Text-Summarization-Amazon-Fine-Food-Reviews

Project Overview

In this project, we are focusing on text summarization. The input is a long review of Amazon Fine Food products, and the output is a concise summary of the review.

Data-Visualization

```
Data columns (total 10 columns):
 #   Column                  Non-Null Count  Dtype 
---  ------                  --------------  ----- 
 0   Id                      42023 non-null  int64 
 1   ProductId               42023 non-null  object
 2   UserId                  42023 non-null  object
 3   ProfileName             42023 non-null  object
 4   HelpfulnessNumerator    42023 non-null  int64 
 5   HelpfulnessDenominator  42023 non-null  int64 
 6   Score                   42023 non-null  int64 
 7   Time                    42023 non-null  int64 
 8   Summary                 42023 non-null  object
 9   Text                    42023 non-null  object
dtypes: int64(5), object(5)
```

Id	ProductId	UserId	ProfileName	HelpfulnessNumerator	HelpfulnessDenominator	Score	Time	Summary	Text
19747	B0030VBRIU	ACU1MBQ7T5YL0	Cynthia C. Eberhardt "CCEber"	3	3	5	1294704000	Yummy protein	I usually buy Sprout and thought I'd try this ...
43860	B001EQ5JLE	A1BUMMBK9WA993	Joyeuse	3	3	4	1290297600	Joyeuse	When we were in London three years ago, I love...

1. We observe that each row contains a text and corresponding summary. The remaining columns are of no use in this task so we just ignore them.
2. As part of basic preprocessing, we drop any rows that are duplicates or contain null values.

Pre-Processing

As part of the preprocessing steps, I have performed the following transformations:

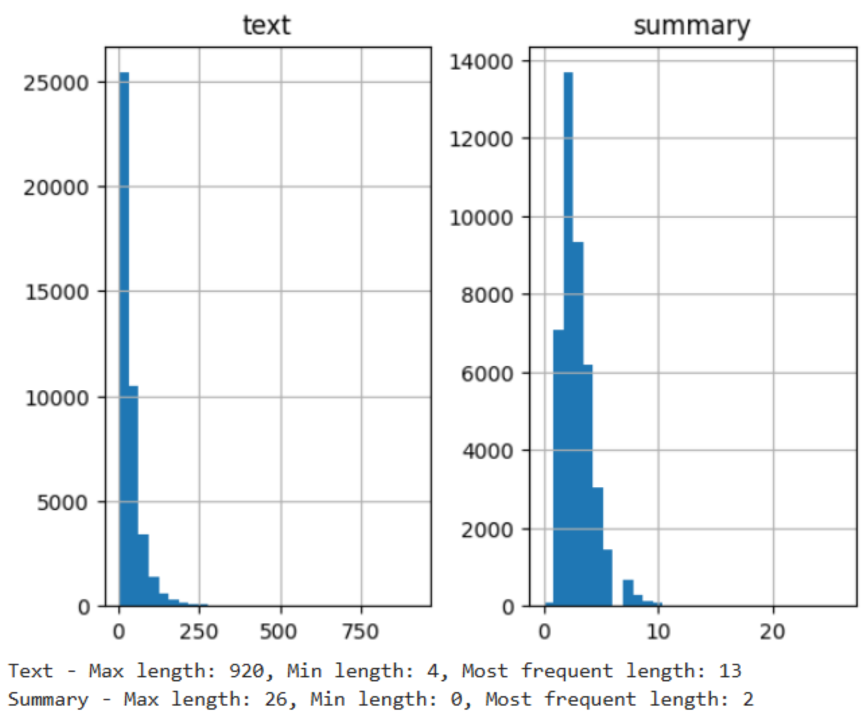
1. Converted all text to lowercase.
2. Replaced emojis with their meanings.
3. Some pre-encoded emojis that are not demojized back to their meanings will be simply removed.
4. Expanded contractions (e.g., `'ve` to `have`).
5. Removed HTML tags.
6. Abbreviated common terms (e.g., `GM` for `Good Morning`).
7. Removed stop words.

- **Why not perform stemming?** We need to show the output, and the root form may or may not be a proper English word.
- **Why not perform lemmatization?** While lemmatization converts the root form back to an English word, it would take a lot of computation time.
- **Why not perform spelling correction?** Simply because the data is too large, and it would consume a significant amount of time.

