grammar, and semantics. The GPT-2 models successfully produce syntactically valid and semantically consistent musical/poetic writings, despite frequent spelling errors and hardware resource limits. The study argues that existing measures such as misunderstandings are insufficient for evaluating artistic aspects and admits a subjective way of judging the quality of such writings. The authors stress that the intention is to enhance human composers work rather than take their place. In order to increase the quality of generated writings, future study will define more relevant metrics for evaluating poetic poems, run further trials for analysis, and investigate the integration of semantic content into datasets. Overall, the study opens the door for more research into the subject of poetic text production by indicating that GPT-2 has special potential for producing text for a variety of genres and purposes.

In the final paper [?], a Large Language Model (LLM) specifically designed for song composing is presented: SongComposer. In contrast to earlier music-focused LLMs, which handle music as quantized audio data, SongComposer uses symbolic song representation to create lyrics and melodies in a manner that resembles human composition methods. This novel method has benefits in terms of accurate representation, token efficiency, and output that is readable by humans. The authors created SongCompose-PT, a large dataset of lyrics, melodies, and paired lyrics-melodies in both Chinese and English, in order to train SongComposer efficiently. SongComposer learns core musical knowledge, including melody-lyric alignment, through pre-training on this dataset. The study shows that SongComposer outperforms advanced LLMs like GPT-4 on a variety of song-related tasks, including text-to-song creation, melody-to-lyric generation, song continuation, and lyric-to-melody generation. SongComposer delivers efficient and effective lyrics-to-melody generation by combining both tasks into a single model and using text-based melody representation. Overall, SongComposer is a major step forward in the use of LLMs in the music industry, giving people who haven't studied music much of a platform to express their creativity. However, during its development and implementation, ethical concerns about copyright infringement and the misuse of intellectual property should be carefully taken into consideration.

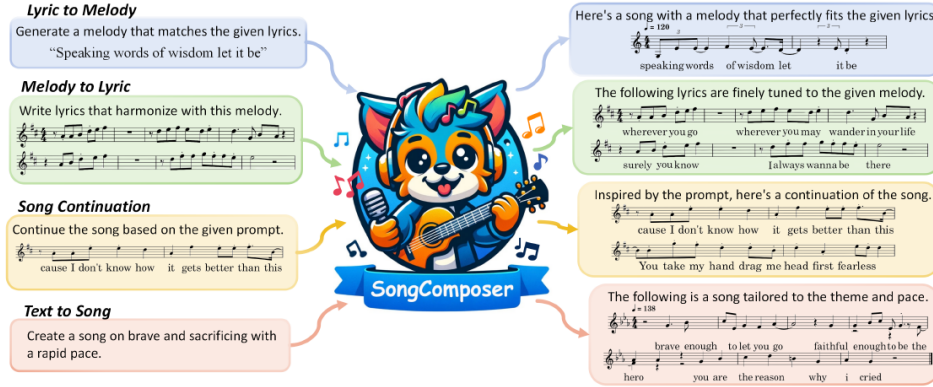


Figure 1: Overview of the song-related instruction-following generation by SongComposer

4 Data Analysis

Firstly, we used the Spotify 1 million songs dataset from kaggle [?] by Shrirang Mahajan.

4.1 Basic data exploration

To begin with, we started the data exploration by transforming the .csv file into a pandas dataframe. We gathered basic information about the dataset such as the total number of entries (57650 entries). Then we discovered the columns/features of our dataset ['artist', 'song', 'link', 'text'], where artist is the name of the singer, the song is the song's title, the link is spotify's link to the song and finally text signifies the lyrics of the song.

Then we discovered some statistics about our dataset such as the uniqueness of our attributes where we found that we have 643 unique artists, 44824 unique songs, 57650 unique spotify links and finally 57494 unique song lyrics.

4.2 Normalization

In the section, an initial assessment was conducted to identify any null values within the dataset. It was determined that the dataset is devoid of any null values. Subsequently, a detailed examination was undertaken to ascertain the presence of complete row duplicates. This analysis revealed an absence of such duplicates, indicating the uniqueness of each entry. However, upon closer inspection of individual feature statistics, it was observed that multiple songs share identical titles, albeit possibly performed by different artists. This phenomenon may suggest the existence of song remixes or cover versions within the dataset. Following the initial data examination, the subsequent step involved employing



```
song
Have Yourself A Merry Little Christmas    35
Angel                                     28
Hold On                                   27
Home                                       27
I Believe                                 26
..
Just Be Good (Bushman Dub)                1
Kingston Town                             1
Kiss And Say Goodbye                      1
Lamsbread                                 1
Generation                                1
Name: count, Length: 44824, dtype: int64
```

Figure 2: Screenshot to show the song name repetitions

tokenization, segmentation, and lemmatization methods. However, prior to executing these processes, the dataframe was transformed into an array of string arrays containing the lyrics of each song. This preparatory step was imperative to ensure the accurate processing of the data. By converting the dataframe in this manner, the potential consequence of tokenizing each individual letter within a word was mitigated. This preemptive measure served to alleviate computational burden and prevent the generation of extraneous and redundant information.

Subsequent to the initial preprocessing steps, extraneous words and punctuations, such as common stop words like "a" and "the", were systematically removed from the text data. This process, known as stop words removal, effectively pruned the dataset of non-essential linguistic elements, thereby enhancing the quality and relevance of the remaining text for subsequent analysis.


After discovering the presence of words starting with uppercase letters within the dataset, we standardized our vocabulary to lowercase. This unified approach facilitates easier detection, comparison, and analysis of the data.

After identifying the presence of newline characters, spaces, and other whitespace characters within our vocabulary, we proceeded to remove them. This action was essential to streamline the analysis process and ensure accurate results.

