NLP Final project

Subha Chakraborty MT2024156 Harshal Gujrathi MT2024055
Omkar Satav MT2024106
Akash Gorakh Chaudhari MT2024012

April 2025

1 Introduction

Our NLP project focuses on the task of **abstractive summarization** using the `nlplabtdtu/xlsum_en` dataset. We employ a transformer-based architecture to generate concise and coherent summaries from news articles.

1.1 Abstractive vs. Extractive Summarization

The summarization task can be broadly categorized into two types:

- **Extractive Summarization**: This approach involves selecting and concatenating key sentences or phrases directly from the source text, without altering the original wording.
- **Abstractive Summarization**: In contrast, this approach aims to generate a summary that may include novel phrases or content that is not present in the source text, closely mimicking how a human would summarize.

We are focused on abstractive summarization in our project.

2 Exploratory Data Analysis

This section of the report focuses on understanding the dataset through **Exploratory Data Analysis (EDA)**. We present the methods and techniques used to explore

the dataset, along with the key insights derived. These insights informed several important decisions in the development and implementation of our summarization model.

2.1 Article length and Summary length

The `nlplabtdtuxlsum_en` dataset, after concatenating the train, test, and validation splits, contains a total of 329,591 instances. Each instance comprises three columns: `title`, `text`, and `target`. The train set consists of 306,521 instances, while the test and validation sets contain 11,535 instances each.

To gain an initial understanding of the dataset, we analyzed the distribution of article and summary lengths.

The figure below illustrates the scatter plot between article and summary lengths.

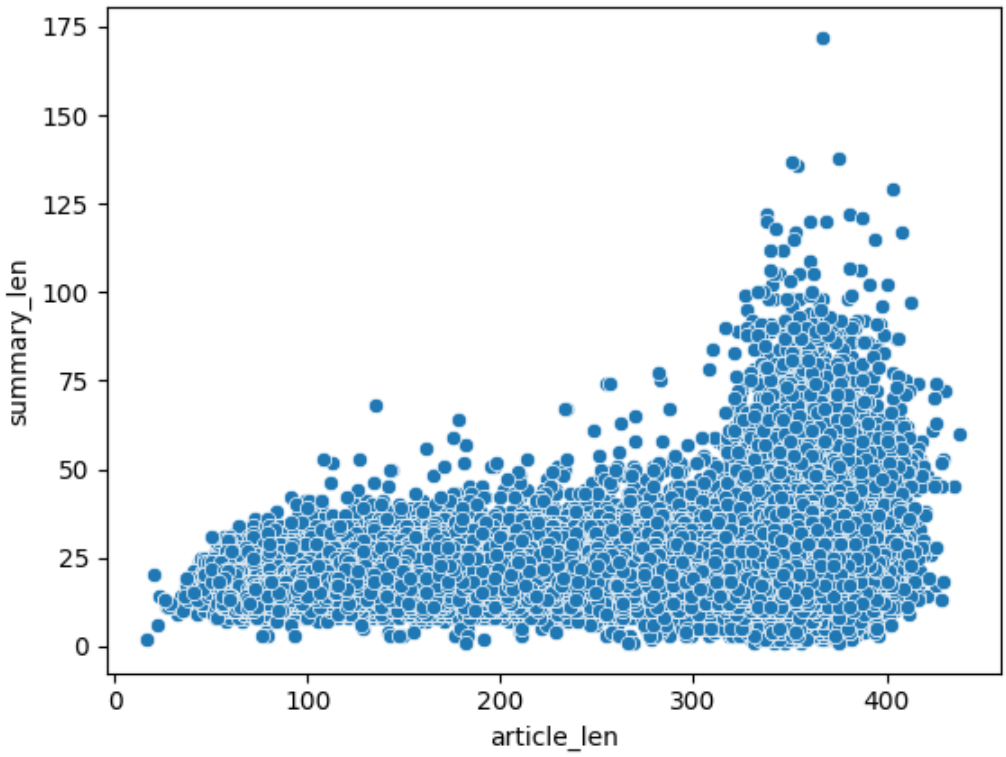

Figure 1: Scatter Plot of summary length vs article length

The plots below illustrate the frequency distribution of word counts in the articles

and their corresponding summaries.