Example – bought several vitality canned dog food products found good quality product looks like stew processed meat smells better labrador finicky appreciates product better : `good quality dog food`

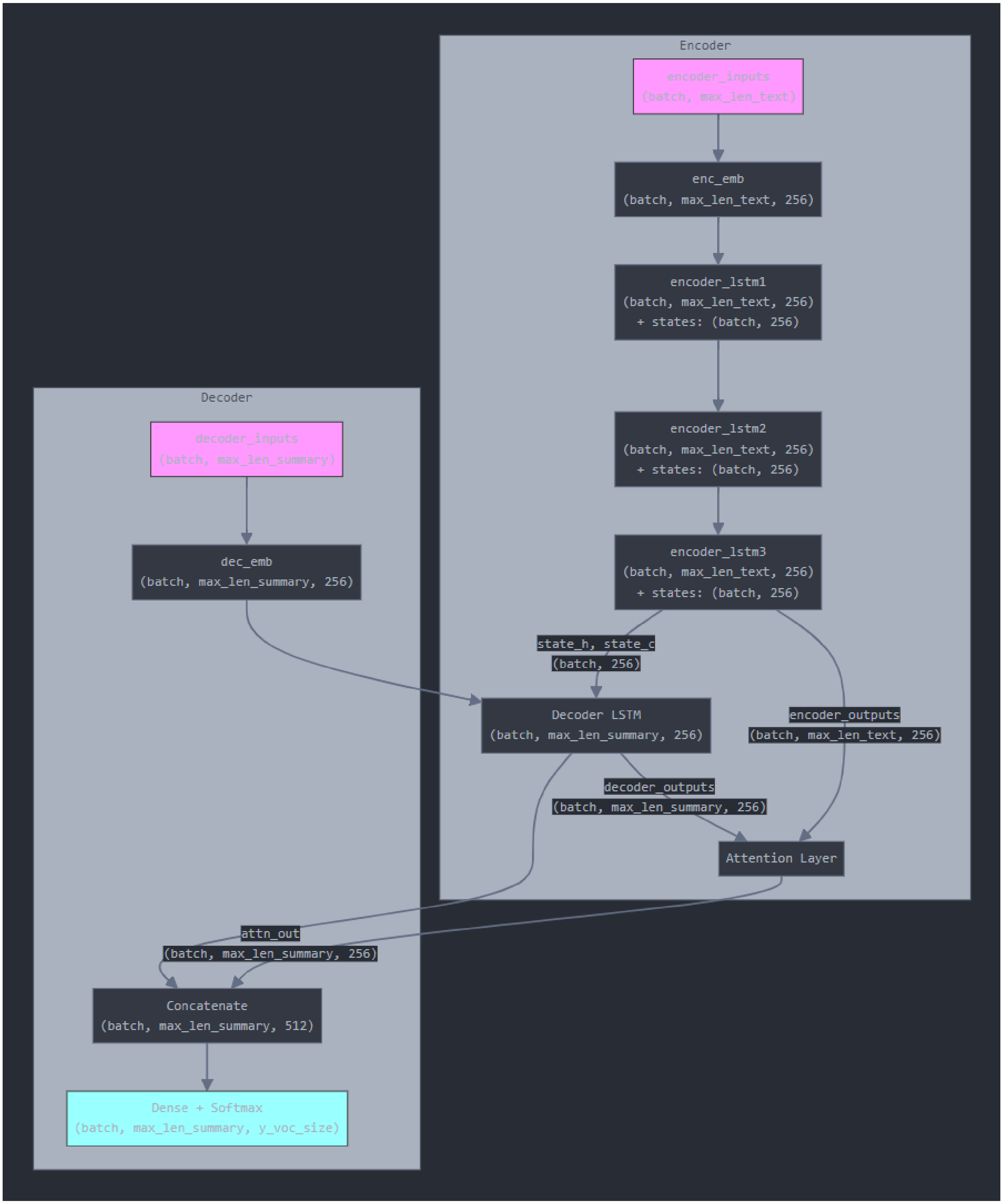
Feature-Engineering

1. Now, as the input length of the model can vary, but all inputs in a given batch must be of the same length, choosing an appropriate `max_length` based on the typical length distribution of your dataset helps retain the most relevant information.



2. We try to pad smaller sentences and truncate the longer ones. Hence, I choose an optimal `max_length` of 80 for text and 7 for summary.
3. Use the BERT tokenizer (uncased) to convert sentence tokens to their corresponding IDs. As for the vocabulary, it will be the BERT tokenizer's. Why do I choose this? Because the data is large and different people provided reviews, so much of it would be covered instead of creating a new vocabulary.