4.3 Insights

4.3.1 Statistical Insights

We commenced our analysis by examining the class distribution within the dataset. Notably, among the dataset's features, the "artist" attribute exhibited the least number of unique entries, suggesting its suitability as the primary classifying category. The utilization of the "artist" feature as the primary classification criterion was deemed the most robust approach. As depicted in the figure below, the top 10 classes within the dataset were identified. For instance, "Donna Summer" emerged as the foremost class, boasting a total of 191 unique entries within the dataset. This visualization provides a comprehensive overview of the distribution of classes, offering insights into the dataset's structure and composition.



artist	
Donna Summer	191
Gordon Lightfoot	189
Bob Dylan	188
George Strait	188
Loretta Lynn	187
Cher	187
Alabama	187
Reba McEntire	187
Chaka Khan	186
Dean Martin	186

Name: count, dtype: int64

Figure 3: Screenshot of Top 10 from the most frequent class

Following our initial analysis, we determined that computing TF (Term Frequency), IDF (Inverse Document Frequency), and TF-IDF scores would provide valuable insights into our dataset. To begin, TF computation entails counting the occurrences of a specific word in the vocabulary of an individual song.

After computing the TF scores, we proceeded to calculate the IDF scores, which offer a measure of how unique or rare a word is across all documents in the dataset. Essentially, IDF evaluates the significance of a term by inversely weighting words that occur frequently across the entire corpus. This weighting helps prioritize terms that are less common but potentially more informative.

Having computed both TF and IDF scores, we merged them to derive the TF-IDF scores, which offer a comprehensive evaluation of word importance within individual documents relative to the entire corpus. TF-IDF underscores terms that demonstrate high frequency within a specific document (TF), while also considering their rarity across all documents (IDF). This integrated metric effectively captures the distinctive relevance of each term in the context of the entire dataset, allowing us to prioritize words and phrases that are less common in the corpus. This strategic emphasis on less common terms contributes to the creation of more novel and engaging lyrics. As depicted in the following figure, the word 'Kind' demonstrates a notably high TF-IDF score in the first song, indicating its frequent occurrence within the song itself coupled with its rarity across the entire dataset. Conversely, the term 'special' exhibits a comparatively lower TF-IDF score, suggesting its moderate frequency within the document alongside its modest rarity across the corpus. This distinction underscores the nuanced evaluation provided by TF-IDF, allowing for the identification of terms that are both commonly encountered within specific documents and distinctive across the broader dataset.

```
{0: {'kind': 9.044448916049255,
'girl': 7.3677222729989325,
'?': 5.007201770982746,
'fine': 4.760091081653636,
'blue': 4.200247046584387,
'believ': 3.9197522596349987,
'face': 3.770781503107308,
'walk': 3.7592859811103083,
'leav': 3.7514418549629873,
'fellow': 3.551284297088087,
'look': 3.2302736818463433,
'squeez': 3.2057048630523988,
'feel': 3.10595302525842,
'lucki': 2.8909811036513897,
'park': 2.8453994764184483,
'special': 2.8438192643103357,
```

Figure 4: Screenshot of example of TF-IDF Scores

4.3.2 Graphical Insights

In the figure presented below, we observe the top 30 artists ranked by song frequency. This graphical representation aids in discerning the distribution of songs and the prominence of each class discussed in Section 4.3.1.

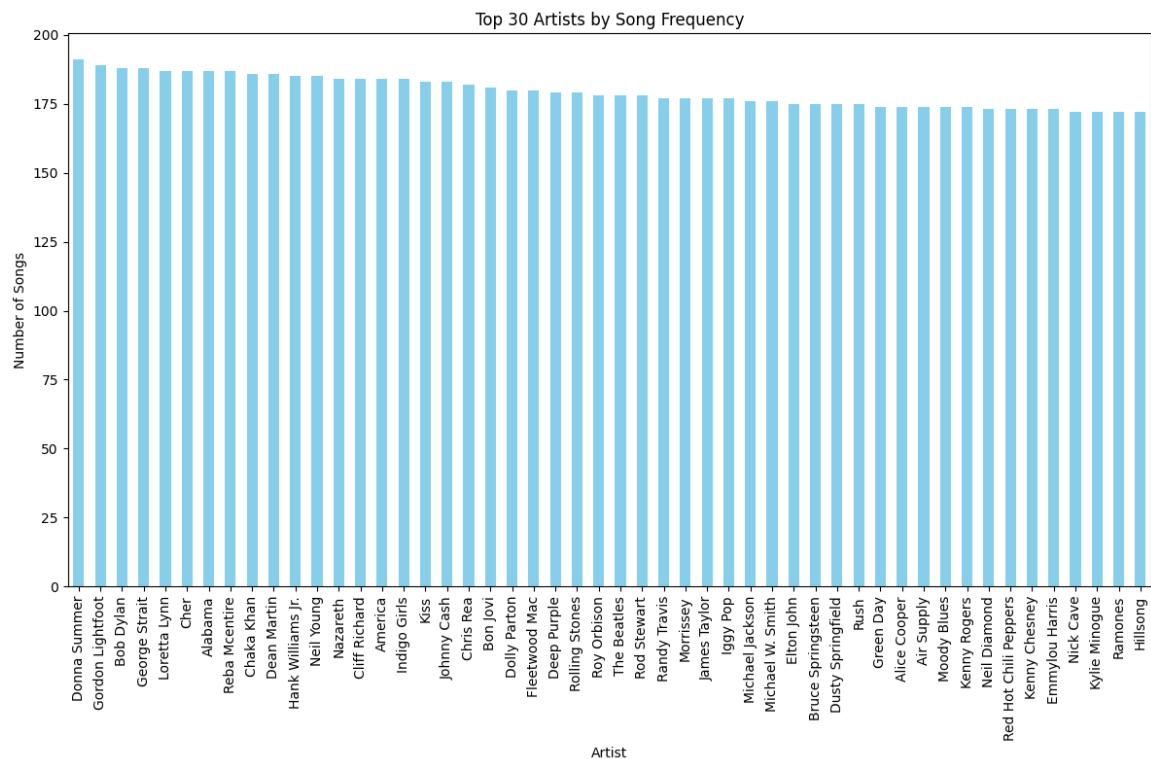


Figure 5: Screenshot of Top 30 most frequent artists

Following our initial investigation, we turned our attention to the lyrical content of the songs. Our aim was to determine the average number of words typically found in a song after the model training process. This analysis provides valuable insight into the expected output length for each song generated by the trained model.