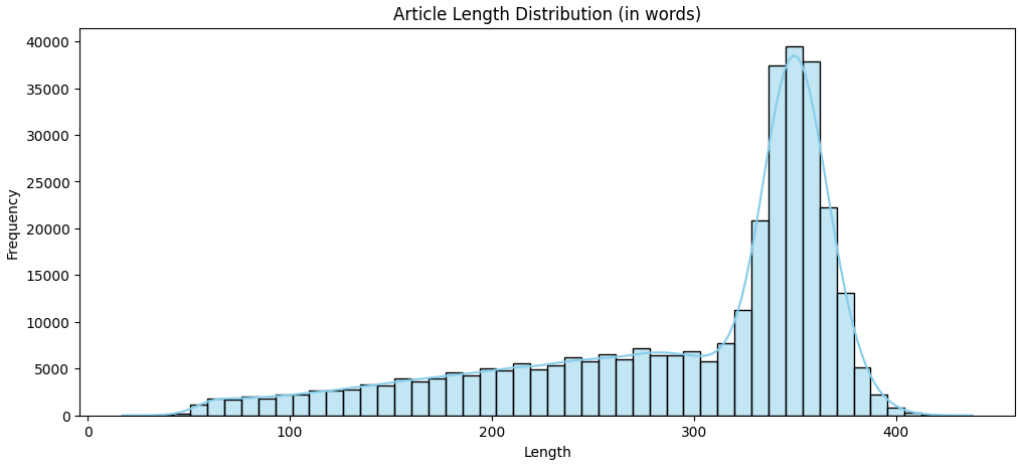

Figure 2: Plot of article length vs frequency of articles

As seen in the plot above, the distribution of article lengths is slightly skewed to the left, with a sharp peak around 350 words. The majority of articles fall within the 320–370 word range, with the maximum article length observed being 438 words. This suggests that a transformer model with a 512-token input limit can accommodate nearly all articles without the need for truncation.

Below is a plot of the summary lengths and the frequency of articles of the said lengths.

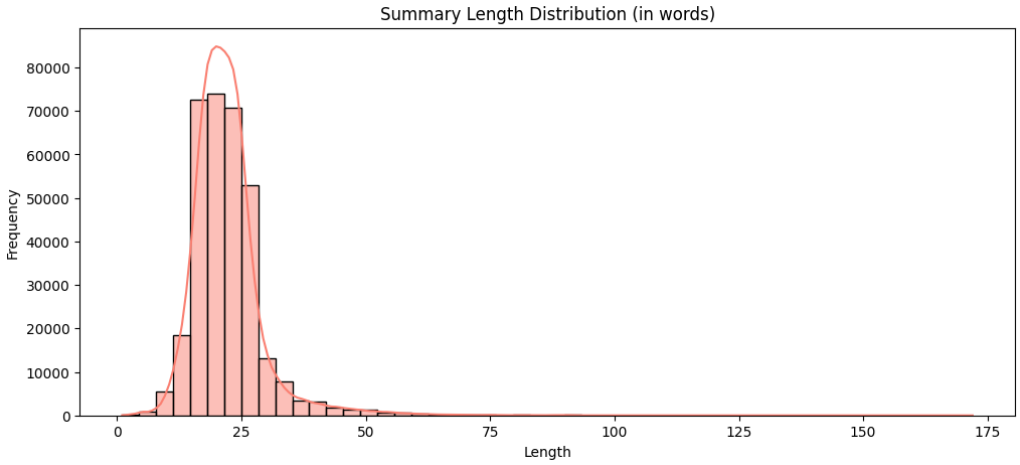

Figure 3: Plot of summary length vs frequency of articles

The plot above shows that most summaries range between 15 and 30 words. The longest summary in the dataset contains 172 words, which still fits comfortably within the typical 512-token limit used in transformer-based architectures.

These observations indicate that both the input (article) and output (summary) sequences fall within a manageable range for transformer models, without requiring excessive truncation or padding.

