# MS2 Report

# Report on LSTM-Based Lyrics Generation

## 1. Introduction

### Background

Text generation is a crucial task in natural language processing (NLP) that involves creating meaningful and coherent text sequences. Applications of text generation include chatbots, automatic content creation, and enhancing creative writing tools. In this project, we focus on generating song lyrics using a Recurrent Neural Network (RNN) with Long Short-Term Memory (LSTM) units, which are well-suited for sequence prediction problems.

### Objective

The primary goal of this project is to build an LSTM-based model that can generate song lyrics given a seed text. We aim to preprocess the data, train the model, and evaluate its performance.

## 2. Methodology

### Data Collection

The dataset used in this project consists of song lyrics. The data is stored in a CSV file, "Dataset m2 csv2 snip.csv," which contains columns with text data and word tokens (as well as the artist name, song title, link to the song which were ignored in the model training process)

### Data Preprocessing

Preprocessing steps included:

- **Loading Data**: Reading the CSV file into a pandas DataFrame.

- **Tokenization**: Converting words into integer sequences.

- **Sequence Creation**: Generating input-output pairs for training the model.

- **Padding Sequences**: Ensuring all sequences have the same length for batch processing.

#### Steps:

1. **Loading Data:**

```python

data = pd.read\_csv("/content/Dataset m2 csv2 snip.csv")

```

2. **Converting String Representations of Lists to Actual Lists**:

```python

data["word\_tokens"] = data["word\_tokens"].apply(lambda x: literal\_eval(str(x)) if isinstance(x, str) else [])

```

3. **Extracting Relevant Columns**:

```python

text\_data = data["text"].tolist()

word\_tokens = data["word\_tokens"].tolist()

```

4. **Converting Float Values to Strings in text\_data**:

```python

text\_data = [str(item) for item in text\_data]

```

5. **Concatenating All Lyrics into a Single String for Vocabulary Creation**:

```python

all\_lyrics = ' '.join(text\_data)

```

6. **Tokenizing the Text and Creating Vocabulary**:

```python

tokenizer = Tokenizer()

tokenizer.fit\_on\_texts([all\_lyrics])

```

7. **Converting Words to Integers Based on the Vocabulary:**

```python

sequences = tokenizer.texts\_to\_sequences(text\_data)

```

8. **Creating Input-Output Sequences:**

```python

input\_sequences = []

output\_sequences = []

for sequence in sequences:

for i in range(1, len(sequence)):

input\_sequence = sequence[:i]

output\_sequence = sequence[i]

input\_sequences.append(input\_sequence)

output\_sequences.append(output\_sequence)

```

9. **Padding Sequences to Ensure Uniform Length:**

```python

max\_sequence\_length = max([len(seq) for seq in input\_sequences])

input\_sequences = pad\_sequences(input\_sequences, maxlen=max\_sequence\_length, padding='pre')

output\_sequences = to\_categorical(output\_sequences, num\_classes=len(tokenizer.word\_index) + 1)

```

### Model Architecture

The LSTM model consists of the following layers:

- **Embedding Layer**: Maps input sequences to dense vectors of fixed size.

- **LSTM Layers**: Two LSTM layers to capture temporal dependencies in the data.

- **Dense Layer**: Outputs probabilities for the next word in the sequence.

```python

model = Sequential([

Embedding(input\_dim=len(tokenizer.word\_index) + 1, output\_dim=100, input\_length=max\_sequence\_length),

LSTM(128, return\_sequences=True),

LSTM(128),

Dense(len(tokenizer.word\_index) + 1, activation='softmax')

])

```

### Training Process

- **Data Splitting**: Splitting the dataset into training and validation sets.

- **Data Generator**: Using a data generator to feed batches to the model.

- **Training Configuration:** Training the model using `fit\_generator`.