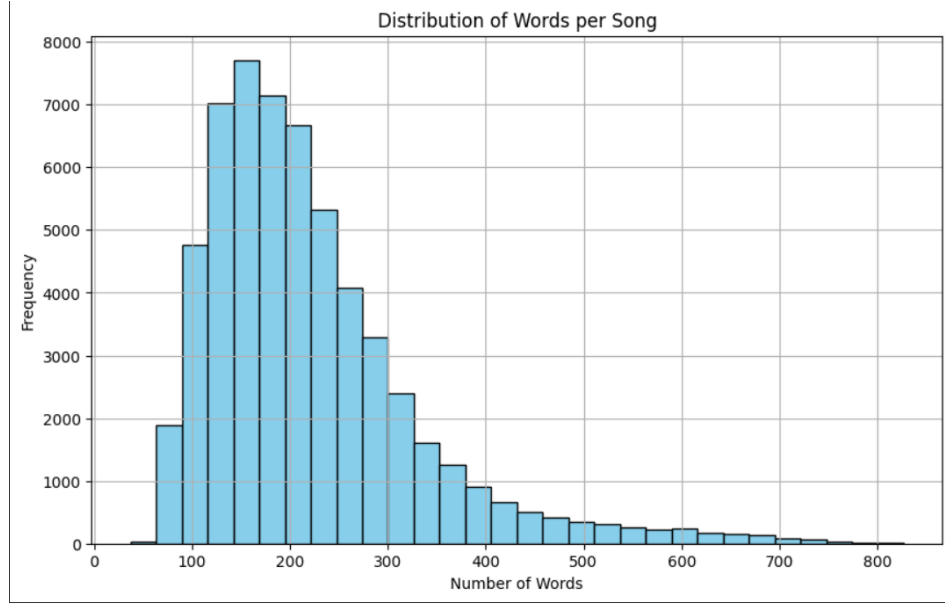


Figure 6: Distribution of words/song

$$\text{Repetition Score} = \frac{\text{Number of Repeated Words}}{\text{Total Number of Words}}$$

This formula quantifies the repetition score, which is the ratio of the number of repeated words in a song to the total number of words present in that song. It provides a metric for assessing the degree of word repetition within individual songs.

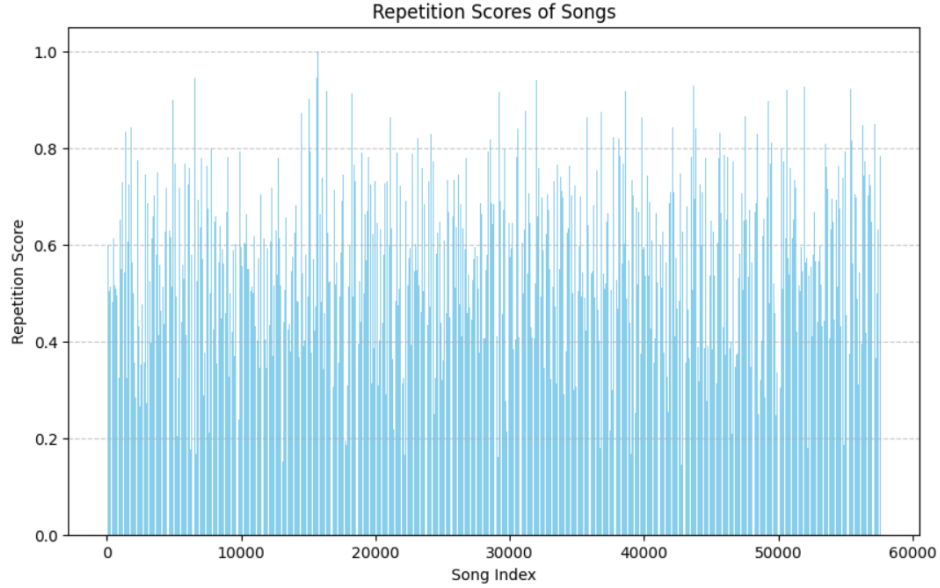


Figure 7: Repetitions of words in lyrics

4.4 Limitations

- Conducting a spelling check on the entire dataset proves to be computationally demanding and often leads to IDE crashes. With an average processing time of 0.3 seconds per song, the task necessitates approximately 4 hours to complete for the entire dataset. However, a pertinent consideration arises: is it imperative to perform a spell check on song lyrics? The distinction between expressions such as "oh" and "ohhhhh" may not be perceived in the same way by humans.
- Another noteworthy limitation of this dataset is its sparse feature set, which lacks crucial attributes pertaining to songs such as genre, duration, musical key, release date, among others. As a result, leveraging web scraping techniques or integrating with external datasets available online becomes imperative to extract or augment this essential information. Such endeavors are necessary to enrich the dataset and enhance its utility for comprehensive analysis and insights generation.