2.2 Compression Ratio

The compression ratio is defined as follows:

$$\text{Compression Ratio} = \frac{\text{Summary Length}}{\text{Article Length}}$$

This metric quantifies the extent to which the original article is reduced in length during summarization. A lower compression ratio indicates more aggressive compression, often meaning that abstractive summarization is required, as extractive methods may struggle to achieve such high levels of condensation while preserving meaning.

The following plot shows the distribution of compression ratios across the dataset:

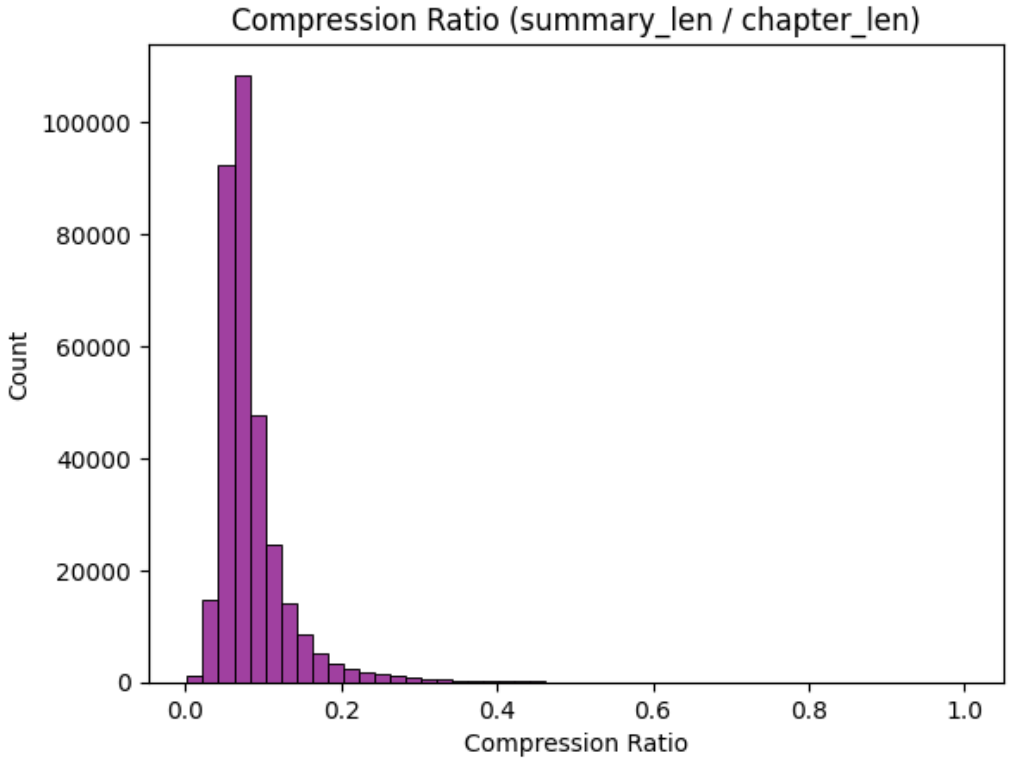

Figure 4: Plot of compression ratio vs frequency of articles

As observed from the plot, the compression ratios peak between 0.1 - 0.12 , which means that most summaries are 10- 12 % of their original article length, which is typical for headline style generation, as we will further showcase in our project results.

Insights from Word Frequency Analysis of Summaries

To get a better idea of what the summaries in the XLSum dataset mostly talk about, we did a word frequency analysis. We combined all the words from the `target` column (which contains the summaries), removed common stopwords (like "the", "is", "and") using NLTK's stopword list, and then counted how often each remaining word appeared.

From this, we found that some of the most common words were *"new"*, *"people"*, *"man"*, *"government"*, *"police"*, *"first"*, *"US"*, and *"UK"*. These words suggest that many summaries are about news topics like current events, politics, public safety, and international issues. Words like *"said"* and *"says"* also appeared a lot, which

makes sense since summaries often include quotes or reported statements.

Overall, this analysis gives a quick idea of what topics are common in the summaries and what kind of vocabulary the model will learn to generate during training.

3 Model Design, Implementation and Experimentation

In this section, we explain how we built and implemented our summarization model, how we tokenized the data, and the transformer-based architecture we chose. We also cover the training setup, evaluation methods, and how we generated summaries from the model. All of these steps were important to make sure the model performs well on the abstractive summarization task.