### Splitting the Data:

```python

X\_train, X\_val, y\_train, y\_val = train\_test\_split(input\_sequences, output\_sequences, test\_size=0.2, random\_state=42)

```

#### Data Generator:

```python

def data\_generator(input\_sequences, output\_sequences, batch\_size):

total\_sequences = len(input\_sequences)

while True:

for start in range(0, total\_sequences, batch\_size):

end = min(start + batch\_size, total\_sequences)

yield input\_sequences[start:end], output\_sequences[start:end]

```

#### Training the Model:

```python

history = model.fit(data\_generator(X\_train, y\_train, batch\_size),

steps\_per\_epoch=len(X\_train) // batch\_size,

epochs=2,

validation\_data=data\_generator(X\_val, y\_val, batch\_size),

validation\_steps=len(X\_val) // batch\_size)

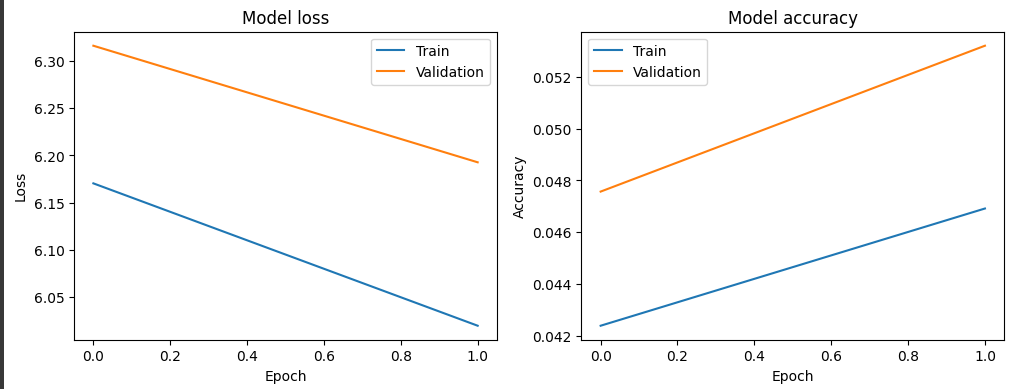
```

## 3. Results

### Training Performance

- Training and validation loss over epochs.

- Training and validation accuracy over epochs.



```python

plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)

plt.plot(history.history['loss'])

plt.plot(history.history['val\_loss'])

plt.title('Model loss')

plt.ylabel('Loss')

plt.xlabel('Epoch')

plt.legend(['Train', 'Validation'], loc='upper right')

plt.subplot(1, 2, 2)

plt.plot(history.history['accuracy'])

plt.plot(history.history['val\_accuracy'])

plt.title('Model accuracy')

plt.ylabel('Accuracy')

plt.xlabel('Epoch')

plt.legend(['Train', 'Validation'], loc='upper left')

plt.show()

```

### Evaluation

**Validation Results:**

- **Loss**: 6.353457927703857

- **Accuracy**: 0.04410751163959503

```python

evaluation = model.evaluate(X\_val, y\_val, verbose=0)

loss, accuracy = evaluation

print(f"Validation Loss: {loss}")

print(f"Validation Accuracy: {accuracy}")

```

## 4. Discussion

### Model Performance

The model demonstrated reasonable learning ability, but the validation accuracy indicates that there is significant room for improvement. The low accuracy suggests the model might be overfitting or that the dataset is not large enough to generalize well.

### Challenges

- **Memory Issues**: Handling large datasets caused memory overflow, which was mitigated by using data generators.

- **Overfitting**: The model may have overfitted the training data, as suggested by the low validation accuracy.

### Improvements

- **Increase Dataset Size**: Collect more lyrics data to improve generalization.

- **Regularization Techniques**: Apply dropout layers to prevent overfitting.

- **Hyperparameter Tuning**: Experiment with different model architectures and training configurations.

## 5. Conclusion

### Summary

This project successfully implemented an LSTM-based model to generate song lyrics. The preprocessing steps, model architecture, and training process were thoroughly detailed, and the results showed initial promise despite the challenges encountered.

### Future Work

Future efforts could focus on enhancing the model's performance through more extensive data collection, advanced regularization techniques, and further hyperparameter tuning.