5 Methodology

5.1 Research and Exploration

RNNs, LSTMs, and GRUs were all the possible options for creating a neural network architecture for the song generation task, but each has advantages and disadvantages. Long-term dependencies are a challenge for standard RNNs and this is essential for capturing the lyrical structure of songs. Although they need more processing power and training data, LSTMs perform well in the domain of capturing the long-term dependencies. A ideal choice is provided by GRUs, which handle long-term dependencies similarly to LSTMs but with a simpler design that requires less resources and allows for faster training. While GRUs are more economical and frequently achieve close performance, LSTMs may have a slight advantage in more complicated problems. Since our dataset is complicated and we are limited with the computational power, GRUs were the best option for choice.

5.2 Our own Model Architecture

5.2.1 Preparing the dataset

Before creating our model, we had to clean the dataset based on the analysis that we gathered in Section 4. To begin with, any song that had a repetition score greater than 0.7 was dropped since it would be redundant if the song lyrics is repeated with a rate of 70%. This action resulted in dropping around 14k rows of data. After that, based on insights in Section 4, we chose the artist with the most number of songs to be able to use their songs for training and testing the model since the dataset was huge and wanted to narrow down our options. This means that the artist name won't be put into consideration while training. The artist of choice was "Donna Summer" since she had 191 songs in the dataset. Finally, we computed the most frequent number of words in the lyrics created by Donna so that we could know the number of words that will be used in the padding process.

5.2.2 Embeddings Creation

We used word2vec to generate our word embeddings because it efficiently captures semantic relationships between words in a high-dimensional space. This method analyzes large text in corpus to understand how words are used in context. Words with similar meanings end up positioned close together in this embedding space. This property is crucial for our model, as it allows it to learn these semantic relationships and translate them something meaningful that can be used in song generation tasks.

To begin with, we split our lyrics column into X_train and Y_train where X_train represents the lyrics without the last word and Y_train represents the last word that should be generated by the model.

Following the preprocessing of our data, we initialized our Word2Vec model with a specific set of parameters:

- **Sentences:** The input to our model was the 'text_cleaned' column from our DataFrame, which had undergone a series of preprocessing steps.
- **Window Size:** Set to 5, this parameter defines the context window for our Word2Vec model. The model uses the surrounding words to learn the representation of a target word. In this case, the model considers 5 words to the left and 5 words to the right of the target word.
- **Min_count:** This parameter is the minimum frequency count of words. The model will ignore all words that appear less than this number of times in the corpus. By setting it to 1, we ensure that even words that appear only once will be included in the vocabulary.
- **Workers:** This parameter, set to 4, is the number of worker threads used to train the model. This can speed up training on multi-core machines.
- **Sg:** This parameter chooses the training algorithm. If sg=1, the model uses Skip-Gram, if sg=0, the model uses Continuous Bag of Words (CBOW).
- **Vector Size:** This parameter is the dimensionality of the word vectors. It defines the size of the output vectors from this layer for each word.

Following the initialization of our Word2Vec model, we embarked on the creation of our word dictionary. This step was instrumental in facilitating the compilation of the model, as well as the extraction and generation of text.

We constructed three distinct types of dictionaries for this purpose:

- **word_embeddings_dictionary**: This dictionary maps each word to its corresponding embedding array.
- **dict**: This dictionary maps each word index to its corresponding embedding array.
- **new_dict**: This dictionary maps each word index to its corresponding word.

Subsequently, we initialized the lookup table using an output mode of "one-hot" encoding. This step was crucial in creating the vocabulary list based on the 'Y_train' values.

Next, we fed the 'X_train' dataset with the encoded matrix of each corresponding song, utilizing the 'word_embeddings_dictionary' that was created earlier. To ensure uniformity in the length of the input data and to avoid any irregularities in the dataset fed into the model, we padded the 'X_train' dataset.

Finally, prior to the creation of the model, we initialized the 'train_df' and 'test_df' datasets in an 80-20 split. This allowed us to compile and fit the model using the training data, and subsequently test the model using the test data. This rigorous approach ensures the robustness and reliability of our model.

Subsequent to the aforementioned steps, we implemented padding on the 'X_train' dataset. The purpose of this operation was to ensure uniformity in the length of the lyrics across all data points. This uniform length was determined during the preparation phase of our dataset for training and testing. The padding process is crucial as it allows our model to handle input data in a consistent manner, thereby enhancing its ability to learn from the data and make accurate predictions. This step further underscores our commitment to ensuring the robustness and reliability of our model.

5.2.3 Model Architecture

Our proposed model is a sequential keras one that uses GRUs and dense layers. It is a 4-layer model consisting of 2 GRU layers and 2 dense layers. The core of the model lies in the two GRU layers. The first GRU layer, equipped with `gru_units` number of processing units, analyzes sequences of lyrics, which are represented as word embeddings. This layer is configured to return sequences (`return_sequences=True`), allowing it to pass on information about the entire sequence to the next layer. The second GRU layer, also with `gru_units`, further processes the information but is set not to return sequences (`return_sequences=False`). This means it condenses the sequence into a single vector, capturing the essence of the analyzed lyrics. Following the GRU layers, two dense layers take over. The first dense layer has 64 units and utilizes the ReLU activation function. This layer performs a linear transformation on the input from the GRU layers and introduces non-linearity with ReLU, allowing the model to learn complex relationships between words within a sequence. The final dense layer is crucial for lyric generation. It has the same number of units as the size of the vocabulary. This layer employs the softmax activation function, which outputs a probability distribution over all possible words in the vocabulary. During generation, the model samples from this probability distribution, selecting the most likely word to continue the sequence. This process is repeated, with the generated word being fed back into the model as input for the next prediction, allowing it to generate a coherent sequence of lyrics that reflects the style of the analyzed input.