3.1 Tokenizer

We explored two different approaches for tokenization in our project: using a pre-trained tokenizer and training a custom tokenizer from scratch. Both approaches were used to convert raw text and summaries into a format suitable for training our transformer model.

3.1.1 Approach 1: Pre-trained BART Tokenizer

In this approach, we used the `facebook/bart-base` tokenizer from the HuggingFace Transformers library. This tokenizer is trained on a large corpus and uses a subword-based vocabulary that works well with transformer models like BART.

We applied tokenization to both the input articles and target summaries. Each input was truncated or padded to fixed lengths: 512 tokens for the source (article) and 128 tokens for the target (summary), as per our hyperparameter setup. The processed data included input IDs, attention masks, and labels. Padding was handled using the tokenizer's `pad_token_id`. After tokenization, we used PyTorch's `DataLoader` to create batches of size 8 for training and validation. This approach gave us a quick and reliable way to prepare the data for the model.

3.1.2 Approach 2: Custom Byte-Level BPE Tokenizer

In the second approach, we trained a custom tokenizer from scratch using the Byte-Level BPE (Byte Pair Encoding) algorithm from the `tokenizers` library. For training the tokenizer, we combined the article and summary texts from all three splits

(train, validation, and test) to form the training corpus. The tokenizer was trained with a minimum token frequency of 2 and a vocabulary size of 30,000. Special tokens such as `<s>`, `<pad>`, `</s>`, `<unk>`, and `<mask>` were also added.

Once trained, we saved the vocabulary and merges files and loaded them using the `BartTokenizerFast` class from HuggingFace. This allowed us to use the custom tokenizer with our transformer model, while maintaining compatibility with the BART architecture.

3.1.3 Tokenization Summary:

- **Max Source Length (article)**: 512 tokens
- **Max Target Length (summary)**: 128 tokens
- **Batch Size**: 8
- **Vocabulary Size (custom tokenizer)**: 30,000
- **Vocabulary Size (pretrained tokenizer)**: 50,265

Both tokenizers helped us transform the raw text into token IDs and attention masks, which are the required inputs for training transformer models. While the pre-trained tokenizer offered convenience and good performance out of the box, the custom tokenizer gave us more control and adaptability for our specific dataset.

3.2 Trainable Positional Encoding

Transformers do not have any built-in sense of the order of tokens in a sequence. To help the model understand the position of each word, we add *positional encodings* to the token embeddings before feeding them into the transformer layers.

In standard transformer architectures, **sinusoidal positional encodings** are often used. These are fixed and non-trainable, based on sine and cosine functions of different frequencies. While they work well in practice, they are not flexible and do not adapt to the data during training.

In our project, we chose to use a **trainable positional encoding** instead. We implemented it as a custom PyTorch module. The idea is to learn positional representations during training that better match the structure of our dataset.

Below is the class we used:

```python
class TrainablePositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=1000):
        super().__init__()
        self.pos = nn.Parameter(torch.zeros(max_len, d_model))
    def forward(self, x):
        seq_len = x.size(0)
        return x + self.pos[:seq_len].unsqueeze(1)
```

Figure 5: Trainable Positional encoding class used

Here, `d_model` is the embedding dimension, and `max_len` is the maximum sequence length we expect. The `pos` parameter is a learnable matrix initialized to zeros and updated during training. In the forward pass, the position embeddings are added to the token embeddings.

Basically using trainable positional encodings gives the model more flexibility and can improve its ability to understand sequential data like our dataset.

After tokenization, we set up the PyTorch `DataLoader` to prepare the data for training and validation. This step ensures that the data is efficiently loaded in batches. We also enabled features like shuffling, pinned memory, and persistent workers to make training faster and more efficient:

```python
def get_loaders(dataset_path):
    splits = load_dataset(dataset_path, split={"train":"train","validation":"validation"})
    tokenized = splits.map(tokenize, batched=True)
    tokenized.set_format(type='torch', columns=['input_ids','attention_mask','labels'])
    train_loader = DataLoader(tokenized['train'], batch_size=batch_size, shuffle=True, num_workers=6, pin_memory=True,persistent_workers=True)
    valid_loader = DataLoader(tokenized['validation'], batch_size=batch_size, num_workers=5, pin_memory=True)

    return train_loader, valid_loader
```

Figure 6: Dataloader

The `train_loader` and `valid_loader` return batches of three items for each example:

- **input_ids**: Tokenized IDs of the article text (the input to the encoder).