Model-Architecture



1. First both the inputs for encoder and decoder go through a embedding layer where each token is convert to a embedding of dimension 256.
2. Then encoder_inputs go through 3 LSTM encoder layers (The hidden states at each time step are passed onto next layer)
3. Remember the Hidden and cell states are of same dimension as the embeddings
4. The final encoder LSTM layer produces encoder_outputs (hidden states at each timestep) and final hidden/cell states (state_h, state_c)
5. The final states (state_h, state_c) are used to initialize the decoder LSTM, transferring the context of the input sequence
6. The decoder processes its embedded inputs while starting with encoder's final states
7. Cross-attention mechanism then:
    - Takes encoder_outputs (all hidden states from encoder) as Keys/Values
    - Takes decoder_outputs (current decoder hidden states) as Queries
    - Computes attention weights for each decoder timestep over encoder outputs
    - Produces context vectors focusing on relevant parts of input
8. The context vectors are concatenated with decoder outputs (combining attended input info with current decoder state)
9. Finally, the concatenated vectors go through a dense layer with softmax to predict the next token probabilities

Training Parameters:

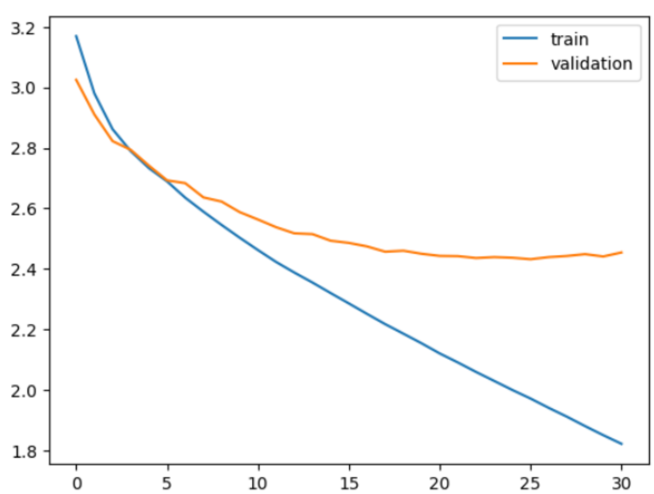
→ The optimizer used here is `RMSprop`.

→ The loss function is `sparse_categorical_crossentropy`.

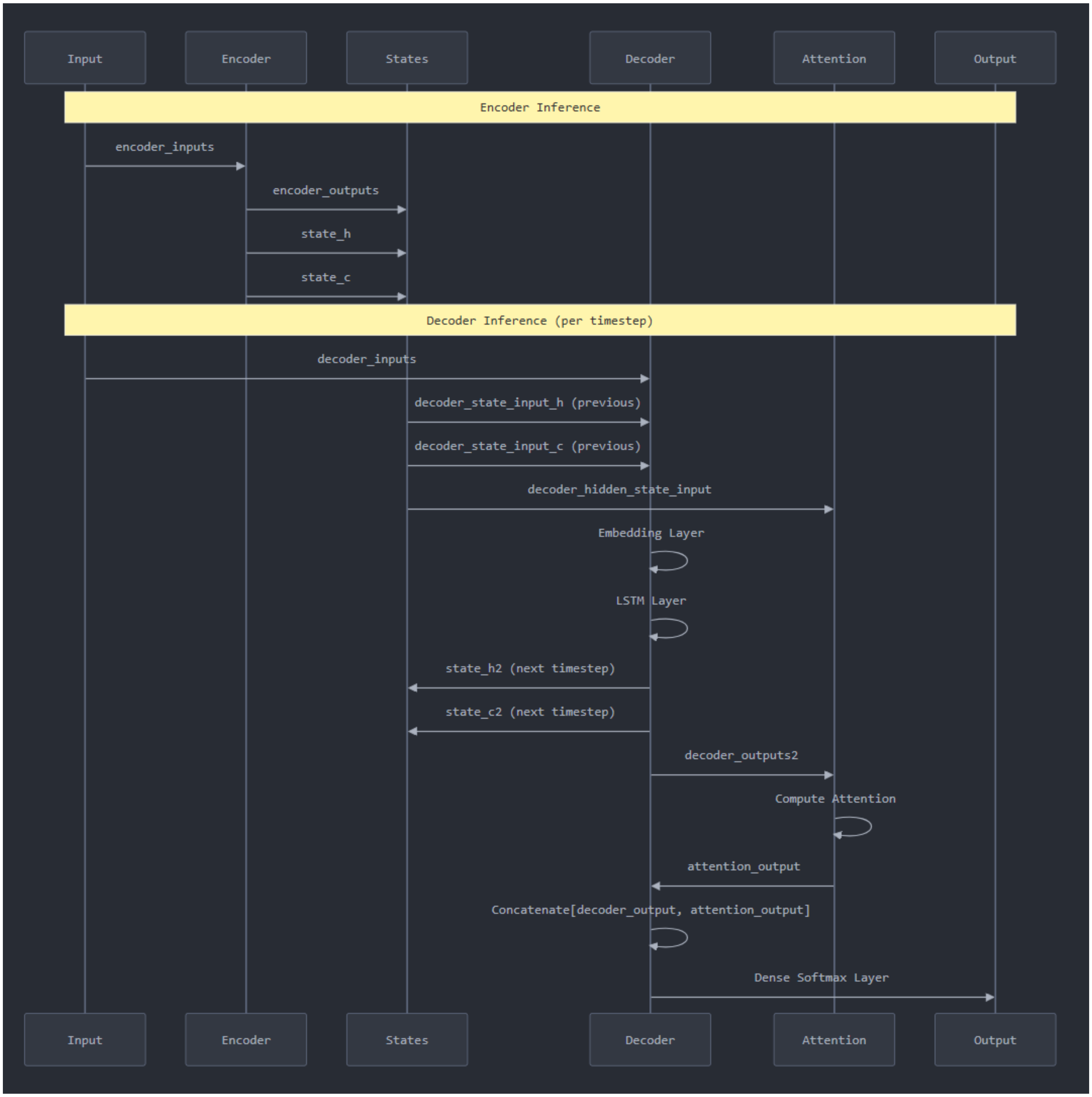
→ The metric used is accuracy.