Model: "sequential_7"

Layer (type)	Output Shape	Param #
gru_14 (GRU)	(None, 224, 128)	53760
gru_15 (GRU)	(None, 128)	99072
dense_14 (Dense)	(None, 64)	8256
dense_15 (Dense)	(None, 109)	7085

=====
Total params: 168173 (656.93 KB)
Trainable params: 168173 (656.93 KB)
Non-trainable params: 0 (0.00 Byte)

Figure 10: Model Summary

Epoch 1/10	5/5 [=====] - 5s 123ms/step - loss: 0.6901 - accuracy: 0.0072
Epoch 2/10	5/5 [=====] - 1s 125ms/step - loss: 0.6670 - accuracy: 0.0072
Epoch 3/10	5/5 [=====] - 1s 122ms/step - loss: 0.5388 - accuracy: 0.0072
Epoch 4/10	5/5 [=====] - 1s 119ms/step - loss: 0.2464 - accuracy: 0.0072
Epoch 5/10	5/5 [=====] - 1s 119ms/step - loss: 0.1530 - accuracy: 0.0072
Epoch 6/10	5/5 [=====] - 1s 118ms/step - loss: 0.0973 - accuracy: 0.0072
Epoch 7/10	5/5 [=====] - 1s 120ms/step - loss: 0.0674 - accuracy: 0.0072
Epoch 8/10	5/5 [=====] - 1s 121ms/step - loss: 0.0559 - accuracy: 0.0290
Epoch 9/10	5/5 [=====] - 1s 127ms/step - loss: 0.0523 - accuracy: 0.1087
Epoch 10/10	5/5 [=====] - 1s 120ms/step - loss: 0.0509 - accuracy: 0.1087

Figure 11: Results

When trying to include more artist songs for training/testing in the dataset without including the artist name while running the model, we achieved lower accuracy values, indicating that each artist has his/her own taste.

```

Epoch 1/10
80/80 [=====] - 13s 110ms/step - loss: 0.1655 - accuracy: 0.0208
Epoch 2/10
80/80 [=====] - 9s 107ms/step - loss: 0.0073 - accuracy: 0.0513
Epoch 3/10
80/80 [=====] - 9s 109ms/step - loss: 0.0072 - accuracy: 0.0431
Epoch 4/10
80/80 [=====] - 9s 113ms/step - loss: 0.0072 - accuracy: 0.0466
Epoch 5/10
80/80 [=====] - 9s 110ms/step - loss: 0.0072 - accuracy: 0.0478
Epoch 6/10
80/80 [=====] - 9s 110ms/step - loss: 0.0072 - accuracy: 0.0462
Epoch 7/10
80/80 [=====] - 9s 112ms/step - loss: 0.0072 - accuracy: 0.0517
Epoch 8/10
80/80 [=====] - 9s 111ms/step - loss: 0.0072 - accuracy: 0.0493
Epoch 9/10
80/80 [=====] - 9s 113ms/step - loss: 0.0072 - accuracy: 0.0501
Epoch 10/10
80/80 [=====] - 9s 113ms/step - loss: 0.0072 - accuracy: 0.0525

```

Figure 12: Trial of increasing number of artists

5.2.4 Training and Evaluation

Upon the successful creation, summary review, compilation, and training of our model, we proceeded to the preparation of our testing data. We began this process by extracting the `text_cleaned` column from the `test.df` dataframe that we had previously established, and used this as input for our model.

The next step involved devising a strategy for managing the generation process. Our approach entailed iterating over all songs in the `test.df` dataframe, or a subset thereof, even if it comprised a single song. For each song, we extracted its `X` value, or its pre-processed text, and fed this into the model. We then looped over each song ten times to generate the subsequent ten words for each song.

In each iteration, after predicting the next word, we appended this word to the original song. This ensured that in each subsequent iteration, the model would take into consideration the newly added word. This iterative process of prediction and concatenation allowed us to generate a continuous sequence of words, thereby creating a coherent and meaningful output. This approach underscores our commitment to generating high-quality, contextually relevant content.

```

[['queen', 'day', 'queen', 'night', 'dressed', 'head', 'toe']]
1/1 [=====] 0s 33ms/step
1/1 [=====] 0s 32ms/step
1/1 [=====] 0s 36ms/step
1/1 [=====] 0s 28ms/step
1/1 [=====] 0s 63ms/step
1/1 [=====] 0s 29ms/step
1/1 [=====] 0s 29ms/step
1/1 [=====] 0s 31ms/step
1/1 [=====] 0s 30ms/step
1/1 [=====] 0s 37ms/step
queen day queen night dressed head toe oh oh oh oh oh oh oh oh oh oh

```

Figure 13: Song Generation Trial

As shown in the preceding figure, the model generated identical words for the next ten positions. This repetition is indicative of the model's low accuracy. It's important to note that the quality of the generated text is directly proportional to the model's accuracy. Therefore, enhancing the model's accuracy would result in more diverse and contextually appropriate word generation. This underscores the importance of model optimization and fine-tuning in improving the quality of the generated text.

5.3 Pre-Trained Model

The pre-trained model that we will be used in this section is T5 Conditional Transformer for Generation, where TF stand for TensorFlow and T5 is the Text-to-Text Transfer Transformer model by Google AI. The conditional generation one was used to tailor for the fact that we want to put the artist name into consideration when training the model.

For the pre-trained model, 2 approaches were used:

1. The regular `model.fit()` approach.
2. A manual approach of training a sequence-to-sequence model with Hugging Face Transformers.

To begin with, we'll start off by discussing the common steps between the 2 approaches such as data pre-processing, tokenization and so on, then each method will be discussed on its own.