- **attention_mask**: A binary mask indicating which tokens are padding (0) and which are actual tokens (1), used by the model to ignore padded positions.
- **labels**: Tokenized IDs of the target summary (used as ground truth for loss calculation).

3.3 Transformer-Based Summarizer Architecture

The core model used for this task is a Transformer-based encoder-decoder architecture built using PyTorch. This model follows the original Transformer design proposed by Vaswani et al., which heavily relies on attention mechanisms, especially **self-attention** and **cross-attention**.

3.3.1 Model Structure

We define a class `TransformerSummarizer` which contains the following:

- **Embedding Layer:** Converts input tokens into dense vectors of size $d_{model} = 512$.
- **Trainable Positional Encodings:** Since Transformers do not have any built-in understanding of the order of tokens, we add positional information. Instead of using fixed sinusoidal encodings, we use learnable positional encodings which the model can optimize during training. These help the model capture word order more effectively, especially in tasks like summarization where position matters.
- **Transformer Encoder:** Each encoder layer uses **self-attention**, which allows each token in the input sequence to attend to all other tokens. This mechanism helps the model understand relationships and dependencies between words regardless of their position.
- **Transformer Decoder:** The decoder layers also use **self-attention** to understand relationships in the generated (target) sequence, and **cross-attention** to connect the generated summary to the original input. Cross-attention allows the decoder to focus on relevant parts of the input while generating each word in the summary.
- **Output Projection:** A linear layer maps the decoder output back to vocabulary logits for token prediction.

3.3.2 Differences Between Approaches

We experimented with two tokenizer approaches:

- **Approach 1:** Used the pre-trained `facebook/bart-base` tokenizer with a vocabulary size of 50265.
- **Approach 2:** Trained a custom Byte-Pair Encoding (BPE) tokenizer on the dataset with a smaller vocabulary size of 30000.

Both approaches use the same Transformer architecture but differ in how tokens are represented and embedded.

3.3.3 Parameters and i/o details

Input and Output Details: The `forward` method:

- Uses `pad_token_id` to create key padding masks so the model ignores padded tokens during attention.
- Applies a causal mask in the decoder to prevent future tokens from being seen during training.
- Returns the output logits, which are used to generate summaries during inference or compute loss during training.

The following hyperparameters were used throughout the training process:

- **Maximum Source Length:** 512 tokens – the maximum number of tokens for input texts.
- **Maximum Target Length:** 128 tokens – the maximum length for the generated summaries.
- **Batch Size:** 8 – number of samples per training batch.
- **Epochs:** 10 (for Custom Tokenizer), 25 (for pretrained Bart Tokenizer)
- **Learning Rate:** 0.0001 – the initial learning rate used with AdamW optimizer.
- **Weight Decay:** 0.01 – regularization term to prevent overfitting.

- **Gradient Clipping:** 1.0 – maximum gradient norm to stabilize training and prevent exploding gradients.
- **Beam Size:** 4 – number of beams used during beam search decoding.
- **Warmup Ratio:** 0.1 – proportion of total steps used for learning rate warm-up.
- **Checkpoint Directory:** `checkpoint/` – path where model checkpoints are saved.
- **Loss Function:** Cross Entropy Loss – used to measure the difference between predicted and actual tokens.
- **Number of Trainable Parameters for our XLSum Dataset:** - 74,616,921

3.4 Decoder Approaches

To generate summaries from the trained Transformer model, we implemented two decoding strategies: **Greedy Decoding** and **Beam Search Decoding**. These methods determine how tokens are selected during inference based on model output probabilities.

Greedy Decoding: In greedy decoding, the model selects the token with the highest probability at each step of the sequence generation. Starting from a beginning-of-sequence (BOS) token, the model iteratively appends the most probable next token until an end-of-sequence (EOS) token is generated or the maximum length is reached. This method is straightforward and fast, but it often leads to suboptimal results because it does not consider alternative candidates that might lead to better overall sequences later on.