→ A scheduler was not used.

→ Early stopping was implemented with a patience of 5 epochs if the model doesn't learn anything new after a while.



Inference Time



Encoder Inference

- **Inputs:** `encoder_inputs`
- **Process:** The `encoder_inputs` are passed through embedding and LSTM layers.
- **Outputs:**
  - `encoder_outputs`: The outputs from the LSTM layer for all timesteps.

- `state_h`: The hidden state from the LSTM layer.
- `state_c`: The cell state from the LSTM layer.

Decoder Inference (per timestep)

- **Inputs:**
  - `decoder_inputs`: The current input token for the decoder.
  - `decoder_state_input_h`: The hidden state from the previous timestep.
  - `decoder_state_input_c`: The cell state from the previous timestep.
  - `decoder_hidden_state_input`: The full `encoder_outputs` used for attention mechanism.

Processing Steps for Decoder Inference

1. **Embedding:** Embeds the `decoder_inputs` using an embedding layer.
2. **LSTM Processing:** The embedded input is passed through an LSTM layer along with the `decoder_state_input_h` and `decoder_state_input_c` (previous hidden and cell states).
3. **State Update:** The LSTM layer generates new hidden and cell states for the next timestep.
4. **Attention Computation:** Computes attention scores over the `encoder_outputs` to focus on relevant parts of the encoder's output.
5. **Concatenation:** Concatenates the LSTM output with the attention output to form a combined context vector.
6. **Probability Distribution:** The combined vector is processed to generate a probability distribution over the vocabulary, indicating the likelihood of each word being the next output.

Function Descriptions

- `decode_sequence(input_seq)`:
  - **Purpose:** Generates summary text from the encoded input sequence.
  - **Process:**
    * Encodes the input using the encoder model.
    * Initializes with the `'start'` token.
    * Iteratively predicts the next tokens until:
      · The `'end'` token is reached, or
      · The maximum summary length is reached.
    * Updates states and the target sequence at each iteration.
- `seq2summary(input_seq)`:
  - **Purpose:** Converts the target sequence into a readable summary.
  - **Process:**
    * Filters out padding (`0`) and special tokens (`'start'`, `'end'`).
    * Converts indices to words using the `reverse_target_word_index`.
- `seq2text(input_seq)`:
  - **Purpose:** Converts the source sequence into readable text.
  - **Process:**
    * Filters out padding (`0`).
    * Converts indices to words using the `reverse_source_word_index`.

Results

```
Review: quick easy portable think tastes great favorite flavor
Original summary: delicious
Predicted summary: good
```

```
Review: like coffee says go said
Original summary: like coffee
Predicted summary: good coffee
```

```
Review: sampled bought fair share ever ##idae sa
Original summary: stuff put everything sauce
Predicted summary: favorite sauce ##s
```

```
Review: discovered ta ##jin san diego al
Original summary: great spice
Predicted summary: great flavor
```

```
Review: search healthy snacks found item org
Original summary: top notch snack
Predicted summary: one best
```