5.3.1 Handling the dataset

Our initial approach involved splitting the "lyrics" column into two separate columns: "X" and "Y". Column "X" contained the first half of the song lyrics, while column "Y" contained the second half. Subsequently, the artist name was concatenated with the content of column "X" using a special separator token. This step aimed to incorporate artist information into the model's conditioning process.

Following this, we calculated the maximum sequence length for both the newly formed concatenated column (artist name + first half of lyrics) and the original "Y" column (second half of lyrics). Determining the maximum sequence length is crucial for the padding process, which ensures all sequences have a uniform length suitable for model training.

Finally, we converted the pandas DataFrame into a Hugging Face Dataset to facilitate further processing steps. This conversion resulted in a dictionary-like structure with three key attributes: "artist_name," "input_text" (containing the concatenated artist and first half lyrics), and "target_text" (representing the second half of the lyrics).

5.3.2 Tokenization and data pre-processing

We implemented a custom tokenization function to prepare the data for the model and the steps include:

1. Preprocessing the text: this adds special markers "<s>" and "</s>" to identify the beginning and end of sequences within the input and target text.
2. Tokenization and Padding: this utilizes the T5 tokenizer to convert the preprocessed text into numerical token sequences. It enforces a maximum sequence length by truncating longer sequences and padding shorter ones with a special token(0).
3. Preparing Labels and Decoder Input: this extracts the input IDs for the target text (labels) and creates a copy for the decoder input. It assigns a specific token ID, the padding token, as the starting point for the decoder during generation.
4. Shifting Decoder Input: this uses the teacher-forcing mechanism by shifting the target token sequence one position to the right. This means the decoder predicts the next word based on the previous word and a starting token during training.
5. Combining Outputs: The function combines the processed elements (input IDs, labels, and shifted decoder IDs) into a single dictionary named (model inputs) suitable for the training process.

After mapping the tokenize function to the entire dataset, we split this mapped dataset into training and test sets

5.3.3 The regular `model.fit()` approach

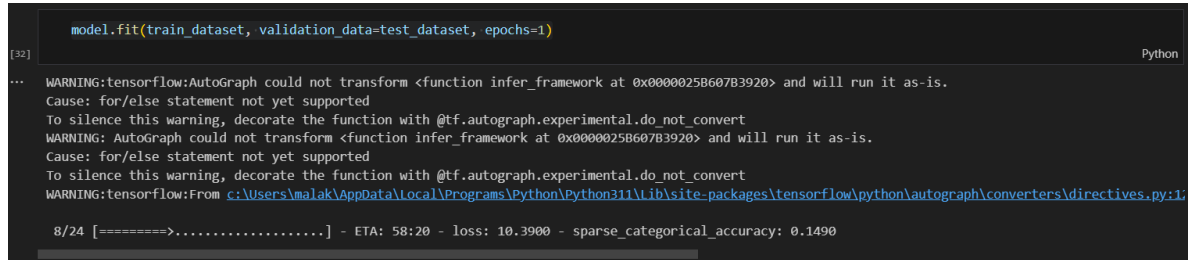
1. We converted both sets into the NumPy format using `.with_format("numpy")` since this format is widely used in machine learning due to its efficiency and making sure it is in a format suitable for the mathematical operations involved in training and evaluating our sequence-to-sequence model.
2. We defined important items such as the optimizer, the loss function and the accuracy metrics.
3. We then compiled the model to combines the previously defined optimizer (optimizer), loss function (loss), and metrics (metrics) for the training process
4. Finally, we then train the model on the provided training data, using the test data to monitor its performance on unseen examples.

5.3.4 Manual approach of training a sequence-to-sequence model with Hugging Face Transformers

1. A function was used to convert the dataset into a format that is compatible with TensorFlow. This conversion step ensures TensorFlow can efficiently process the data during training. The function also enables shuffling of the data within the dataset. Shuffling helps the model learn from a more diverse order of examples, improving its generalizability.
2. Two functions were then defined :
 - **train_step** : It retrieves a batch of data from the training dataset where this batch contains multiple examples, each with its input text, attention mask, shifted target text, and ground truth labels. It feeds the batch data through the model to perform a forward pass through the model's layers, where the model processes the input text and attempts to predict the continuation. It then calculates the loss by comparing the model's predictions with the actual labels in the batch. The loss represents the difference between the model's output and the desired outcome. Then, it utilizes a `tf.GradientTape` to record the operations performed during the forward pass. This allows it to calculate the gradients of the loss function with respect to the model's trainable variables. These gradients essentially tell the model how much each weight or bias needs to be adjusted to minimize the loss and improve its predictions. It uses the optimizer to update the model's weights based on the calculated gradients. The optimizer acts as a guide, using the gradients to adjust the weights in a direction that minimizes the loss. This weight update is the core learning step, where the model adapts its internal parameters based on the training data. Finally, it updates the accuracy metric to track how well the model performs on the training data over time.
 - **evaluate_step**: Similar to `train_step`, it retrieves a batch of data from the evaluation dataset. This dataset contains examples the model hasn't been explicitly trained on. It performs a forward pass through the model just like in the training step. The model processes the input text and predicts the continuation based on its learned patterns. Then, it calculates the loss by comparing the model's predictions with the actual labels in the batch. Finally, It updates the accuracy metric to track the model's accuracy on the evaluation data. This helps evaluate how well the model generalizes to unseen examples.
3. The most important step in this approach is the training loop where the code enters a loop that iterates for a predefined number of epochs. Each epoch represents a complete pass through the entire training dataset.
 - **Within each epoch**: The code trains the model on the training dataset using the `train_step` function. This function performs the essential steps for a single training step, including forward pass, loss calculation, gradient calculation, weight update, and accuracy metric update. It keeps track of the total loss and accuracy accumulated over the entire training epoch. It periodically prints informative messages about the training progress, including the current epoch number, average loss so far, and the current accuracy.