Beam Search Decoding: To improve upon greedy decoding, we also implemented beam search. Beam search keeps track of the top-k (beam size) most promising sequences at each decoding step instead of committing to a single token choice. For every sequence in the current beam, the model considers all possible next tokens and computes the log-probabilities of the resulting sequences. The top k sequences based on cumulative log-probabilities are retained for the next step.

Beam search also incorporates mechanisms like:

- **Length Penalty:** Adjusts the final score of each sequence based on its length to prevent the model from favoring shorter sequences.
- **Early Stopping:** Halts the search process if all candidate beams have generated the EOS token before reaching the maximum sequence length.

This approach significantly increases the chances of generating more fluent and informative summaries compared to greedy decoding, albeit with a higher computational cost.

Implementation Summary: Both decoding methods utilize the trained Transformer model with the encoder's output and an auto-regressive decoder input. The decoder begins with the BOS token and iteratively generates tokens based on the model's predictions. In beam search, multiple beams are expanded and scored at each step, while in greedy decoding, only the highest probability path is followed. In both cases, the final generated sequence is decoded back into human-readable text using the tokenizer.

4 Conclusion and Observations

4.1 Approach 1: Using Pretrained `BartTokenizer`

This is the result of our first approach using the pre-trained Bart Tokenizer. Training was performed over 25 epochs, and both the training and validation loss steadily decreased, suggesting effective learning.

Loss Trend.

Epoch	Train Loss	Validation Loss
1	6.1565	4.8470
5	3.3396	3.2916
10	2.8303	3.0883
15	2.5269	3.0493
20	2.2777	3.0587
25	2.0708	3.0722

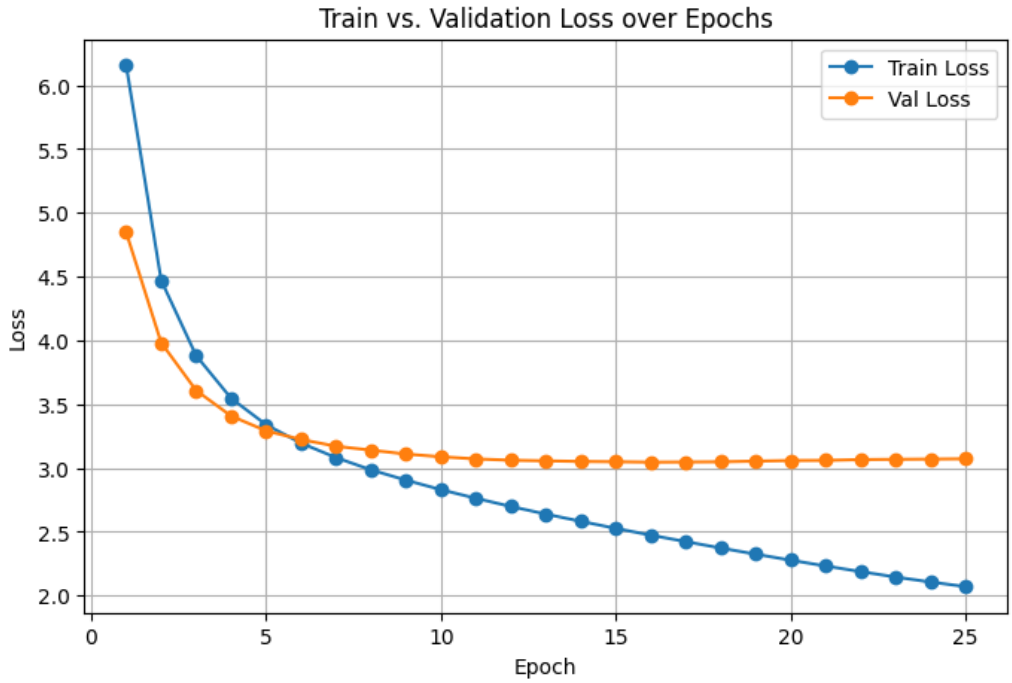

Figure 7: Train and Validation loss for approach 1(pretrained BartTokenizer)

ROUGE Scores.