- **After each training epoch:** The loop switches to the evaluation dataset and uses the `eval_step` function to evaluate the model's performance on unseen data. It tracks the total loss and accuracy during evaluation. Finally, it prints the validation loss and accuracy for the current epoch, providing insights into how well the model performs on data it hasn't been explicitly trained on.

The training process was challenging due to the large dataset and computationally expensive model. Local training caused crashes, so the model was migrated to Kaggle's cloud platform with powerful GPUs to speed up training. Even with these resources, training the entire dataset was slow. To ensure the pipeline functioned correctly, the model was trained on a small sample (top 5 artists) and then successfully trained the model on the entire dataset using Kaggle's GPUs. This approach balanced efficiency with validating the pipeline's functionality.



```

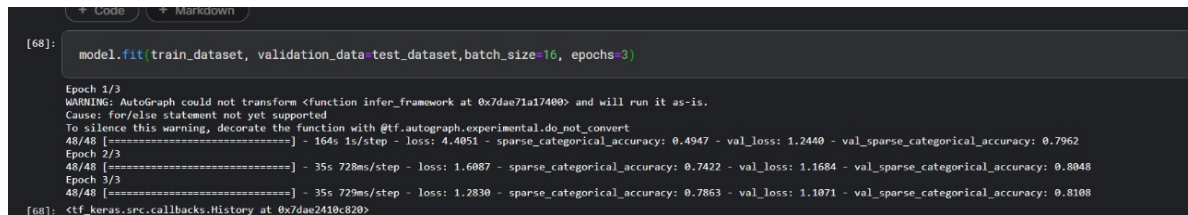
model.fit(train_dataset, validation_data=test_dataset, epochs=1)
[32]
... WARNING:tensorflow:AutoGraph could not transform <function infer_framework at 0x00000258607B3920> and will run it as-is.
Cause: for/else statement not yet supported
To silence this warning, decorate the function with @tf.autograph.experimental.do_not_convert
WARNING:tensorflow:AutoGraph could not transform <function infer_framework at 0x00000258607B3920> and will run it as-is.
Cause: for/else statement not yet supported
To silence this warning, decorate the function with @tf.autograph.experimental.do_not_convert
WARNING:tensorflow:From c:\Users\malak\AppData\Local\Programs\Python\Python311\Lib\site-packages\tensorflow\python\autograph\converters\directives.py:1
8/24 [=====>.....] - ETA: 58:20 - loss: 10.3900 - sparse_categorical_accuracy: 0.1490

```

Figure 14: Laptop Crashed after running on visual studio

5.3.5 Approach 1 Training and Evaluation

We figured out the pipeline is working on the top 5 artists and with fine results.



```

model.fit(train_dataset, validation_data=test_dataset, batch_size=16, epochs=3)
Epoch 1/3
WARNING: AutoGraph could not transform <function infer_framework at 0x7dae71a17400> and will run it as-is.
Cause: for/else statement not yet supported
To silence this warning, decorate the function with @tf.autograph.experimental.do_not_convert
48/48 [=====>.....] - 154s 15/step - loss: 4.4051 - sparse_categorical_accuracy: 0.4947 - val_loss: 1.2440 - val_sparse_categorical_accuracy: 0.7962
Epoch 2/3
48/48 [=====>.....] - 35s 72ms/step - loss: 1.6887 - sparse_categorical_accuracy: 0.7422 - val_loss: 1.1684 - val_sparse_categorical_accuracy: 0.8848
Epoch 3/3
48/48 [=====>.....] - 35s 72ms/step - loss: 1.2830 - sparse_categorical_accuracy: 0.7863 - val_loss: 1.1071 - val_sparse_categorical_accuracy: 0.8108
[68]: <tf.keras.src.callbacks.History at 0x7dae2410c820>

```

Figure 15: Top 5 artists approach 1 success

When trying the approach on the full dataset, we decided to experiment with Kaggle accelerators as well thus we ran the model on both GPU T4X2 and GPU P100. It took around 3 hrs and a half for GPU T4X2 to successfully train the model vs 1 hr 51 mins for GPU P100 to do the same training which means that also GPUs have an impact on model's execution time.

5.3.6 Approach 2 Training and Evaluation

We figured out the pipeline is working on the top 5 artists and with good results.

```
TRAINING
Epoch 1/3
Step 100/754, Loss: 2.9951, Accuracy: 0.6227
Step 200/754, Loss: 2.0151, Accuracy: 0.7241
Step 300/754, Loss: 1.6826, Accuracy: 0.7564
Step 400/754, Loss: 1.5234, Accuracy: 0.7707
Step 500/754, Loss: 1.4276, Accuracy: 0.7794
Step 600/754, Loss: 1.3637, Accuracy: 0.7860
Step 700/754, Loss: 1.3252, Accuracy: 0.7892
EVALUATION
Validation Loss: 0.9789, Validation Accuracy: 0.8265
TRAINING
Epoch 2/3
Step 100/754, Loss: 0.9598, Accuracy: 0.8302
Step 200/754, Loss: 0.9571, Accuracy: 0.8305
Step 300/754, Loss: 0.9486, Accuracy: 0.8316
Step 400/754, Loss: 0.9558, Accuracy: 0.8298
Step 500/754, Loss: 0.9559, Accuracy: 0.8300
Step 600/754, Loss: 0.9609, Accuracy: 0.8292
Step 700/754, Loss: 0.9592, Accuracy: 0.8291
EVALUATION
Validation Loss: 0.9464, Validation Accuracy: 0.8313
TRAINING
Epoch 3/3
Step 100/754, Loss: 0.9786, Accuracy: 0.8276
...
Step 600/754, Loss: 0.9045, Accuracy: 0.8372
Step 700/754, Loss: 0.9134, Accuracy: 0.8352
EVALUATION
Validation Loss: 0.9322, Validation Accuracy: 0.8326
```

Figure 18: Top 5 artists approach 2 success

When trying the approach on the full dataset it remained around an 3 hours in the first 10,000 rows of the first epoch thus it needs around 30-40+ hours to finish.

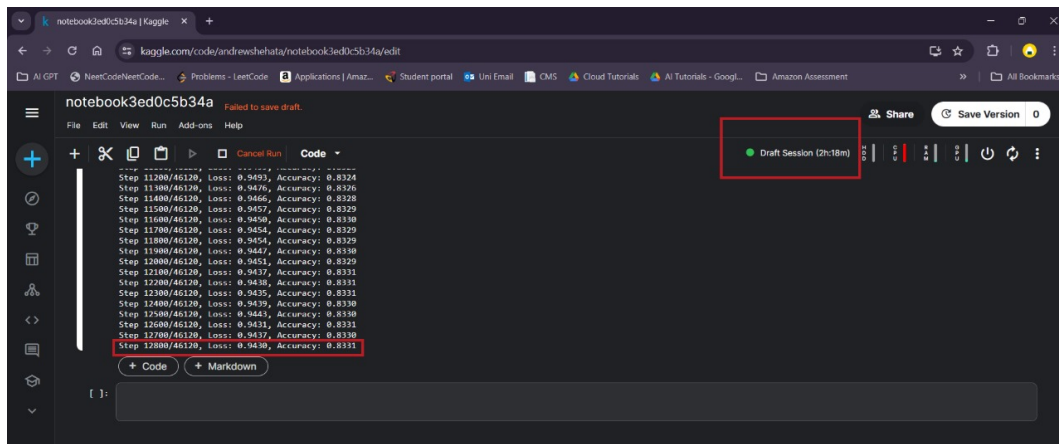


Figure 19: Full Dataset approach 2 fail

5.3.7 Generation

We select a specific lyric snippet and feed it to the tokenizer. The tokenizer transforms this text into a sequence of numbers that the model can understand. It then extracts the specific part representing the numerical token sequence. We use the model to create a continuation of the lyrics based on the provided input. We specify certain attributes such as:

- `max_new_tokens=40`: This limits the generated text to a maximum of 40 words.
- `do_sample=True`: This injects some randomness into the generation process, potentially leading to more diverse outputs.
- `top_k=30`: This restricts the options for the next word by considering only the top 30 most probable tokens at each step.
- `top_p=0.95`: This controls the model's exploration-exploitation trade-off. It prioritizes high-probability tokens while still allowing for some exploration of less probable but potentially interesting continuations.

The model generates one or more possible continuations. We focus on the first generated sequence and use the tokenizer again to convert it back into human-readable text by decoding it. The `skip_special_tokens=True` argument ensures we exclude any special tokens the model might have added during generation (like start or end markers).

6 Findings and Conclusion

1. Our proposed model architecture:

- **Advantages:**
 - (a) Easier to understand and analyze how the model processes sequences due to its simpler structure.
 - (b) Having the full control over the model architecture and can experiment with hyperparameters for optimization.
 - (c) Training a custom GRU model might require less computational power compared to pre-trained Transformers.
- **Limitations:**
 - (a) May not achieve the same level of performance as pre-trained Transformers in complex tasks like lyric generation.

2. `model.fit()` using TFT5 conditional transformer:

- **Advantages:**
 - (a) Leverages the pre-trained knowledge of TFT5, potentially leading to better lyric generation quality.
- **Limitations:**
 - (a) Understanding how a pre-trained Transformer arrives at its predictions can be challenging.
 - (b) Have less control over the overall model architecture as you're primarily fine-tuning an existing model.
 - (c) Training even with fine-tuning might require more computational resources compared to a custom GRU model.

3. Manual training loop using TFT5 conditional transformer:

- **Advantages:**
 - (a) Provides the most granular control over the training process by defining custom training steps, metrics, and loss functions tailored to your specific needs.

- (b) By carefully crafting the training loop and hyperparameter tuning, one can achieve even better performance compared to the simpler `model.fit` approach.

- **Limitations:**

- (a) Requires a deeper understanding of TensorFlow and Transformers to implement effectively. Debugging and troubleshooting custom training loops is challenging.
- (b) Defining and optimizing custom training loops takes more time and effort compared to using the built-in `model.fit` function.

Therefore as a conclusion, if interpretability and customization are the top priorities, a manual GRU model might be suitable. However, it would lead to a potentially low performance and longer training times. If faster training and potentially higher performance are the main goals, using `model.fit` with a pre-trained TFT5 model would be the best option as this offers a good balance between ease of use and performance. Finally, If someone has the expertise and require maximum control and flexibility, defining custom training functions and a training loop for TFT5 provides the most control but also demands the most effort and expertise.

Project Limitations

During the course of this project, we encountered several limitations, which were both hardware-related and dataset-related:

1. **Dataset Size:** The dataset comprised 56,000 entries, making it substantial in size. Training such a large dataset presented challenges, particularly when using smaller batch sizes to enhance the semantic understanding and depth of the model. Properly training the dataset under these conditions significantly increased the required training time.

2. **Hardware Constraints:** Our hardware setup, primarily personal laptops, encountered severe limitations. Specifically, we experienced crashes due to insufficient **RAM** and **CPU cores**, which impeded our ability to execute the `model.fit()` function and continue with the training process.

3. **Model Accuracy:** In our custom model architecture, we observed a very low accuracy percentage. This was largely attributed to the dataset limitations. Training a model of this data magnitude and not using a pre-trained model could extend the training duration to 30-40 hours or even longer.