Epoch	ROUGE-1	ROUGE-2	ROUGE-L
5	0.271596	0.0710721	0.211202
10	0.290982	0.0851596	0.226525
15	0.300244	0.0915641	0.234104
16	0.299387	0.0918897	0.232703
17	0.300989	0.0920619	0.234134
20	0.304150	0.0948534	0.236394
25	0.305513	0.0959938	0.237651

4.2 Approach 2: Custom Tokenizer

This is our second approach with out custom tokenizer . Though its ROUGE scores were modestly lower than the pretrained tokenizer approach, the performance gap was minimal.

Loss Trend.

Epoch	Train Loss	Validation Loss
1	5.6396	4.3961
5	3.2176	3.2373
10	2.6749	3.0764

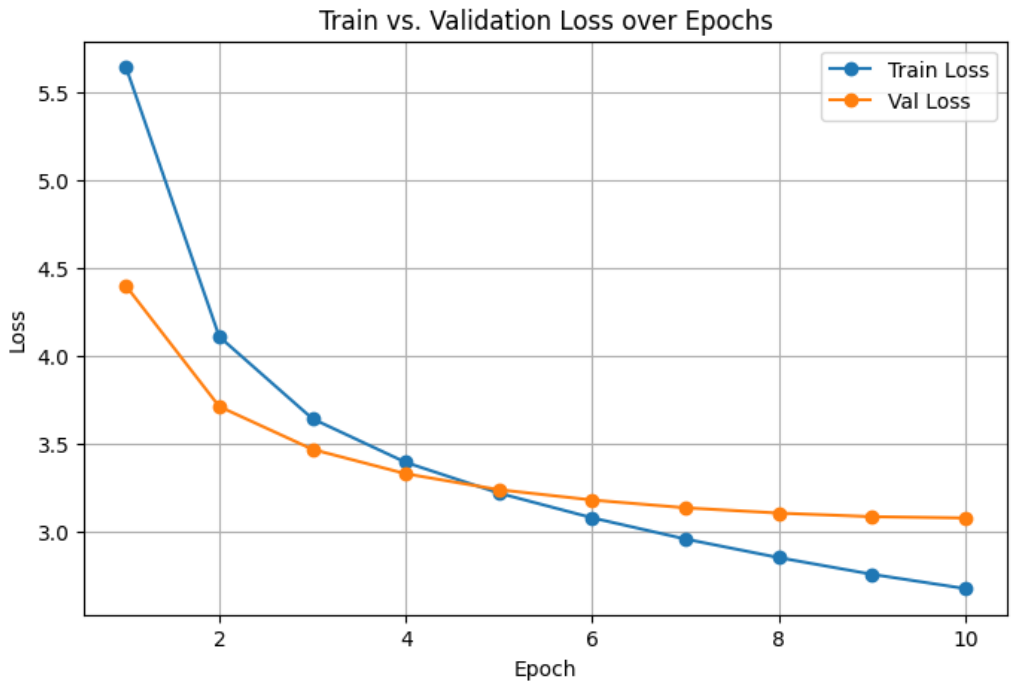

Figure 8: Train and Validation loss for approach 2(with Custom Tokenizer)

ROUGE Scores.

Epoch	ROUGE-1	ROUGE-2	ROUGE-L
5	0.278962	0.0759262	0.215716
10	0.296736	0.0881564	0.230208

4.3 Final Remarks

Both approaches demonstrated solid training behavior and yielded quality summaries. Approach 1, which uses a pretrained tokenizer, slightly outperformed the

custom tokenizer in terms of ROUGE scores and long-term learning stability. However, the difference was not drastic, indicating that a carefully trained tokenizer with a smaller vocabulary can still provide competitive performance.

Beam search decoding notably enhanced output quality compared to greedy decoding, producing more coherent and contextually relevant summaries.

4.4 Contributions of members

Member	Contribution
Subha Chakraborty and Akash Gorakh Chaudhari	EDA and Report
Akash Gorakh Chaudhari and Subha Chakraborty	Transformer Architecture
Omkar Satav and Harshal Gujrathi	Decoder Approaches